Contents

Omnis Programming

Omnis Software Ltd

Released May 2023
Updated Jun 2023 Revision 35439
Updated Oct 2023 Revision 35659
Updated Mar 2024 Revision 36251

About This Manual

This manual starts with a general introduction to the Omnis Environment and goes onto to describe Libraries and Classes. It then focuses on all aspects of Programming and Debugging code in Omnis Studio, including information about creating and using List Variables, accessing and managing SQL databases, as well as describing the Omnis VCS. To learn more about creating web and mobile applications, you need to refer to the Creating Web & Mobile Apps manual.

In addition, there are the *Omnis Reference* manuals containing information about all the Commands and Functions available in Omnis Studio, plus there is a comprehensive **Help** system, available from within the Omnis IDE using the **F1** key, which contains a complete list of all Omnis Notation including all properties and methods.

The majority of the information in this manual is relevant if you are using the *Community Edition,* except the chapters on Window Classes and Window Components which are used for creating desktop apps and are therefore not available in the Community Edition.

If you are new to Omnis

When you start Omnis Studio you will see the **Studio Browser** (if this is not visible press F2 on Windows or Cmnd-2 on macOS) which allows you to create a new project library or open an existing library. Under the **Hub** section, you can look at example Omnis applications under the **Applets** and **Samples** options: you can open each example in your web browser or within Omnis itself, and you can examine the Omnis code in the associated library under the **Project Libraries** option in the Studio Browser. Some of the example code in this manual is taken from the example applications in the Hub.

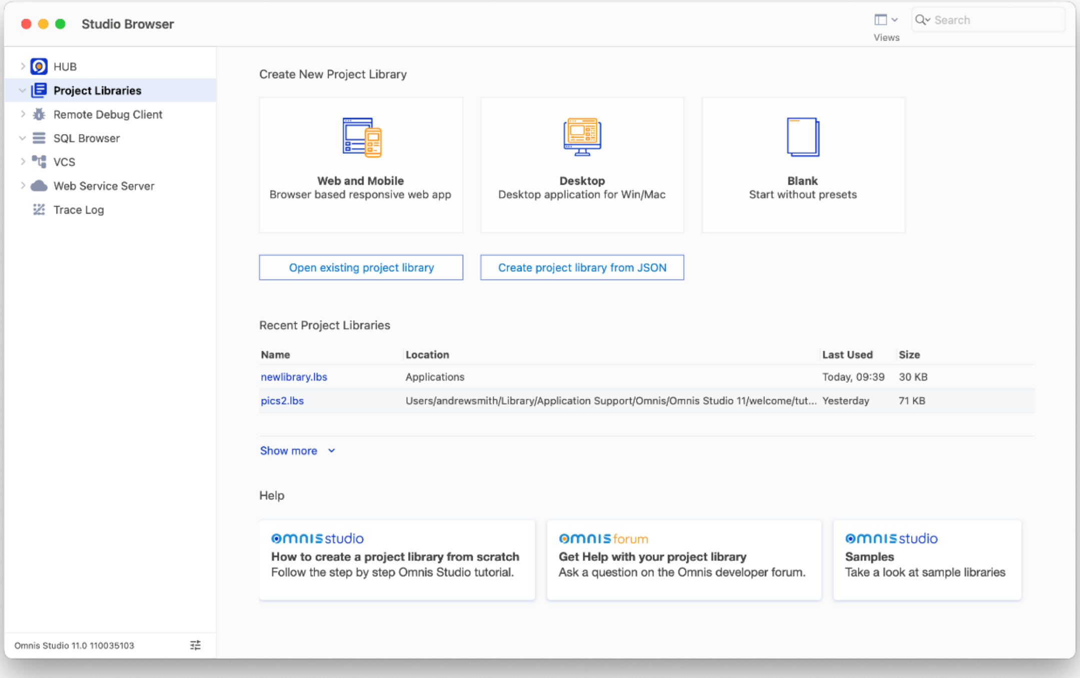
In addition to looking at the example applications in the **Hub,** you may like to work through the Tutorial which covers all the main tasks in creating an application in Omnis Studio, including creating SQL data classes, logging onto a SQL database, and building remote forms.

Copyright info

Chapter 1—The Omnis Environment

The Omnis development environment contains all the tools you need to build enterprise, web, and mobile applications in a cross-platform, multi-developer environment. The Tutorial introduces you to some parts of the Omnis IDE, including the **Studio Browser, Property Manager** and the **Catalog.** This chapter goes into more detail about these tools and others in the Omnis Studio IDE. Some of the tools and development features in Omnis, such as the Omnis VCS and using the SQL Browser, are described in greater detail in their own respective chapters later in this manual.

When you start Omnis Studio you will see the **Project Libraries** option in the Studio Browser, which allows you to create a new library or open an existing library.


Studio Browser
Views
Search
HUB
Project Libraries
Remote Debug Client
SQL Browser
VCS
Web Service Server
Trace Log
Omnis Studio 11.0 110035103
Create New Project Library
Web and Mobile
Browser based responsive web app
Desktop
Desktop application for Win/Mac
Blank
Start without presets
Open existing project library
Create project library from JSON
Recent Project Libraries
Show more
Help
omnis studio
How to create a project library from scratch
Follow the step by step Omnis Studio tutorial.
omnis forum
Get Help with your project library
Ask a question on the Omnis developer forum.
omnis studio
Samples
Take a look at sample libraries

Name	Location	Last Used	Size
newlibrary.lbs	Applications	Today, 09:39	30 KB
pics2.lbs	Users/andrewsmith/Library/Application Support/Omnis/Omnis Studio 11/welcome/tut...	Yesterday	71 KB

If the **Studio Browser** window is not visible, you can press the **F2** key on Windows, or **Cmnd-2** (or **Fn F2**) on macOS, or you can select the **Browser** option from the **View** menu, or under Windows you can click on the **Browser** button (compass icon) on the main Omnis toolbar (on macOS, you can enable the toolbars via the View>>Toolbars option).

The list of options down the left-hand side of the Studio Browser gives you access to all the main tools for creating applications, including the **SQL Browser,** the Omnis **VCS,** and the **HUB** which contains many example apps under the **Samples** option (see below for more information about these options).

Note that some features (and class types) in the Studio Browser are not available in the Community Edition, such as those relevant to developing desktop applications.

Studio Browser

The **Studio Browser** is the main window in Omnis Studio for developing your applications and managing server database sessions. You can use the **Views** droplist on the window toolbar (title bar on macOS) to set the main view: this can be the **Details** view (the default), **Large Icons,** or **Small Icons.** The following screen shot shows the **Studio Browser** with the contents of a library in Large Icons view.

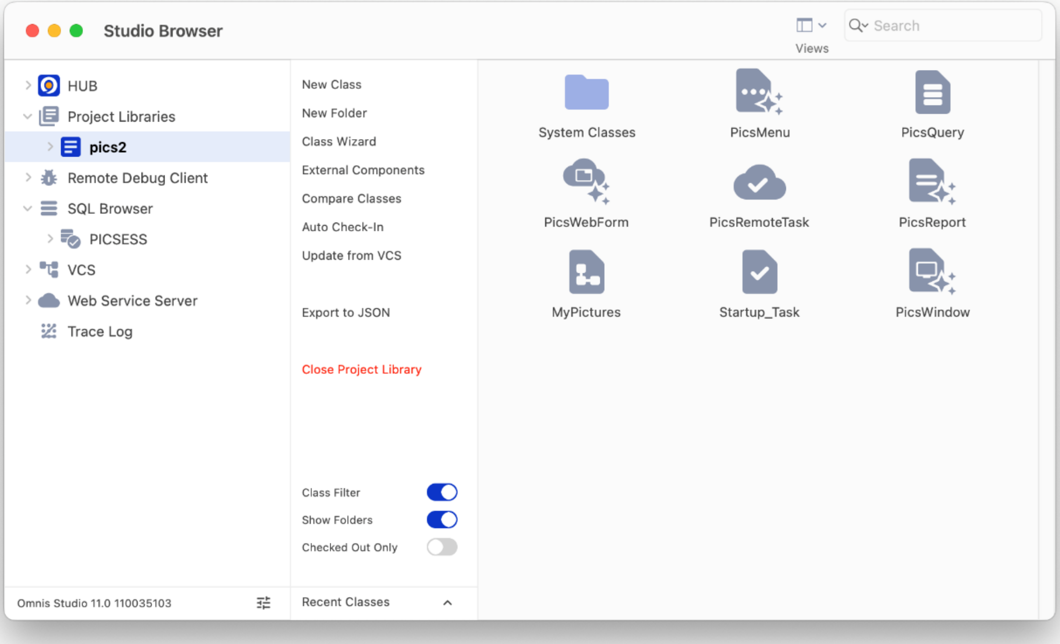


Figure 1:

You can Right-click in the Studio Browser window to open its context menu, which contains options to set **Single Window Mode,** show or hide **Library Folders,** plus the **Save Window Setup** option lets you save the current settings including the size and position of the Studio Browser window. The context menu also lets you **Arrange Icons by** various criteria (including class Type or Name), plus the **New** option lets you create a new class or folder.

The hierarchical tree list down the left side of the Studio Browser contains the following options:

- **HUB**
  contains example libraries, fault reports, and options to configure the Omnis IDE; see Developer Hub
- **Project Libraries**
  allows you to open a library, or create a new one, and shows a list of all open libraries in your project or application; see Libraries and Classes
- **Remote Debug Client**
  allows you to debug your application on a remote server; see Remote Debugger
- **SQL Browser**
  allows you to open a SQL database session, manage server tables, and access your data on a remote database server; see SQL Browser
- **VCS**
  allows you to manage a project using the Omnis VCS; see the Version Control chapter; not available in the Community Edition
- **Web Service Server**
  allows you to setup Omnis to access third-party Web Services, or to setup your own web service using Omnis server code; see Web Services

- **Trace Log**
  allows you to trace and debug your Omnis code; see Debugging Methods

The right-hand pane in the Studio Browser displays a list of objects or files for the current folder or selected object under the main tree list on the left, e.g. it displays all the classes in a library when a library is selected in the tree list. The list of **Options** (list of hyperlinks in the center) will change depending on the object currently selected in the tree list or the file browser on the right and will always provide options and shortcuts to perform actions on the current object. For example, when a library is selected on the left, its classes are shown in the list on the right, and the hyperlink options relate to classes, such as creating a **New Class, New Folder,** or **Class Wizard.** The **Class Filter** option allows you to show or hide certain types of classes.

Search Filter

The Studio Browser has a **Search** box that allows you to filter the objects displayed in the library or class list allowing you to find objects more easily. The Search filter is available for most of views in the Studio Browser, including Libraries, Classes, SQL sessions, VCS projects, and various parts of the Hub including the Sample apps and Faults; on Windows, the search box is on the left under the window title, but on newer macOS systems, the toolbar is integrated into the window title and the Search box may be over to the right (as shown).

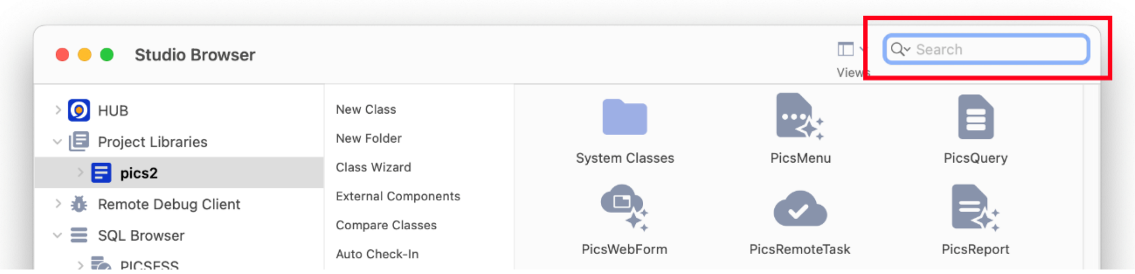


Figure 2:

To search for an item, navigate to the correct view in the Studio Browser (e.g. the class list view), type one or more characters, and the list will instantly redraw, displaying only those items that *contain* the character(s) you typed. For example, in the class list for a library, you could type “task” to find all the classes containing “task”, as shown:

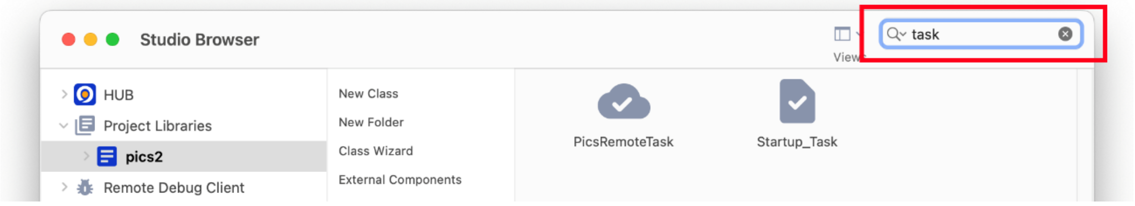


Figure 3:

The Search box has a dropdown list that stores the last few searches you typed, which you can select from with the pointer.

Copy Class Name

You can copy the name of a selected class to clipboard by pressing Ctrl-N or via the **Copy Name** option on the Class Browser context menu: for multiple selected classes the names are copied in a list.

Developer Hub

The **HUB** option in the Studio Browser provides useful information for developers, such as the status of the most recent reported and fixed faults, together with information and tips for new Omnis developers. The HUB option itself contains information and news about Omnis Studio, including where to get help and online training, plus links to the Omnis Twitter feed.

Applets and Samples

The **Applets** option provides a number of example Omnis applications that show the full capabilities of Omnis Studio for building web and mobile applications. You can open each of the examples in a web browser (when you select an example it is opened in your browser automatically), and you can examine the code in the associated library under the **Project Libraries** option in the Studio Browser.

The **Samples** option provides a large range of sample Omnis libraries demonstrating specific components or programming techniques in Omnis. Once you have opened the sample library, you can examine its classes and underlying code under the **Project Libraries** option in the Studio Browser. When the Samples option is selected in the Studio Browser, you can use the Search option (at the top) to find specific examples, e.g. type 'list' to find all list examples. You can use the Examples filter to hide or show categories or types of examples, including a New option for showing any examples added in the latest major release of Omnis Studio.

You can use the Omnis libraries under the Applets and Samples option as templates for your own libraries, or you can reuse individual classes or the Omnis code within the libraries.

Faults

The **Faults** option provides information about the latest *Reported* and *Fixed* faults in Omnis Studio – this is real-time information so you can check the most recent faults. If you have reported a fault in Omnis Studio you can check its status here.

IDE Options

The **IDE Options** allows you to configure the behavior, contents and appearance of the Studio Browser. The Options window is split into different tab sections:

Browser tab

- **Show These Tools**
  specifies which tools (nodes) are displayed in the Studio Browser. By default, the SQL Browser, Trace Log, VCS, and Web Services are enabled. (The Datafiles option is disabled by default and is hidden in some editions of Omnis)
- **Default Browser Node**
  specifies whether the **Hub** or **Project Libraries** node is displayed by default when Omnis starts up.

Themes tab

The theme settings under this tab relate to color themes used in the Omnis IDE or desktop window classes (not the JS Themes which are used to manage the colors in web and mobile apps).

- **Appearance Themes**
  The Appearance option allows you to change the theme used in the Omnis Studio IDE. Together with the Default theme, there are several other themes from which you can choose. See Color Themes and Appearance. You can create a custom theme, as well as import or export themes from this window. (Note the themes available in the IDE are not the same as those available for remote forms.)
- **Window Frame Theme**
  (Windows only) The 'Window Frame Theme' option allows you to set the color theme or style for the frame edge of windows and forms. The options are Default, Windows 7, 8, or 10.

Proxy Server tab

The 'Use proxy server' option allows you to access the faults information if you use a proxy for outgoing requests. Check the option and enter a hostname and service or port for your proxy server.

Project Libraries in the Studio Browser

The **Project Libraries** option under the Studio Browser lets you create a new library, the main file where you store all the classes in your application, or open an existing library. See the Libraries and Classes chapter for more information about creating and opening libraries in the Studio Browser.

The **Create project library from JSON** option allows you to import a library from a JSON representation of an Omnis library (you can download JSON files from our GitHub). See Importing Libraries in the Libraries and Classes chapter for more information.

Library Conversion

When you open a library created in a previous version of Omnis Studio it will be converted and can no longer be opened in the old version of Omnis Studio – *THE LIBRARY CONVERSION PROCESS IS IRREVERSIBLE. Therefore, and in all cases, YOU SHOULD MAKE A SECURE BACKUP of all old Omnis libraries and datafiles BEFORE OPENING THEM in the latest version of Omnis Studio.*

Class Locking and Library Conversion

In order to enhance the integrity and security of deployed Omnis Studio libraries, the mechanism used to lock classes in a private library has changed in Omnis Studio Revision 35659.

Consequently, all libraries opened in Omnis Studio 11 revision 35659 *WILL REQUIRE CONVERSION, INCLUDING LIBRARIES CREATED WITH ALL PRIOR REVISIONS OF OMNIS STUDIO 11* (as well as Studio 10 or earlier libraries). *THE LIBRARY CONVERSION PROCESS IS IRREVERSIBLE.*

THEREFORE, AND IN ALL CASES, YOU SHOULD MAKE A SECURE BACKUP of all existing Omnis Studio 11 libraries BEFORE OPENING THEM in Omnis Studio 11 Revision 35659.

Recent Libraries or Classes

When the **Project Libraries** option is selected in the Studio Browser, the **Recent Project Libraries** option (half way down the Studio Browser window) allows you to open a library that you have previously opened; this is a handy shortcut and saves you having to navigate to your library.

When a ***library*** is selected in the Studio Browser, the **Recent Classes** option (at the bottom of the Studio Browser window) allows you to open any classes that you have previously opened. You can open the method editor for a class (that is, a class that can contain methods) from the Recent Classes list by holding down the **Shift** key and selecting the class. To open the Interface Manager for a class, hold down the **Control/Command** key while selecting the class in the Recent classes list.

Library Folders

You can create folders in a library so you can organize the classes within each library. Folders are in fact Omnis classes, but for display only (they are visible during development only). Folders do not perform any function in your library, other than organizing your classes, and they are ignored at runtime.

To hide/show library folders within the Browser, select your library and click on the 'Library Folders (on/off)' option, or alternatively, right-click on the white space of the Studio Browser (when any library is selected) and select the 'Library Folders' option from the context menu. You can save the state of this option using the **Save Window Setup** option for the Studio Browser.

Hiding and Showing Classes

When the contents of a library are displayed in the Studio Browser, you can hide and show different types of class by pressing the Shift and Control/Command keys plus the appropriate letter key. For example, pressing Shift+Ctrl+I displays only the Remote Forms in the library, or Shift-Ctrl/Cmnd-A shows all classes. The following keyboard shortcuts are available:

Class type	Shortcut - Shift + Ctrl/Cmnd + Letter key
Shows all	A
Code	C
File	L
Menu	M
Object	O
Query	Y
Remote form	I
Remote menu	N
Remote object	E
Remote task	K
Report	R
Schema	S
Search	H
Table	B
Task	J
Toolbar	T
User constants	U
Window	W

The above key presses activate the Class Filter, which you can open and set manually by clicking on the **Class Filter (on/off)** option in the Studio Browser; you can check or uncheck options to show or hide specific class types.

Note that class types relevant to developing desktop applications are hidden in the Community Edition. Note also that when you show all classes using the Shift-Ctrl/Cmnd-A option, the #PASSWORDS system table is displayed and is only relevant to desktop apps and should not be used for web or mobile apps.

Diacritical Character input in the IDE

The Studio IDE will provide the diacritical character popup to allow you to enter diacritical characters wherever text entry is required. For example, you can enter diacritical characters in comments in the Method Editor or into a label on a form. To disable the feature for the IDE, remove or rename the Keyboard folder which is in the local folder.

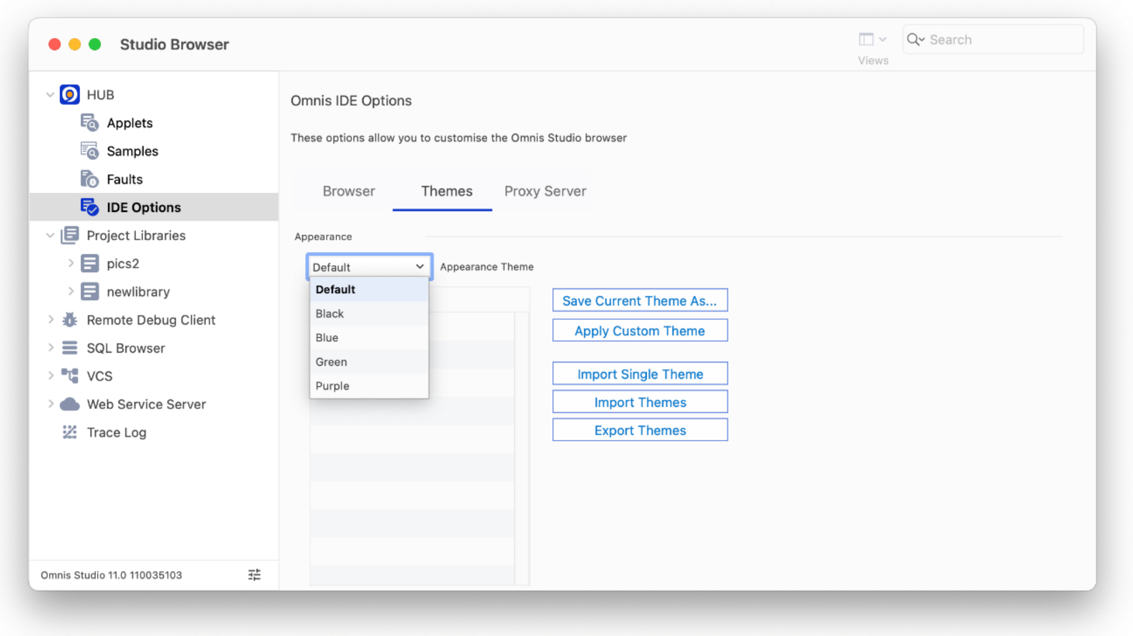
Using Multiple Screens on macOS

You can move any of the design tools, such as the Property Manager and Method Editor, or any class editor window, to another screen using the **Move Top To <screen>** command in the Window menu (macOS only – 10.15 Catalina and later). The option will only appear when there is more than one screen connected to your Mac computer, and in this case will move the top window to the named additional screen.

Color Themes and Appearance

The colors used throughout the Omnis Studio design environment (the IDE) are stored as a *design theme* which can be changed under the **IDE Options>>Themes** setting in the Hub in the Studio Browser: each theme contains a list of color settings for the objects and tools in the Omnis Studio IDE. Color themes are available on Windows and macOS only, but not for Linux. (Note the design themes available in the Omnis IDE are not the same as JS Themes available for Remote Forms.)

There are a number of themes for the Omnis IDE and available in the Hub, listed under the 'Appearance Theme' droplist, including the 'Default' theme which is intended to match the colors used on different platforms supported in Omnis.



Omnis IDE Options

These options allow you to customise the Omnis Studio browser

Appearance Theme: Default

Figure 4:

You can change individual colors or settings within a theme (under the $appearance property, see below), and in this case, your modified theme will be saved as a 'Custom' theme alongside those provided. The themes for the IDE are stored as configuration files in the 'Studio' folder under the main Omnis folder.

Exporting & Importing Custom Themes

You can create multiple custom themes, and you can export and import themes. There is a list of the custom themes currently installed (located in the folder /studio/themes/custom) underneath the themes droplist.

To create a custom theme, press the "Save Current Theme As" button. Once saved, the name you give the theme will then appear in the list of custom themes.

If you are setting a custom theme, you will need to select it first in the list and then press the "Apply Custom Theme" button, since you need to be able to select a custom theme without applying the theme when exporting.

To export custom themes, select the required themes in the list and press the "Export Themes" button. This allows you to select a folder to copy the themes into.

You can also import either a single theme or a folder of themes. Once imported they are copied to the /studio/themes/custom folder and will appear in the list.

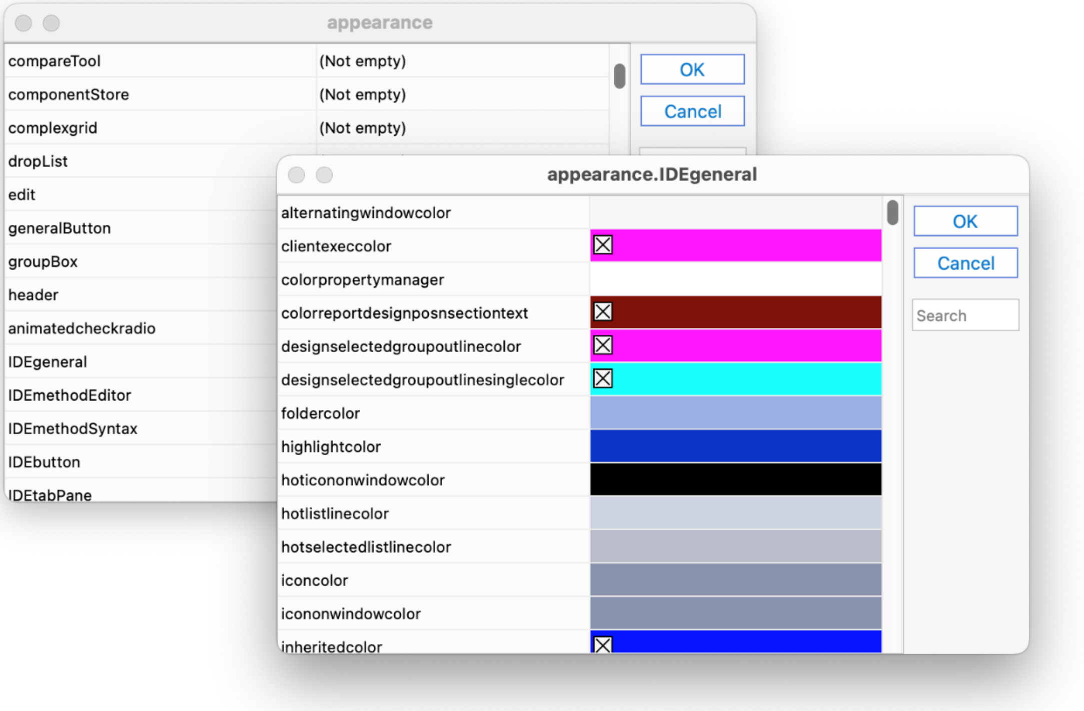
Window Frame Themes

For Windows only, you can also change the theme used for the outside edge or frame of windows used in the Studio IDE and your own windows, which can be set to match the style of the window frame displayed on various Windows platforms. These are shown under the ‘Window Frame Theme’ droplist.

Appearance Property

The current theme in the Omnis Studio IDE is stored in an Omnis preference called $appearance, which can be edited via the Property Manager. To edit this property, click on the Options button at the bottom-left of the Studio Browser, select the **Preferences** option, then select the **Appearance** tab in the Property Manager (if the Advanced option is on) and click on the droplist next to the **$appearance** property; alternatively, you can use the Search box to find $appearance in the Property Manager.

Note that when editing $appearance in the Property Manager, the default colors may not always draw correctly since the editor grid itself uses the configured colors, and therefore if a color is set to the default setting, there is a check mark icon (an X) to the left of the color.



compareTool	(Not empty)
componentStore	(Not empty)
complexgrid	(Not empty)
dropList	
edit	
generalButton	
groupBox	
header	
animatedcheckradio	
IDEgeneral	
IDEmethodEditor	
IDEmethodSyntax	
IDEbutton	
IDEtabPane	

alternatingwindowcolor	
clientexeccolor	☒ 🟪
colorpropertymanager	⬜
colorreportdesignposnsectiontext	☒ 🟥
designselectedgroupoutlinecolor	☒ 🟪
designselectedgroupoutlinesinglecolor	☒ 🟦
foldercolor	🟦
highlightcolor	🟦
hoticononwindowcolor	⬛
hotlistlinecolor	⬜
hotselectedlistlinecolor	⬜
iconcolor	⬜
icononwindowcolor	⬜
inheritedcolor	☒ 🟦

Figure 5:

In addition, there is an Omnis preference called $windowoptions that stores the current Window Frame theme which you can also edit via the Property Manager: this is described at the end of this section.

Searching Colors & Themes

There is a search field on the $appearance preference dialog to help you find colors. As you type into the search box, Omnis will highlight any matching lines in the Property Manager and scroll to the first match. Tabbing from the search field sets the focus in the grid to the first match.

Appearance and Theme Files

The color and appearance settings used in the Omnis Studio IDE, and displayed in $appearance via the Property Manager, are stored in a configuration file called **‘appearance.json’,** which is located in the Studio folder in the main Omnis folder. This file contains the current color theme settings in $appearance which will either be the default theme, or one of the other themes provided, or a custom theme.

The Default and other themes (Blue, Green, etc) are stored in separate JSON files in the ‘studio/themes’ folder. As you change the theme setting under the Options setting in the Hub, the appropriate JSON theme file is written to the appearance.json file in the Studio folder.

When Omnis loads, it will copy the appropriate theme file into /studio/appearance.json. If you alter one of the theme colors using $appearance in the Property Manager, Omnis will recognize that the default or one of the built-in themes has changed and therefore will create a custom theme 'appthemecustom.json' in the themes folder. In this case, when Omnis restarts, it will load the Custom theme and it will add it to the droplist in the Options setting in the Hub.

When a theme has been changed, if you then try to switch to another theme using the Hub, you will receive a Yes/No message asking you to confirm if you wish to overwrite the custom changes.

Appearance File Contents & Help

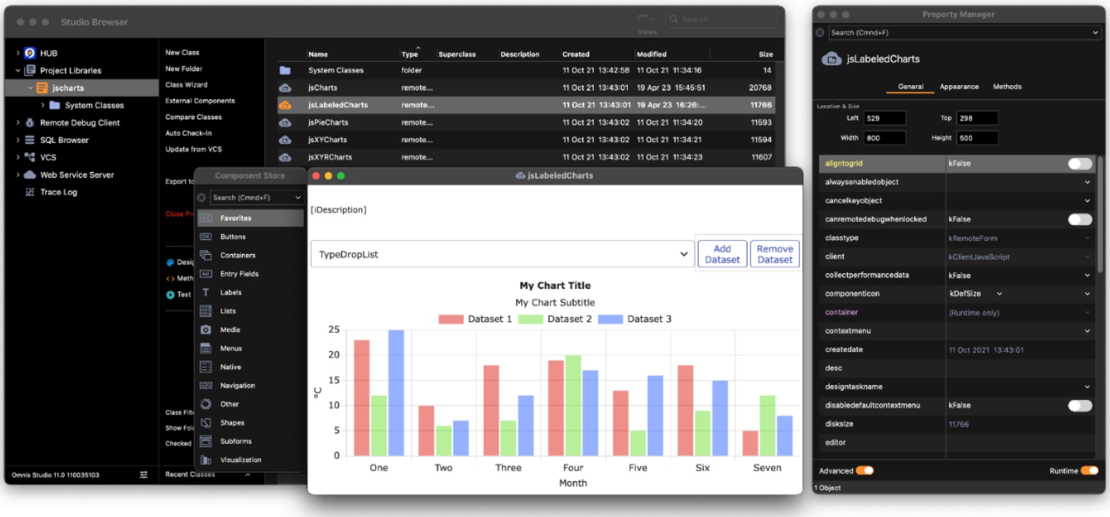
There is a file called **appearance.desc.en.json** in the 'local' folder under the main Omnis folder which contains a description of all the items in the $appearance property and appearance.json file: the descriptions in this file are used as helptips in the Property Manager when viewing the items in the $appearance property.

Colors are stored in appearance.json as a string, which can be either "#RRGGBB" or "kColorDefault" or one of the 16 standard colours: kBlack, kDarkBlue, kDarkGreen, kDarkCyan, kDarkRed, kDarkMagenta, kDarkYellow, kDarkGray, kBlue, kGreen, kCyan, kRed, kMagenta, kYellow, kGray, or kWhite.

The subgroups prefixed with **IDE** refer to colors in the IDE only, such as the Method Editor and Method Syntax, so any changes you make to colors in these groups are only visible in the IDE, not the Runtime. All other theme subgroups affect colors in the IDE and the Runtime version of Omnis Studio. When you edit the $appearance preference in the Property Manager you will see the subgroups, so to edit colors in the Method Editor syntax you can open the "IDEmethodSyntax" group.

Dark Mode

Omnis Studio supports Dark and Light modes when using *the default Omnis design theme* (studio/themes/appthemedefault.json); the theme for design mode in Omnis is set via the **Themes** tab under the **IDE Options** option in the Hub in the Studio Browser. You can change the system color mode via the **System Preferences > General** option on macOS, or **Settings > Personalization > Colors** option under Windows. Note dark mode is supported on macOS 10.14 and later or Windows 10/11 or above.



Name	Type	Superclass	Description	Created	Modified	Size
System Classes	folder			11 Oct 21 13:42:58	11 Oct 21 11:34:16	14
jsCharts	remote...			11 Oct 21 13:43:01	19 Apr 23 15:45:51	20768
jsLabeledCharts	remote...			11 Oct 21 13:43:01	19 Apr 23 16:26:...	11766
jsPieCharts	remote...			11 Oct 21 13:43:02	11 Oct 21 11:34:20	11593
jsXYCharts	remote...			11 Oct 21 13:43:02	11 Oct 21 11:34:21	11594
jsXYRCharts	remote...			11 Oct 21 13:43:02	11 Oct 21 11:34:23	11607

Location & Size
Left 529 Top 298
Width 800 Height 500

aligntogrid	kFalse
alwaysenabledobject	
cancelkeyobject	
canremotedebugwhenlocked	kFalse
classtype	kRemoteForm
client	kClientJavaScript
collectperformancedata	kFalse
componenticon	kDefSize
container	(Runtime only)
contextmenu	
createdate	11 Oct 2021 13:43:01
desc	
designtaskname	
disabledefaultcontextmenu	kFalse
disksize	11766
editor	

Figure 6:

Dark mode is supported in the theme and appearance files using "[item].dark" items. For example, as well as a "tree" item in appearance.json, there is a "tree.dark" item which is used when the system is in dark mode; if there is no ".dark" entry, the normal entry is used in dark mode.

If an appearance.json file does not contain any ".dark" entries, Omnis will use the light system theme when determining any defaults that come from the system, although system dialogs will display in the current mode for the system.

User Defined Colors

User defined colors can be added to the appearance.json which can be used for theme colors for window class controls in desktop (fat client) apps. The colors are defined using the groups "user" and "user.dark" in the appearance.json theme file, using the names **color1** to **color16.** They are represented by 16 new color constants **kColorUser1** to **kColorUser16.**

IDE Window Colors (Windows only)

You can specify dark mode colors for some of the IDE window colors defined in the $windowoptions Omnis preference; these only apply on Windows OS and are used automatically when dark mode is being used. The following colors can be defined:

```
titleactivecolor.dark
titleinactivecolor.dark
smalltitleactivecolor.dark
smalltitleinactivecolor.dark
borderactivecolor.dark
borderinactivecolor.dark
captionactivecolor.dark
captioninactivecolor.dark
smallcaptionactivecolor.dark
smallcaptioninactivecolor.dark
minmaxbuttonhotcolor.dark
minmaxbuttonhottrackingcolor.dark
closebuttonhotcolor.dark
closebuttonhottrackingcolor.dark
```

Platform Specific Notes

System dialogs and Menus

System dialogs (file dialogs etc) do not use the theme colors.

Menus on macOS do not use the theme colors. On Windows, the colorlistlineselectedwin color is used to highlight menu lines.

JavaScript Client

The JavaScript client will use the system colors configured on the Omnis App Server running your app. Therefore if you want to use a specific theme for your deployed web and mobile apps, you need to copy your appearance.json file to the Omnis App Server.

macOS and Cocoa

The term system theme is a little loose for Cocoa. Omnis tries to match the system theme, but unlike Windows, there are not always APIs in the OS to perform the drawing (hence some system theme drawing on Cocoa is actually Omnis imitating the OS theme).

Some of the theme background colors have been rationalized (this does not apply to Carbon) as follows:

kBGThemeWindow, kBGThemeContainer and kBGThemeTabStrip now fill with kColorWindow on both platforms. Previously these used kColor3DFace on Windows.

kBGThemeTabPane now fills with the selected tab background (using either the system theme or colortabselected - the system theme applies with colortabselected is kColorDefault).

Window Frame Appearance on Windows

In addition to the color management outlined above, you can change the appearance of the frame edge of window classes in desktop apps (on Windows operating systems only), which allows you to comply with the latest style for window frame edges on Windows 8, 8.1 and 10. For most purposes you can accept the default settings for the current Windows platform, but you can change the frame theme if you want.

You can change window frame themes under the **IDE Options** setting under the **Hub** section of the Studio Browser (Windows only). You can select Default, Windows 7, 8, or 10 which allows you to view how your application will appear on different Windows platforms.

In addition there is a new property in the Omnis preferences, $windowoptions, which allows you to edit the appearance of window frames in libraries running on Windows – note this preference is only editable on the Windows platform.

Window Frame Configuration files

There is a file called ‘window.json’ in the Studio folder, which stores the values of the $windowoptions preference. The window.json file configures the appearance of the window frame, and also configures the operating systems for which the configured appearance will be used, including the old appearance for Windows 7, and the new configured appearance for later Windows operating systems. There are a number of theme files in the ‘studio/themes’ folder which are copied to window.json as appropriate.

Active Caption Colors

useborderactivecolorfordefaultactivecaption
specifies the color of the active caption in Windows (stored in window.json). This is an integer with 3 possible values.

If it is zero, the behavior is as before (the active caption defaults to white if it is set tokColorDefault).

It it is one, and the active caption color is kColorDefault, then the active caption color is the same as the active border color.

If it is two (the default), then the default active caption color depends on the system setting on the accent color settings panel: "show colour on start, taskbar, action centre and title bar" - if the system setting is off, then this is equivalent to useborderactivecolorfordefaultactivecaption equal to zero - if the system setting is on, then this is equivalent to useborderactivecolorfordefaultactivecaption equal to one.

Note that this applies to both small and normal size captions, and only applies when the relevant caption colour is kColorDefault.

Omnis Preferences

The ***Omnis Preferences*** control the behavior and appearance of the Omnis Studio IDE, rather than individual libraries or classes, and they can be viewed in the **Property Manager.** The Omnis preferences include groups (tabs) for: **General** options, **Appearance, Devices, Page Setup,** and **Methods.** The Omnis preferences are sometimes referred to as the Omnis "Root Preferences", since they are properties of the $root object in the Omnis object hierarchy, and referenced in the notation as $root.$prefs.<property-name>.

To view the Omnis preferences

- Under Windows, select the **Options** option in the **Tools** menu, or click on the Options button on the **Tools** toolbar (you can enable this option using the **Toolbars** option in the **View** menu)

or

- On macOS, select the **Preferences** option from the **Omnis** menu, or press Cmnd-, (comma)

or

- On all platforms, click on the Options button at the bottom-left of the Studio Browser (next to the version number) and select the **Preferences** option

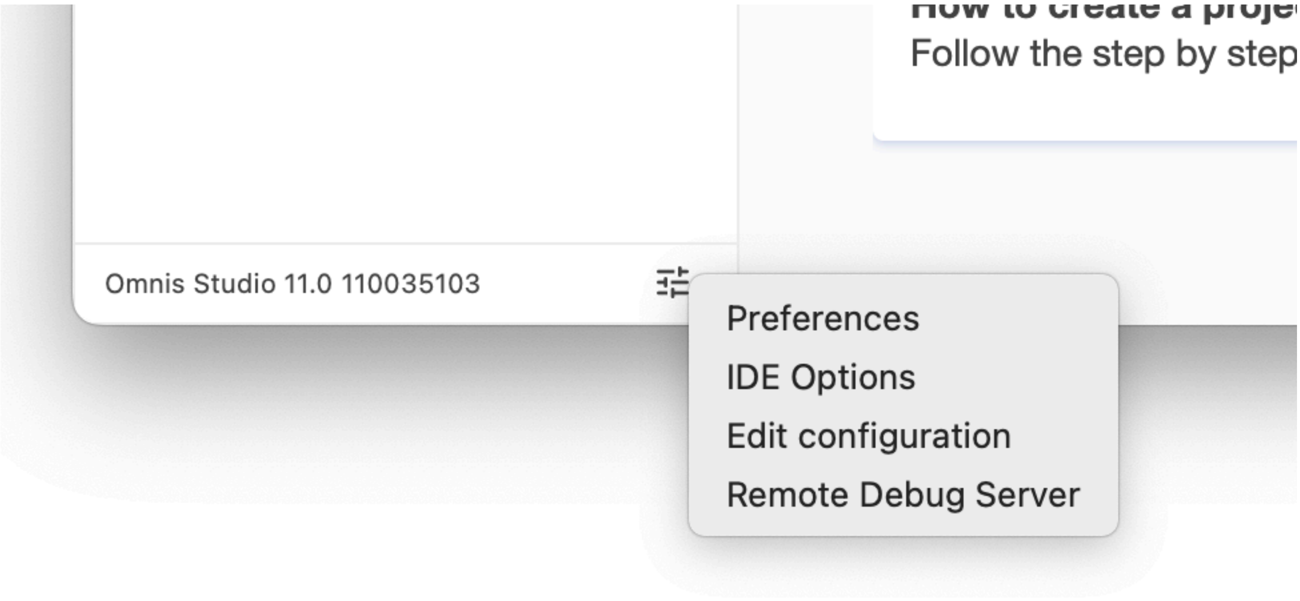
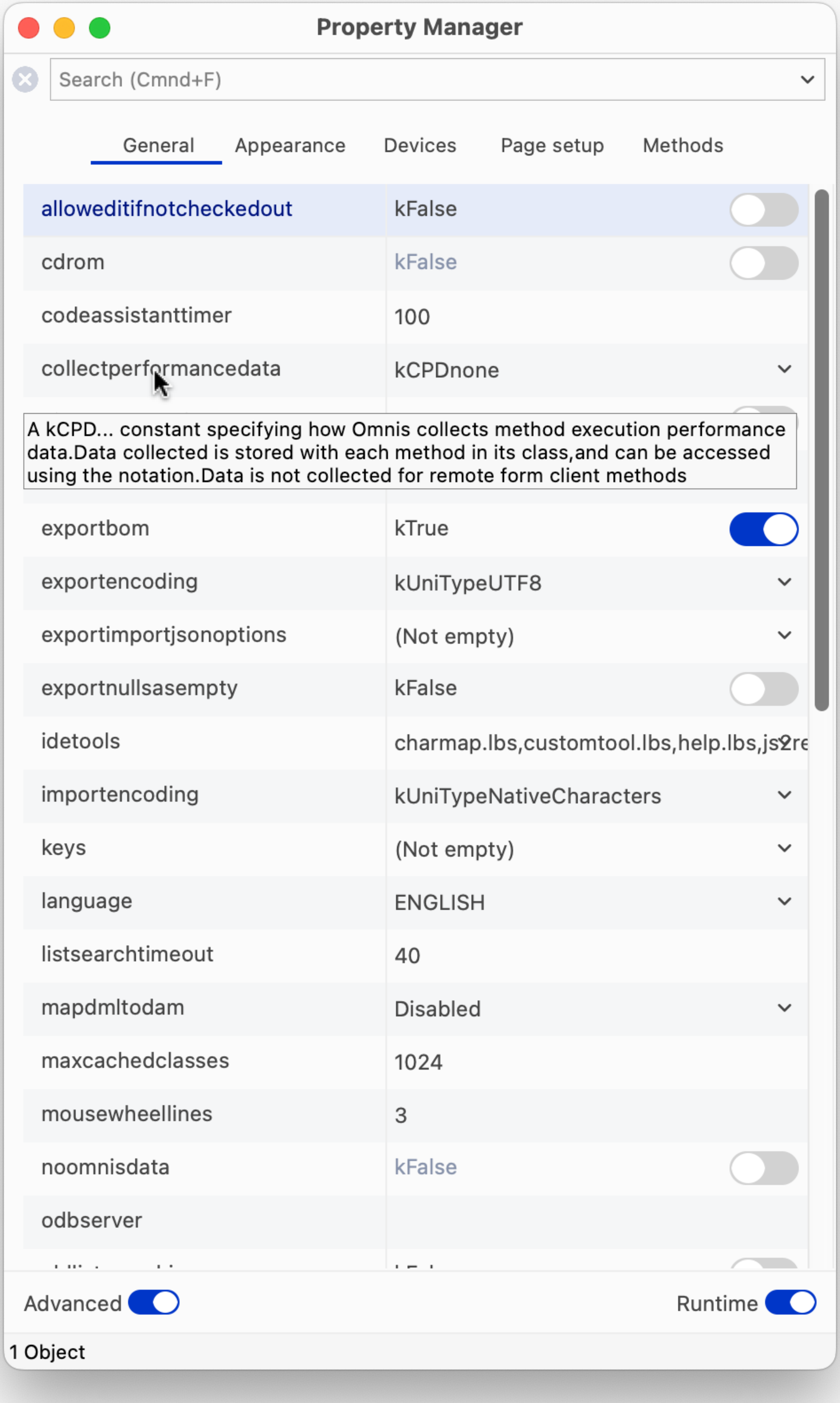


Figure 7:

To view all the Omnis preferences, ensure the **Advanced** option is enabled (on) in the Property Manager; with the Advanced option disabled (off) only a small subset of Omnis preferences is shown. You can position your mouse over any property and view its description in a tooltip (e.g. collectperformancedata is shown and its help tip). If you are looking for a specific property or preference, you can find it using the **Search** box at the top of the Property Manager window; the Search box only appears when the Advanced option is enabled.

You can select a preference in the Property Manager and press the **F1** key (Fn F1 on macOS) to open a **Help** window for that preference (or property); in this case, the page for $root.$prefs opens, listing all the Omnis Preferences.



alloweditifnotcheckedout	kFalse
cdrom	kFalse
codeassistanttimer	100
collectperformancedata	kCPDnone
exportbom	kTrue
exportencoding	kUniTypeUTF8
exportimportjsonoptions	(Not empty)
exportnullsasempty	kFalse
idetools	charmap.lbs,customtool.lbs,help.lbs,js2re
importencoding	kUniTypeNativeCharacters
keys	(Not empty)
language	ENGLISH
listsearchtimeout	40
mapdmltodam	Disabled
maxcachedclasses	1024
mousewheellines	3
noomnisdata	kFalse
odbserver	

A kCPD... constant specifying how Omnis collects method execution performance data.Data collected is stored with each method in its class,and can be accessed using the notation.Data is not collected for remote form client methods

Figure 8:

Preferences in config.json

Some properties in the Omnis Preferences ($root$.prefs) are replicated in the "prefs" group in the Omnis configuration file (config.json), which means you can set their values in either the Property Manager or the Configuration Editor. See Omnis Configuration about editing config.json.

Environment Font Size

The Omnis preference $idelistpointsize specifies the font point size used for lists within the Property Manager and Catalog.

You can increase or decrease the font size used in many of the lists in the Omnis IDE using the **Ctrl+** (increase) or **Ctrl-** (decrease) keyboard shortcuts. These shortcuts are only temporary, are not saved with the window setup, and will return to their respective default sizes when you close Omnis Studio.

Single Instance preference

Under Windows you can run multiple instances of Omnis Studio (this is not allowed under macOS). When the $singleinstance Omnis preference is set to kTrue only one instance of Omnis Studio is allowed. The "singleInstance" item in the "windows" section of the Omnis configuration file can be used to set the value of the Omnis preference $prefs.$singleinstance.

Omnis Configuration

You can control the behavior of the Omnis executable and many other elements of the Omnis IDE by editing a configuration file called **config.json,** *which is created when Omnis is first launched* and is located in the 'Studio' folder under the main Omnis folder. The file is stored in JSON format and should be edited using the Configuration File Editor.

When you start developing your application you may not need to edit the Omnis configuration since it contains all the default settings for running Omnis Studio in development mode. Some options in the Omnis configuration file relate specifically to how the Omnis App Server is setup for deploying and running web or mobile apps: when you deploy your application you will need to edit the Omnis configuration in the Omnis App Server to configure your server settings, such as the server port number.

In addition to the Omnis configuration file (config.json), there is a configuration file called **omnis.cfg** in the 'studio' folder that contains information regarding the Omnis development environment and other internal settings, including specific settings saved with Save Window Setup, e.g. Show tree for the method editor window. This file is created when you first start Omnis and is updated when you shutdown. Note that you cannot edit omnis.cfg.

The **positions.cfg** configuration file, also located in the 'studio' folder, holds the position information saved for various screens in the IDE using the Save Window Setup option. This ensures that the IDE screens are returned to their saved size and position when you reopen Omnis. The information includes window positions and sizes, split bar positions, and so on, for each screen layout. Note that you cannot edit positions.cfg.

Configuration File Editor

The Configuration File Editor allows you to edit the settings in the Omnis Configuration file (config.json) inside Omnis Studio. To open the editor, click on the Options button at the bottom-left of the Studio Browser (next to the revision number), and select the **Edit Configuration** option.

The Configuration Editor shows the main groups of items in the Omnis configuration file in the left hand list, such as 'defaults', 'ide', and 'methodEditor', and for each selected group the items within that group are shown on the right, for example, the **ide** group of items is shown below:

Some items require a string value, in which case you can click on the item and edit it directly in the text field, otherwise, when you click an item to edit it, a droplist may appear containing its possible values (such as True/False values), or some other kind of dialog will open, such as a file select dialog or a color picker.

Configuration Help

When an item is selected, the **Help** panel below the configuration items grid provides a full description of the item. In addition, the status bar beneath the help panel indicates whether or not a restart is required after changing the item and saving the configuration file. The status bar is empty when the item is not relevant to the current platform.

The contents for the Help provided for the Configuration items are stored as HTML pages in a folder called 'confighelp' in the Studio folder; this folder is not present in the Headless Server version.

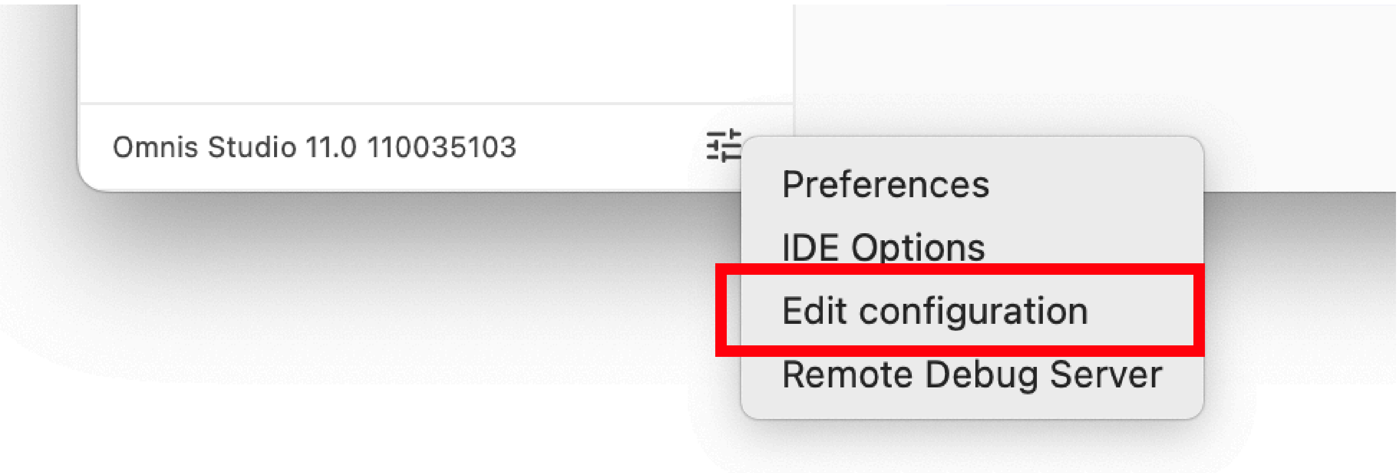
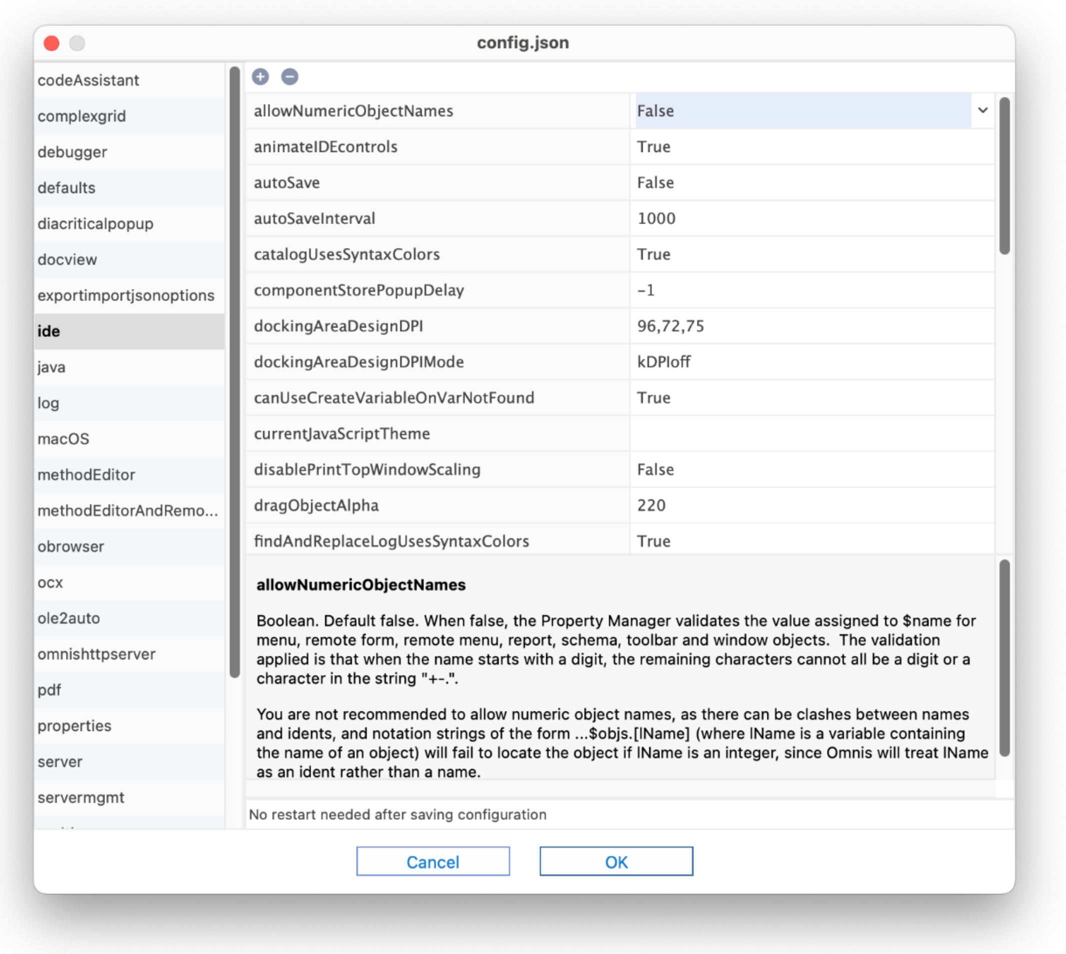


Figure 9:



allowNumericObjectNames	False
animateIDEcontrols	True
autoSave	False
autoSaveInterval	1000
catalogUsesSyntaxColors	True
componentStorePopupDelay	-1
dockingAreaDesignDPI	96,72,75
dockingAreaDesignDPIMode	kDPIoff
canUseCreateVariableOnVarNotFound	True
currentJavaScriptTheme	
disablePrintTopWindowScaling	False
dragObjectAlpha	220
findAndReplaceLogUsesSyntaxColors	True

allowNumericObjectNames

Boolean. Default false. When false, the Property Manager validates the value assigned to $name for menu, remote form, remote menu, report, schema, toolbar and window objects. The validation applied is that when the name starts with a digit, the remaining characters cannot all be a digit or a character in the string "+-.".

You are not recommended to allow numeric object names, as there can be clashes between names and idents, and notation strings of the form ...$objs.[lName] (where lName is a variable containing the name of an object) will fail to locate the object if lName is an integer, since Omnis will treat lName as an ident rather than a name.

No restart needed after saving configuration

Figure 10:

Adding Configuration items

The Omnis configuration file contains all the settings required to run Omnnis Studio in development mode or the Omnis App Server (or Omnis Runtime). However, there may be specific items that are included in the documentation or provided by Technical Support, that are not included in the default Configuration file, which you can add using the Configuration editor.

The + and - icons at the top of the window allows you to add or remove items, however in general however, ***you should not delete items,*** rather just change their values; for example, for a Boolean item value which you want to disable, set the value to False to disable it rather than deleting the item.

To add an item, click on the + icon, enter the exact name of the Config item, and choose the *type,* which is one the following types:

- **Boolean**
  a True or False value
- **Character**
  a string
- **Integer**
  an integer value (usually in a specific range of values), or a constant value
- **List**
  For list items, enter the item name, then you can specify the list of items in a popup window; they are displayed as a comma-separated list

In addition, you can enter \t to mean tab, e.g. for log.conversionLogDelimiter.

Errors in config.json

Errors in the Omnis configuration file config.json are written to the Omnis trace log, which can be viewed from the **Tools** menu or in the **Studio Browser.** As Omnis is loaded, it parses the config.json file and if it fails, an error is written to the trace log.

User Configuration File

Any changes or additions you make to the Omnis Configuration file using the Configuration Editor are saved into a separate file called **userconfig.json,** which is stored in the 'Studio' folder alongside config.json. This ensures that the default settings in config.json are retained and all your changes or additions are stored separately in userconfig.json. This also means that if you upgrade to a newer version of Omnis you can copy across your copy of userconfig.json to ensure all your settings are maintained in the new version.

It is recommended that you *do not edit the config.json file externally using a text file editor,* but you should use the Configuration Editor.

Configuration Editor Visibility

The Configuration file editor is available in the Development version of Omnis Studio, as well as the Runtime and Server versions (but not the Linux Headless server). To open the Configuration file editor in the Runtime or Server version, select the **Edit Configuration…** option from the **File** menu. You can hide this option in the Runtime or Server version by executing the sys(246) function, or sys(247) will show it again; the default setting is for it to be visible.

Windows Configuration

The "windows" section of the config.json file contains settings to control the visual appearance of Omnis or various Startup options when running under Windows.

```
"windows": {
    "highDPIaware": true,
    "readBorderActiveColorFromSystem": true,
    "scaleScreenCoordsUsingPhysicalSize": false,
    "pythonPath": "",
    "miniconid": 2033,
    "hideStudiorgMessage": false,
    "noAdmin": false,
    "updateFileAssociations": true
},
```

The visual appearance settings are:

- **highDPIaware**
  Defaults to true. If true, Omnis will tell Windows to operate in the system DPI, and it will scale pixel dimensions if necessary, by a factor of 2; if false, Omnis will operate at 96dpi
- **readBorderActiveColorFromSystem**
  Defaults to true. If true, Omnis attempts to read the default value for borderactivecolor (a color in window.json) from the operating system. If false, use a hard-coded default rgb(244, 112, 35).
- **scaleScreenCoordsUsingPhysicalSize**
  Defaults to false. Only applies when $clib.$screencoordinates is true. If false, screen coordinate scaling is based on the size of the main window; if true, it is based on the physical screen size.
- **pythonPath**
  Default is empty. Identifies the pathname of the python executable if installed (used for the Python Worker Object). Otherwise, if empty, defaults to python\App\python in the Omnis data folder.
- **miniconid**
  The icon id of the application icon. Default value is 2033.

In addition, you can specify the following Startup options under the windows group:

- **hideStudiorgMessage**
  If true, the message dialog about running Studiorg when Omnis starts up will not be displayed. If false (the default) or omitted, the message is shown.
- **noAdmin**
  If true, Omnis will run with the current user's access level; consequently, it will not attempt to register file associations or event log, and this allows you to run updates (via update.bat) if required. If noAdmin=false Omnis will run as the Admin user (the default behavior, as in previous versions).
- **updateFileAssociation**
  If true (the default), Omnis will attempt to set file associations; if set to false Omnis will not attempt to set file associations.

IDE Animation

Various parts of the Omnis IDE are animated, including the main tree list in the Studio Browser, and the Method names tree list in the Method Editor is animated when you open the editor or redraw the list.

The "animateIDEcontrols" option in the "ide" section of config.json controls whether or not animation is enabled in the IDE: it is set to True by default. Set this to false if you don't want any objects in the IDE to be animated.

Configuration File Methods

There are some methods in the Omnis Preferences that allow you get and set the contents of the Omnis configuration file. These would allow you, for example, to create your own config.json from code which could be used for deployment of your app.

- $getconfigjson()
  Returns config.json as a row (empty if config.json could not be parsed)
- $setconfigjson(wConfigJson)
  Sets config.json to the supplied row

These are methods of $root.$prefs, and they appear on the **Methods** tab of the Property Manager, but only when used with the Notation Inspector.

You can use them to modify existing items, or add new items. For example, the following code adds/modifies some items in the "obrowser" section of the config.json file:

```
Do $prefs.$getconfigjson() Returns cRow
If isnull(cRow.obrowser.$cols.$findname("htmlcontrolsFolder"))
  Do cRow.obrowser.$cols.$add("htmlcontrolsFolder",kCharacter,kSimplechar,1000000)
End If
Calculate cRow.obrowser.htmlcontrolsFolder as "htmlcontrols"
```

```
If isnull(cRow.obrowser.$cols.$findname("clearCacheWhenLoaded"))
  Do cRow.obrowser.$cols.$add("clearCacheWhenLoaded",kBoolean)
End If
Calculate cRow.obrowser.clearCacheWhenLoaded as kTrue
If isnull(cRow.obrowser.$cols.$findname("remoteDebuggingPort"))
  Do cRow.obrowser.$cols.$add("remoteDebuggingPort",kInteger,0)
End If
Calculate cRow.obrowser.remoteDebuggingPort as 5989
Do $prefs.$setconfigjson(cRow)
```

Note that Omnis needs to be restarted after some items in the config.json file have been edited.

Dock Key Events (macOS)

The monitorDockKeyEvents option in the "macOS" section of the config.json file allows you to disable the Keystroke Receiving dialog at startup on macOS. If set to false, Omnis does not attempt to monitor keyboard events from the Dock and the dialog will not be shown. The option is set to true by default for backwards compatibility.

Keystroke Receiving Prompt (macOS)

On macOS, when first running a new version of Omnis Studio, the end user will be presented with a prompt: "Omnis Studio 11 would like to receive keystrokes from any application – Grant access to this application in Privacy & Security settings, located in System Settings."

This is required to provide full keyboard support to Omnis Studio for monitoring events from the macOS Dock and Mission Control. Access can be granted when prompted for Keystroke Receiving, or you can ensure there is an entry granting access in the Input Monitoring section of the Privacy system setting.

To show a one-time only prompt in Omnis Studio, prior to the system prompt, set the "monitorDockKeyEventsInfoPrompt" config entry to true (the default is false). The message can be customized by changing the entry for CORE_RES_19003 in /Contents/Resources/[LOCALE].lproj/Localizable.strings

Keystroke receiving and the access prompt can be disabled by changing the "monitorDockKeyEvents" to false in config.json. When Omnis Studio does not have full keyboard support, the order of windows displayed may not be correct after using Mission Control with the keyboard.

Studio Toolbars and Menus

The main toolbar at the top of the Omnis application window provides access to all the development tools in Omnis, such as the Studio Browser, the Catalog, the Property Manager, and so on. The Standard, View, and Tools toolbars contain many of the same options as the **File, View,** and **Tools** menus, respectively. The **Desktop** toolbar (hidden by default) lets you switch between design and runtime environments.

You can drag any of the toolbars out of the docking area and place them anywhere in the Omnis application window (referred to as "floating"). You can place your pointer over any button to display its name or tooltip description.

The visibility of the main Omnis Toolbar and Menubar is different for Windows and macOS, as follows.

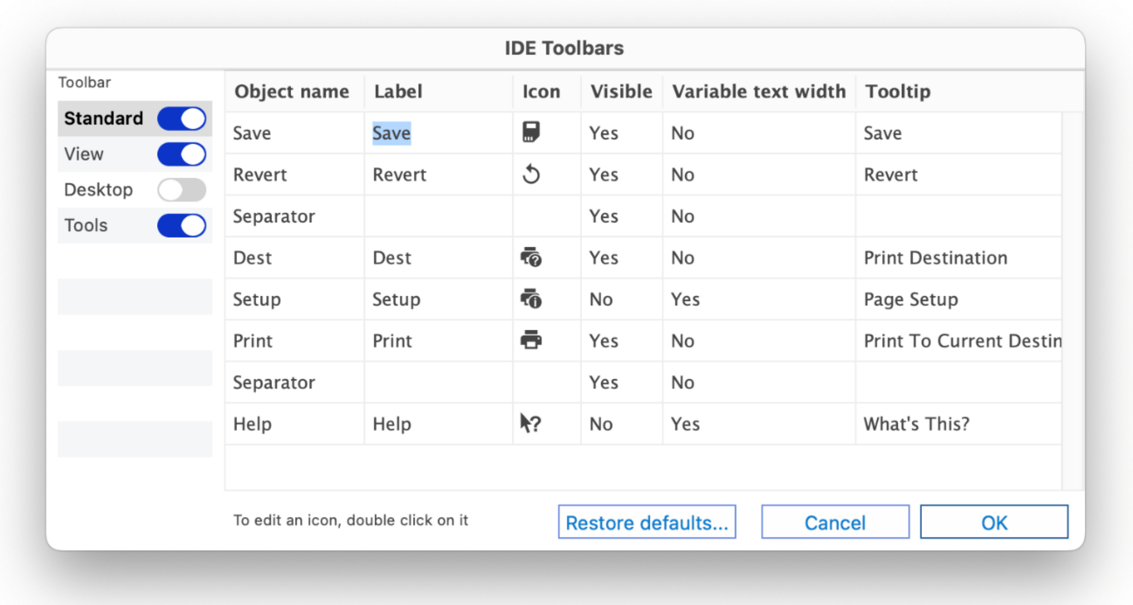
Windows OS

Under Windows, the main Omnis Toolbar is shown, but the main Menubar is not, by default. To hide or show any of the toolbars, or to install the main Omnis menu bar, under Windows, you can Right-click/Ctrl-click in the toolbar area of the main Omnis development window and select the **Toolbar Options…, Toolbars…** (see below) or **Menu Bar** option: in addition, you can double-click in the toolbar area to open the **Toolbar Options,** which allows you to enable Text labels and switch to Large icons. See Omnis Menu Bar.

macOS

On macOS, the main Omnis Menubar is shown by default, but the main Toolbar is not: the concept of an application toolbar is not present in macOS, but you can display the Omnis toolbars manually. To hide or show any of the toolbars on macOS, select the **Toolbars…** option from the **View** menu: this allows you to show or hide the **Standard, View,** or **Tools** toolbars which you may find useful for development.

The **IDE Toolbars** dialog allows you to show or hide the main Omnis toolbars, as well as configure each toolbar, including to show or hide individual buttons.



IDE Toolbars

Toolbar

- Standard ●
- View ●
- Desktop ○
- Tools ●

Object name	Label	Icon	Visible	Variable text width	Tooltip
Save	Save		Yes	No	Save
Revert	Revert		Yes	No	Revert
Separator			Yes	No	
Dest	Dest		Yes	No	Print Destination
Setup	Setup		No	Yes	Page Setup
Print	Print		Yes	No	Print To Current Destin
Separator			Yes	No	
Help	Help		No	Yes	What's This?

To edit an icon, double click on it

Figure 11:

Menus & Timers (macOS)

On macOS, when Omnis starts to track menus, Omnis timers are suspended: in versions prior to Studio 10.2, there was an error whereby Timer execution did not interrupt when using a menu and clicks on the menu were being lost. You can override this behavior, allowing timers to run during tracking process by setting the “menuTrackingSuppressTimers” config.json item in the “macOS” group to false.

Standard toolbar

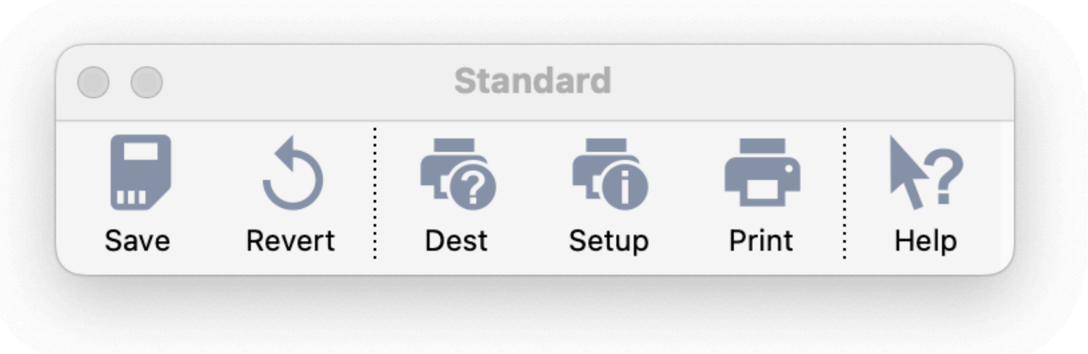


Figure 12:

*Note the screenshots here show all the options in each toolbar; you can enable the buttons in the **Toolbars** option, or change the text options in the **Toolbar Options** option in the **View** menu, or by right-clicking on the main Omnis toolbar.*

The **Standard** toolbar has more-or-less the same options as the **File** menu. In addition, the File menu lets you create a **New Library** (blank other than system classes and Startup_Task) or **Open** an existing library.

The **Save** option (Ctrl/Cmnd-S) saves the class you are currently working on. If a class design window is not currently selected this option is grayed out.

The **Revert** option rolls back any changes you have made to the class you are currently working on. Note that if you close an Omnis class it will be saved automatically and therefore cannot be reverted. The **Revert** menu and toolbar command is not available when Auto Save is enabled.

When the **Auto Save** option in the File menu is enabled, Omnis will save all classes that are currently open in design mode automatically; the option defaults to disabled, meaning that you have to save a class manually or the class is saved when it is closed. The state of the Auto Save option is saved under the “autoSave” option in “ide” section of the config.json configuration file. The interval between each auto save can be configured in the “autoSaveInterval” option, also in the “ide” section of config.json: this is the number of milliseconds between each auto save, which is set to 1000 by default. Auto Save applies to all class and method editors except for system classes, provided that the class is not read-only, and the method editor is not in read-only mode.

The **Dest** option opens the **Print Destination** dialog (Shift-Ctrl/Cmnd-P) which lets you set the destination of the current output. Reports are sent to the Screen by default, but you can choose another destination from this dialog. See Report Destination Dialog.

On macOS, the print **Setup** button is available allowing you to configure the Print Setup for the current report.

The **Print** option (Ctrl/Cmnd-P) prints the current class or report to current destination, if applicable. For example, when you are working in the method editor the option prints the currently selected method or set of class methods.

The **Help** option changes the mouse to a Help pointer allowing you to click on any part of Omnis to get Help; the option opens the Help window at the relevant topic.

View toolbar

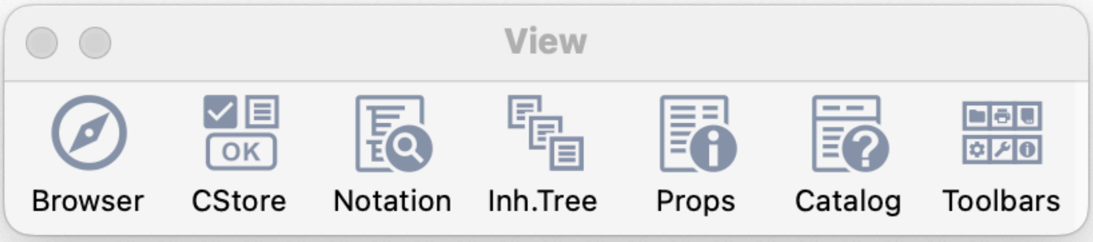


Figure 13:

The **View** toolbar opens the main Studio Browser and other tools available in the Studio IDE and has the same options as the **View** menu. Many of these tools are described in greater detail in this chapter. (On macOS, you can use Cmnd-Number or the equivalent Fn key to open any tool instead, e.g, to open the Studio Browser you can use Cmnd-2 or Fn+F2.)

The **Browser** option (F2/Cmnd-2) opens the Studio Browser which lets you create and examine libraries and classes. If the Studio Browser is already open and in Single Window Mode, this option will bring it to the top. If the Browser allows multiple copies of itself, this option opens the initial Browser displaying the libraries. See Studio Browser.

The **CStore** option (F3/Cmnd-3) opens the **Component Store,** to allow you to build web forms, windows, and reports. *This option will be disabled if there is no design window or report on top, or when a library is selected in Studio Browser.* See Component Store.

The **Notation** option (F4/Cmnd-4) opens the **Notation Inspector** which lets you view the complete Omnis notation tree. If the Notation Inspector is already open and in Single Window Mode, this option will bring it to the top. If the Notation Inspector allows multiple copies of itself, this option opens a new instance of the Notation Inspector. When you open the Notation Inspector the Property Manager will also open. See Notation Inspector.

The **Inh Tree** option (F5/Cmnd-5) opens the **Inheritance Tree** which lets you view the inheritance or superclass/subclass hierarchy in the current library. If you select a class in the Browser and open the Inheritance Tree it shows the inheritance for that class.

The **Props** option (F6/Cmnd-6) opens the **Property Manager** which lets you view or change the properties of an object: the properties displayed in the Property Manager will depend on the object or window currently selected or on top in the Omnis development environment (the Advanced option must be checked to see all the properties for the current object). If the Property Manager is already open, this option will bring it to the top.

The **Catalog** option (F9/Cmnd-9) opens the **Catalog** which lists all the field names and variables in your library, together with the functions, constants and event messages. If the Catalog is already open this option will bring it to the top.

The **View** menu has the **Toolbar Options…** and **Toolbars…** options to allow you to hide or show and configure the main toolbars in the Omnis IDE; this is useful on macOS since the toolbars are not shown by default.

The **JavaScript Theme** option (Ctrl/Cmnd-J) allows you to change the color theme used in all JavaScript remote forms (only available in the View menu).

Recent Classes

The **View** menu lists all the classes that have been opened recently, as does the Recent Classes option in the Studio Browser (at the bottom above the status bar). The maximum number of classes shown is limited to 9, but you can configure the number of classes shown by setting the “maxRecentClassEntries” in the “ide” section in config.json, which defaults to 9 (the value in earlier versions), but can be set to any value in the range 9 to 32 inclusive.

Note that this setting also affects the Recent Classes option in the Studio Browser, but since that only shows classes (or methods when the Shift key is pressed), there are typically fewer recent class items on the recent classes option than on the main View menu.

Tools toolbar

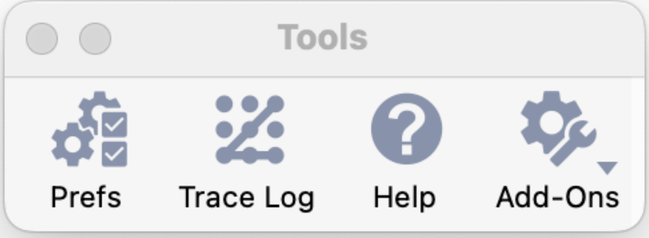


Figure 14:

The **Tools** toolbar contains the following options (some may be hidden in your operating system but you can show then using the **Toolbar Options…** and **Toolbars…** options in the View menu):

The **Prefs** option opens the Omnis Preferences in the Property Manager.

The **Trace Log** option opens the Trace log in a separate window (it is also displayed in the Studio Browser). The Trace Log provides a record of the operations and commands you have carried out. If there is an error in Omnis, such as when you start it up, you can look in the trace log for information about possible causes of the error. Omnis code execution and errors are also reported in the trace log: see Debugging Methods for further details.

The **Help** option opens the **Help Project Manager** to allow you to create your own built-in Help system, similar to the Omnis Help system (F1).

The **Add-Ons** button (or sub-menu option on the Tools menu) lets you open various additional tools, including:

Character Map Editor for mapping local and server character data; see Non-Unicode Compatibility
Compare Classes tool for comparing the methods in classes and libraries; see Converting Omnis Data Files
Convert Data File to RDBMS for converting an Omnis datafile to SQLite or PostgreSQL
Deployment Tool allows you to build an application package including your application file(s) for deploying to end users on desktop only (not web or mobile); see Deployment Tool
JS to Responsive migrates $screensize based remote forms to responsive layout; see Remote Form Migration
JSON Control Editor lets you create JSON defined JavaScript components; see JSON Control Editor
Method Checker for checking methods in classes and libraries; see Checking Methods
ODBC Admin tool for setting up ODBC access to Omnis datafiles
Port Profile editor for setting up your ports; see Port Profiles
Push Notifications lets you setup notifications for mobile apps; see Push Notifications
String Table Editor for providing multi-languages in your application; see Localization
SVG Themer allows you to convert SVG icons to Omnis themed icons; see Themed Icons
Sync Screens for synchronizing layouts in old fixed screensize remote forms (not used for responsive forms)
Synchronization Server opens the Admin tool for the Sync Server; see Serverless Client
Web Client Tools provides access to the JS Icon Export tool, the Icon Set Renaming Tool, and the JavaScript Theme Editor; see Theme Editor.

The following options on the Tools toolbar are hidden by default and are not required for creating new applications:

The **Icon Edit** option opens the Icon Editor to allow you to manage PNG-based icons in an Omnis icon datafile or #ICONS; note this is not required to create icons sets including SVG icons; these must be edited in a separate image editor.

The **Export** and **Import** options allow you to export or import data from an Omnis datafile which are used for legacy desktop apps only.

Desktop toolbar

The **Desktop** toolbar (hidden by default) is only relevant to running or testing desktop applications.* It lets you toggle between the **Design** environment and a simulated **Runtime** environment, so you can see your desktop application in user mode. You can also change the mode using the **Desktop** option in the **File** menu.

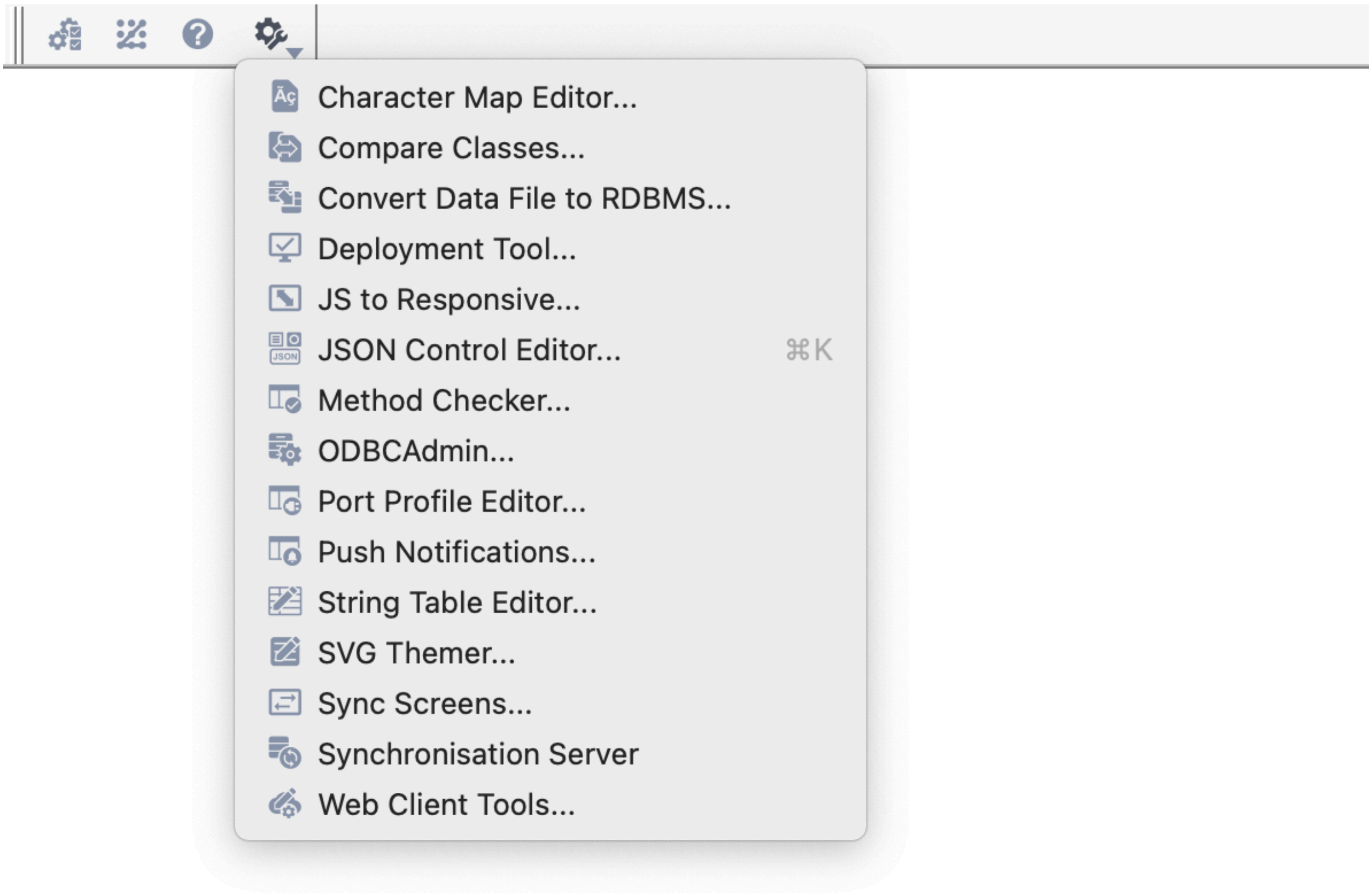


Figure 15:

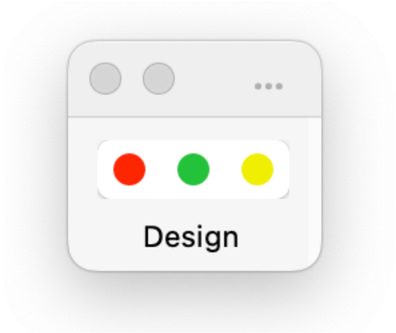


Figure 16:

In *Design mode* the standard menus, such as File, Edit, View, and Tools toolbars are visible. In *Runtime mode* all these are hidden and only user menus, data entry windows and toolbars defined in your library are visible. Also in *Runtime* mode, there is a cut-down version of the File and Edit menus on the main menu bar. Being able to switch to Runtime mode lets you see exactly how your application will look when the user runs it. In *Combined mode* (the default) all design and user menus, windows, dialogs, and toolbars are visible. When you select the Runtime option, the Desktop toolbar is installed so you can get back to the Combined mode.

Omnis Menu Bar

The main Omnis Menu Bar gives you access to various **File** operations (Save etc), the standard **Edit** menu functions (Undo/Redo, Copy, Paste, etc), as well as the **View, Tools, Window,** and **Help** menus. Many of the options in these menus appear on the main toolbars and are described in the previous sections: the other options are reasonably self-explanatory.

On **macOS,** the main Omnis menubar is always visible at the top of the screen when you start Omnis (see below). For macOS only, the **Omnis** menu contains additional options including the **About Omnis** option, **Preferences** (to open the Omnis preferences in the Property Manager), and the **Quit Omnis** option (Cmnd-Q).

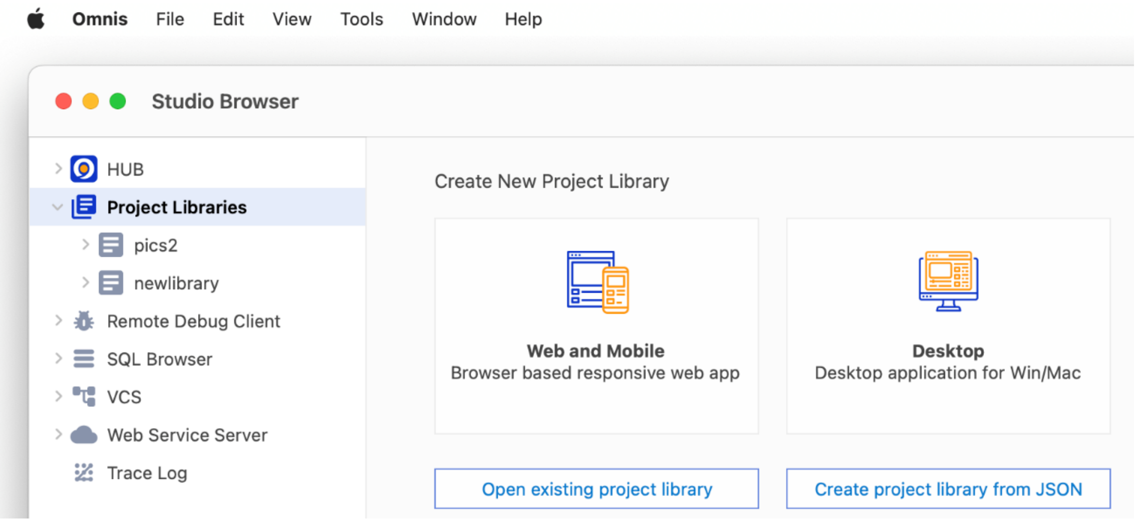


Figure 17:

On **Windows,** the main Omnis menu bar may not be visible, but you can press the **Alt** key to display it temporarily; it is located within the Omnis application window. The main Omnis toolbar is shown by default on Windows.

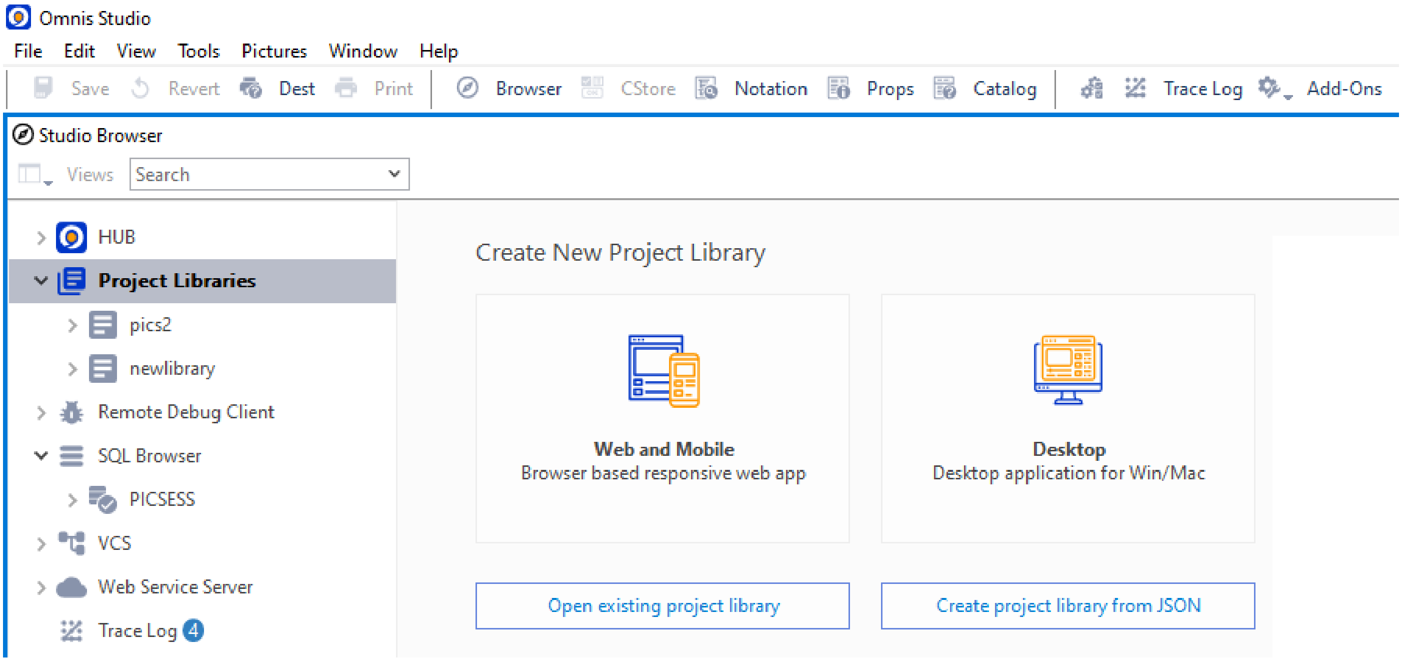


Figure 18:

On Windows, to display the main Omnis menu bar permanently, Right-click on the main Omnis toolbar and select the **Menu Bar** option.

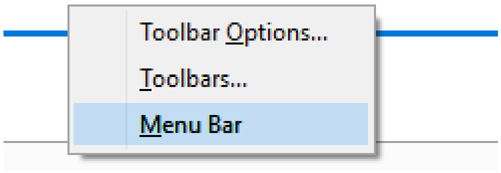


Figure 19:

The other options on the Toolbar context menu let you configure the main toolbars in Omnis Studio. The **Toolbar Options...** option lets you install or remove the Standard, View, Tools and Desktop toolbars.

Multi- Undo and Redo

The **Edit** menu in Omnis supports the standard edit options including **Undo, Redo, Cut, Copy, Paste, Clear, Select All** and **Past from File.** The Edit menu also lets you open the Find and Replace tool.

Omnis supports multiple **Undo** and **Redo** operations in the class design editors and the Method Editor. Omnis stores most operations on an *Undo* and *Redo Stack* which can be called using the **Undo** or **Redo** commands in the **Edit** menu, or using **Ctrl-Z** or **Ctrl-Y** key strokes on Windows, or **Cmnd-Z** or **Shift-Cmnd-Z** on macOS.

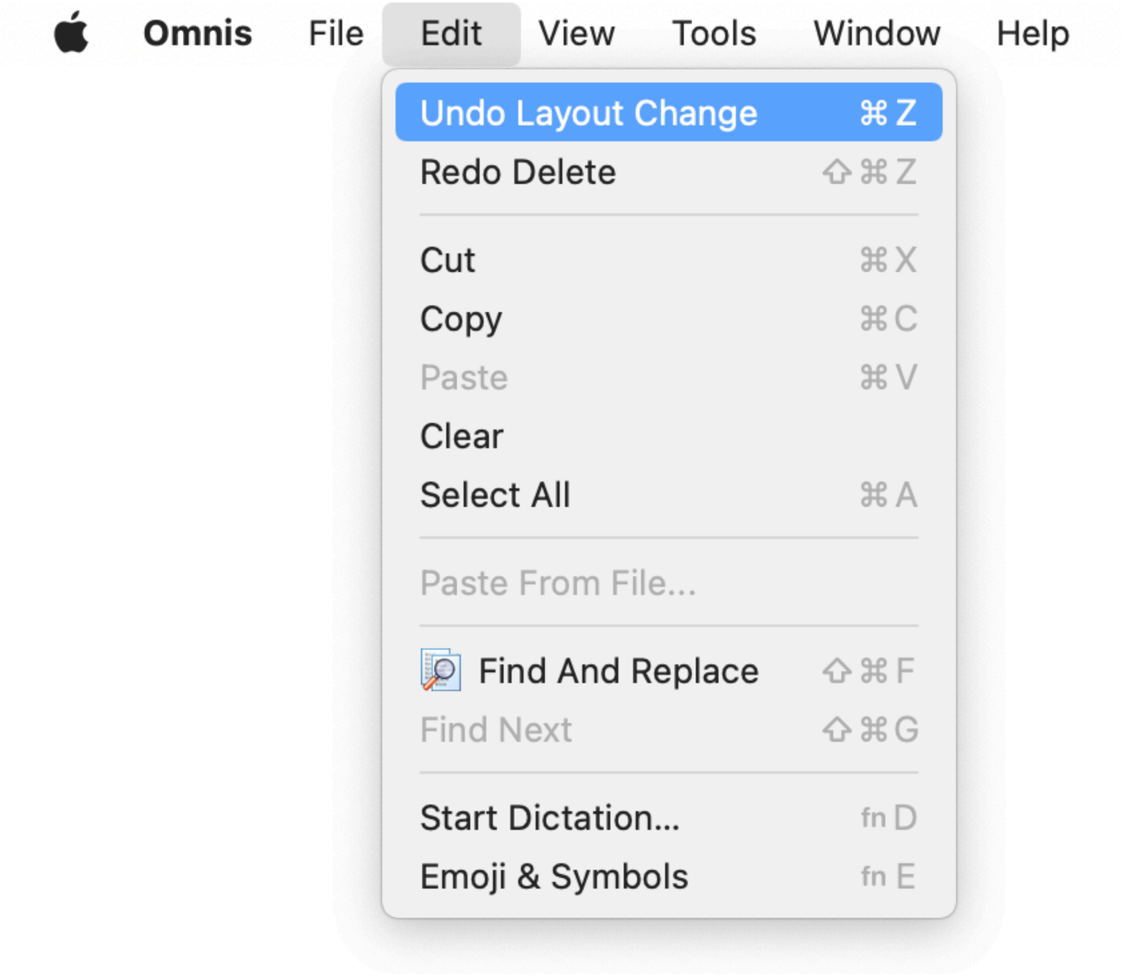


Figure 20:

As you undo and redo operations in a class editor, or the method editor, the **Undo** and **Redo** commands will update in the Edit menu to reflect the next operation that can be undone or redone (see below). When there are no operations that can be undone or redone the corresponding option in the Edit menu will be grayed out.

In general, most operations that support (single) Undo support multiple Undo and Redo, including moving and resizing objects, adding and deleting controls (including Cut and Paste), object property changes (in the Property Manager), align menu operations, and changing or deleting layout breakpoints in remote forms.

In effect, a separate Undo stack is kept for each editor, so as you switch from one editor to another, e.g. between two remote forms, the Undo or Redo commands will apply to the stack for that class editor (this does not apply when opening the Method Editor, see below). There is currently no limit on the number of operations that can be stored on the Undo stack.

To enable multiple Undo and Redo, Omnis saves a copy of the class data before and after an operation. To support this, there is a new temporary folder named ‘undotemp’ created automatically in the ‘studio’ folder at startup, which contains temporary copies of class data associated with undo stack entries; these files are deleted automatically, but in case they are not, any stray files are deleted when Omnis starts up.

Property Manager

You can Undo a property change when the Property Manager has the focus, provided that the current line in the Property Manager does not itself have an Undo stack (this can apply when the edit field has the focus). When you undo a property change, Omnis tries to select the affected property in the Property Manager. Undo works for inheriting and overloading a property.

Method Editor

If you open the method editor for a class, while the design editor for the class is open, Omnis clears the undo stack of the class design editor (but only if something is changed in the method editor). This prevents Undo or Redo in the class editor overwriting the class and losing any method changes.

Report Editor

Undo works in the report editor for the following operations: moving a report section, inserting or deleting a report line, and editing the page setup. Note that the report editor does not support Undo or Redo for the sort fields dialog. When you open this dialog, Omnis clears the report editor undo and redo stacks.

Form or Window Editor

Most operations within complex objects, such as a Complex grid or a Tab strip, support multi- Undo and Redo, such as, setting column widths in a Complex grid using the mouse or changing a grid line property.

Tools menu

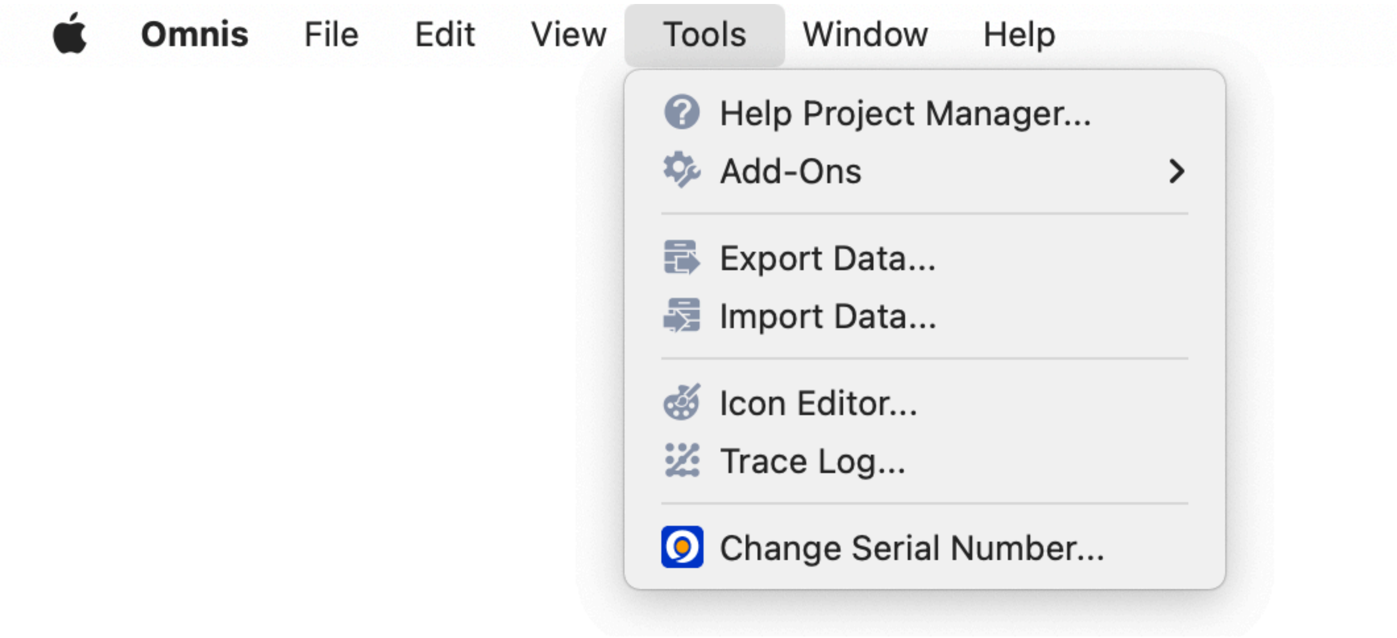


Figure 21:

The **Help Project Manager** lets you build help into your own libraries for the benefit of your own users.

The **Add-Ons** option lets you open various additional tools, described in the Tools toolbar section above.

The Export and Import options are only available for legacy apps using Omnis datafiles and should not be used for new apps. The **Export Data** option lets you export data from an *Omnis data file* (not a SQL database) using a number of different data formats. The **Import Data** option lets you import data into a data file from an existing export file or text file from another application.

The **Icon Edit** option opens the Icon Editor to allow you to manage PNG-based icons in an Omnis icon datafile or #ICONS; note this is not required to create icons sets including SVG icons; these must be edited in a separate image editor.

The **Trace Log** option displays the trace log which is a record of the operations and commands you have carried out. See Debugging Methods.

The **Options** option (available on Windows only) opens the Property Manager displaying the main **Omnis Preferences,** including settings for the Appearance of the Omnis environment, the Print devices for Omnis reports, and the main Page Setup. On macOS, the **Preferences** option in the **Omnis** menu (or **Prefs** on the Tools toolbar) opens the Omnis Preferences. See Omnis Preferences.

The **Change Serial Number** option which lets you re-serialize your copy of Omnis.

Context Menus

A *context menu* is a useful shortcut when you want to modify an object, or change the Omnis development environment. You can open a context menu from almost anywhere in Omnis by **Right-clicking** on an object or window background. On macOS, you can open a context menu by holding down the **Ctrl key** and clicking your pointer on the object; or you can use Right-click on your trackpad or mouse. The options in a context menu are context-sensitive and will depend on the object you right-clicked on. For example, if you Right-click on the Studio Browser its View menu will be opened containing options for changing its view and creating new classes.

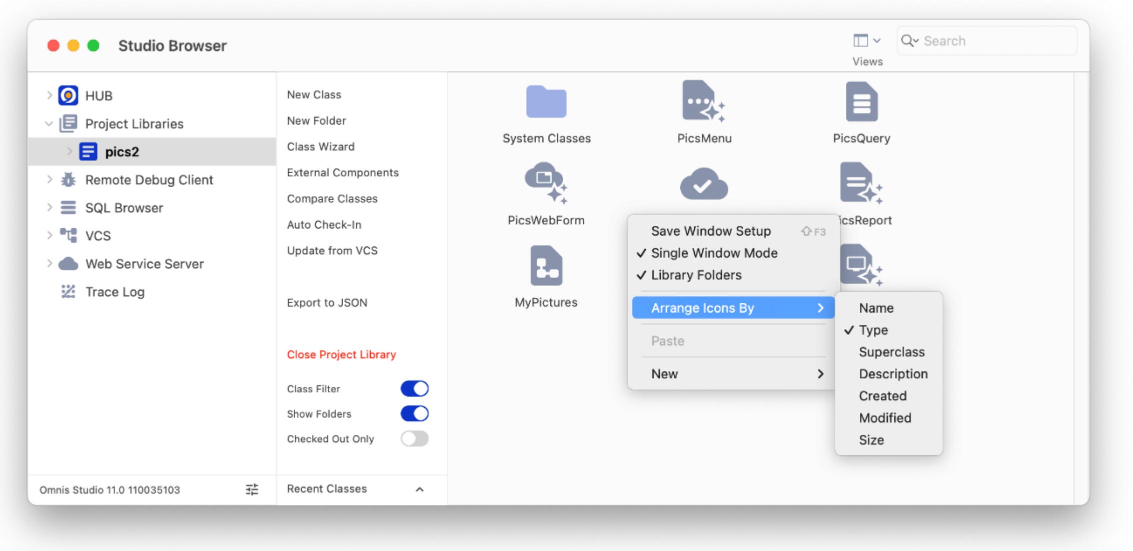


Figure 22:

The context menu on the Studio Browser lets you change the current view, or create a new class or folder when a library is visible in the Browser.

Save Window Setup

The context menu for most of the design tools in Omnis will have the **Save Window Setup** option which will save the settings or view for the current window: for example, the option will store the current setting for the Icons or Details (list) view in the Studio Browser.

The keyboard shortcut **Shift+F3** executes the **Save Window Setup** command for the current design window; the shortcut applies to all built-in dialogs and design windows. The 'saveWindowSetup' option in the IDE section of the keys.json file stores the shortcut. Function key shortcuts in macOS menus are shown as Fn rather than Cmnd+<n>.

Class Context Menu

If you right-click or Ctrl-click on a class displayed in the Studio Browser the class context menu will open: the contents of the menu will depend on the type of class, but some options are available for all classes, e.g. options for the VCS. Classes that can contain methods will show the **Methods** option (F8) which lets you open the Method Editor for the class. UI classes like remote forms will have an option to **Test Form** (or Open for window classes) to open a class instance, or report classes have the **Print Report** option to print the report (to the Preview window by default).

Variable Context menus

In the Omnis Method Editor or Catalog, you can right-click on a variable name and get its current value. The Variable context menu shows the variable name, its value and its type and allows you to open the variable window showing its current value.

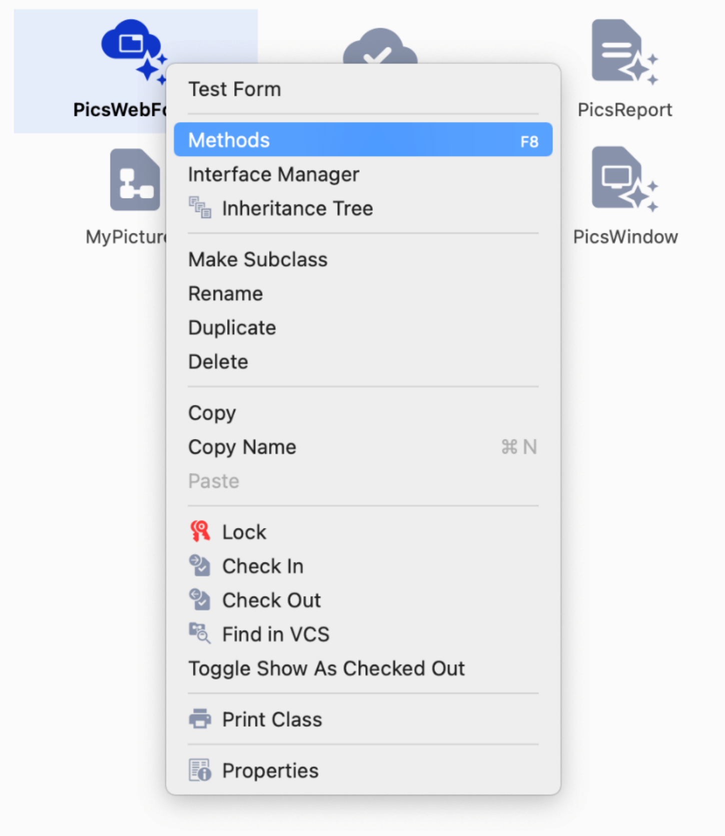
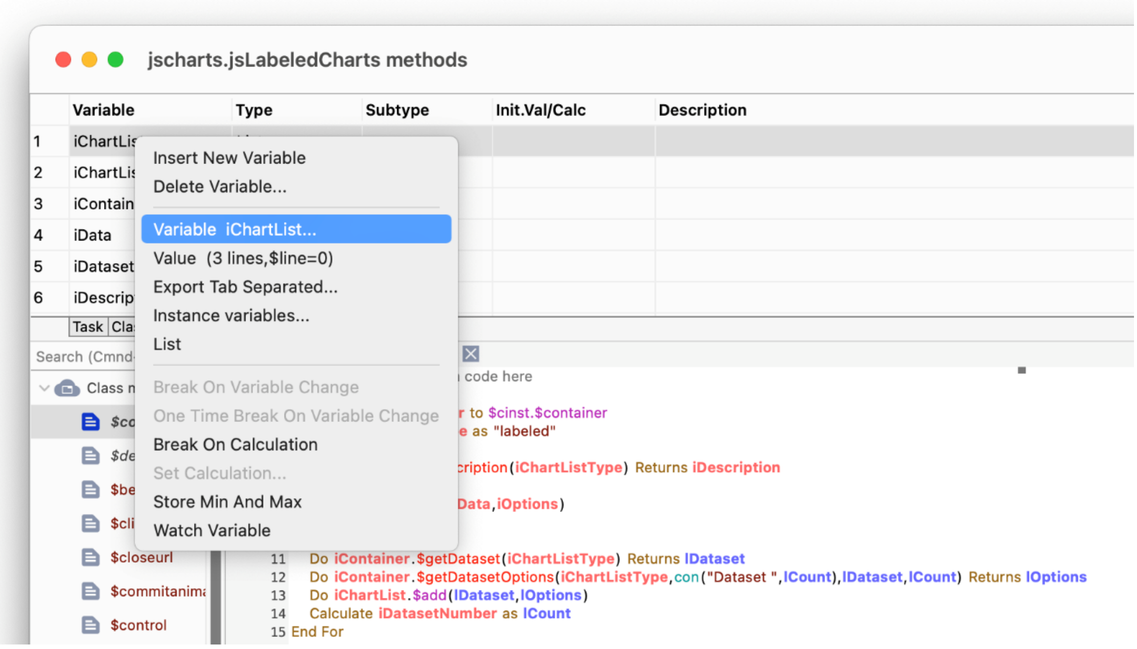


Figure 23:



```
code here

r to $cinst.$container
e as "labeled"

cription(iChartListType) Returns iDescription

Data,iOptions)

  Do iContainer.$getDataset(iChartListType) Returns lDataset
  Do iContainer.$getDatasetOptions(iChartListType,con("Dataset ",lCount),lDataset,lCount) Returns lOptions
  Do iChartList.$add(lDataset,lOptions)
  Calculate iDatasetNumber as lCount
End For
```

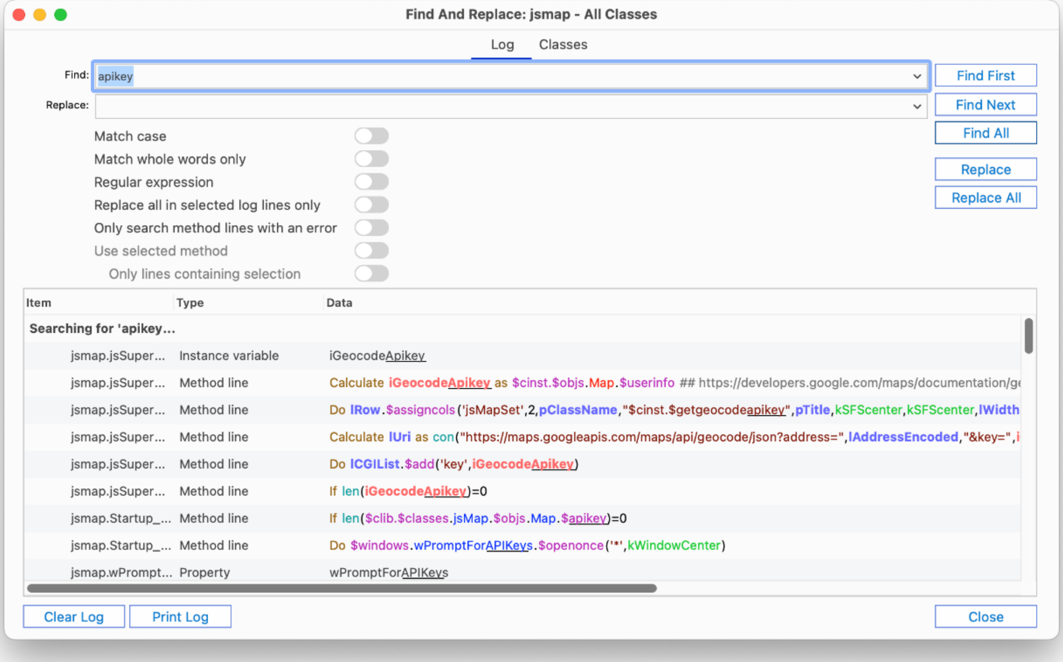
Figure 24:

You can right-click in many parts of Omnis and open up a menu appropriate to the object or item under the mouse, like a variable displayed in the method editor, as above.

Find and Replace

The Find and Replace tool lets you search through a class or library, or a number of classes or libraries, to find a particular text string. You can selectively replace occurrences of an item or replace all items.

The **Find And Replace** option under the Edit menu (or Ctrl/Cmnd-F) opens the Find and Replace dialog. The **Find Next** option (Ctrl/Cmnd-G) lets you find the next occurrence of the current find string.



Find: apikey

Replace:

Match case ○

Match whole words only ○

Regular expression ○

Replace all in selected log lines only ○

Only search method lines with an error ○

Use selected method ○

Only lines containing selection ○

Item	Type	Data
Searching for 'apikey...		
jsmap.jsSuper...	Instance variable	iGeocodeApikey
jsmap.jsSuper...	Method line	Calculate iGeocodeApikey as $cinst.$objs.Map.$userinfo ## https://developers.google.com/maps/documentation/ge
jsmap.jsSuper...	Method line	Do lRow.$assigncols('jsMapSet',2,pClassName,"$cinst.$getgeocodeapikey",pTitle,kSFScenter,kSFScenter,lWidth
jsmap.jsSuper...	Method line	Calculate lUri as con("https://maps.googleapis.com/maps/api/geocode/json?address=",lAddressEncoded,"&key=",i
jsmap.jsSuper...	Method line	Do lCGIList.$add('key',iGeocodeApikey)
jsmap.jsSuper...	Method line	If len(iGeocodeApikey)=0
jsmap.Startup_...	Method line	If len($clib.$classes.jsMap.$objs.Map.$apikey)=0
jsmap.Startup_...	Method line	Do $windows.wPromptForAPIKeys.$openonce('*',kWindowCenter)
jsmap.wPrompt...	Property	wPromptForAPIKeys

Figure 25:

If the Find and Replace dialog is already open and you bring it to the top, it selects the top-most open class ready to be searched (controlled by the **findAndReplaceSelectsTopClass** item in the 'ide' section of config.json).

The **Match case** and **Match whole words only** options can be used to find only those items that match the case of the search string or whole words only.

If you click on the **Find First** button Omnis will jump to the first occurrence of the text string in your selected classes or libraries. For example, if the specified item is found in a method, Omnis will open the class containing the method with the found item highlighted. **Find Next** (Ctrl/Cmnd-G) will jump to the next occurrence of the text string, and so on.

The **Find All** button finds all occurrences of the specified item in all the classes or libraries you selected and lists them in the Find log, and the *found* or *replacement* text is highlighted. The matched text is underlined if the **Highlight Matches** option in the context menu for the log is enabled (the default is on). If the text occurs more than once, up to the first 16 occurrences in the log are highlighted.

You can interrupt a find and replace operation at any time by pressing Ctrl-Break under Windows, Cmnd-period under macOS (or Ctrl-C under Linux).

The 30 most recent searches entered into the **Find:** box are saved for re-use, which you can view by clicking on the drop arrow in the search box. Note that all droplists and combo boxes in the IDE, including the Find and Replace dialog, use the **maxDisplayedDropListLines** configuration item in the 'ide' section of config.json to specify their maximum number of displayed lines. This defaults to 30, and can be 5-50 inclusive.

The **Show Checked Out Classes In Log** option in the Find and Replace log context menu (right-click on the results list) allows you to show which classes in the Find and Replace log are checked out of the VCS; the option is enabled by default and is saved in Window Setup. Changing the option via the context menu does not cause lines already in the log to be updated.

Classes tab

On the **Classes** tab you can select the libraries and classes in which you want to perform the find and replace. For a single library you can select some or all of the classes in the library. If you select more than one library under the Classes tab, all classes in all selected libraries are searched.

There is a button in the title of the class list (a folder icon, on the right) that allows you to search for parent folder(s) using a regular expression, and then select the classes contained in those folders in the class list.

Regular Expressions

When the Regular expression check box is enabled, Find & Replace supports PCRE2 compatible regular expressions which are sequences of literal characters and metacharacters that let you perform complex text search and modification. PCRE2 is an open source library of functions that provides syntax and semantics like Perl 5 for defining a search. See www.pcre.org for more information and full documentation about what metacharacters you can use.

PCRE2 provides improved error message reporting when there are problems with regular expression syntax, and these are reported where applicable. An error with the regular expression passed to the rxpos() function generates a debugger error with the specific error text rather than a generic invalid regular expression error.

However, when using the find field in the Code Editor, note that errors are not reported because the editor attempts to compile and use the regular expression on every keystroke.

Note for existing users: If you want to use the old regex syntax, you can set the useOldRegularExpressionSyntax configuration option to true (false is the default, so PCRE2 is used by default); this is in the ‘defaults’ section of the config.json file. When this is set to true, it only affects the rxpos() function.

Selected Log lines and Errors

The **Replace all in selected log lines only** option allows you to replace all occurrences of the search string in selected log lines only. The **Only search method lines containing an error** option restricts the search to only those method lines that contain an error.

Searching Selected Methods

The “Use selected method” and “Only lines containing selection” options help you find items while working in the Code Editor (the options are grayed if Find is opened from elsewhere). To use these options you need to *select at least one character* in a method in the Code Editor: Find and Replace searches all lines containing a part of the selection, and when it completes, it selects the text for the searched (and possibly replaced) lines.

Code Syntax Colors in Find Log

The code syntax colors used in the Code Editor are used to display method lines in the Find and Replace log (and the Trace log). You can set this using two entries in the ide section of config.json: findAndReplaceLogUsesSyntaxColors and traceLogUsesSyntaxColors, which are both enabled by default.

Find Log sorting & searching

You can sort the Find log list by clicking on the header buttons, plus you can sort the list by either of the first two columns by using the context menu. The context menu also has an item to sort the last column. Keyboard searching of the Find log list searches column one and two, which means you can locate entries of a particular type more easily in the Find log, by clicking on the Type header to sort the list, and then typing the type name to search the list.

$findandreplace method

The $findandreplace() method allows you to find and replace items within a class (it is a class method). The definition of the method is:

- **$findandreplace**(*cFind*, cRep [,*bIgnCase*=kTrue, *bWholeWord*=kFalse, *bRegExp*=kFalse, *bClearLog*=kFalse, *bReturnLog*=kFalse]) Returns *row*
  If cRep is #NULL, the method finds all instances of cFind; otherwise, the method replaces all instances of cFind with cRep. Returns the status in *row*.

The optional parameters bIgnCase, bWholeWord, bRegExp, and bClearLog replicate the options on the Find and Replace window. When bReturnLog is kTrue (the default is kFalse), the status row has an additional column named **Log** that contains the Find and Replace log; this has the same structure as the list returned by sys(241).

Renaming Objects

When you rename certain objects in your library, Omnis will replace all references to the object automatically. For example, if you rename a class variable in the method editor Omnis will replace all references to the variable within the class automatically. However, if you try to rename most other types of object, such as renaming a class in the Browser, Omnis will prompt you to find and replace references to the object. If you answer Yes to the prompt, Omnis will open the Find & Replace tool and the Find log which lets you monitor the find and replace or control whether or not certain references are replaced.

Spell Checking

Omnis checks spelling in text in end user apps, as it is entered in desktop forms (on the fat client), and in the Studio IDE during development; note this does not apply to JavaScript client remote forms. Spell checking allows words to be validated, based on the local language setting, and spelling suggestions are presented in the UI or used automatically, including the highlighting of misspelled words, and correcting misspelled words as they are entered.

Support for spell checking is provided by calling the *Spell Checker API* on the current operating system, including under Windows and macOS. Spell checking is enabled by default and will be used in the right context automatically, such as in Entry fields or in the Code editor, and there are various options or settings in the Studio IDE to manage spell checking.

Configuration

There are two options for how Omnis chooses a language to use with the system Spell Checker APIs. Which of these two applies depends on the entry **useSystemSettingsForSpelling** in the 'defaults' section of the Omnis Configuration file (config.json).

If useSystemSettingsForSpelling is true (the default), Entry fields use the system settings to identify the current language or languages. For macOS, this means the settings in the Keyboard, Text panel in System Preferences. For Windows, this means the System Locale.

If useSystemSettingsForSpelling is false, Entry fields use the National sort ordering locale for the current language in the Omnis localization data file.

If Omnis fails to initialize the system Spell Checker API to use the required language it reports this failure to the Trace log.

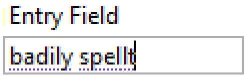
Window Class Controls

The following Window class (fat client) controls allow spell checking: Single Line Entry field, Multi Line Entry field, Combo box, String grid, and Data grid. These controls have the following properties to control spell checking:

Property	Description
$showspellingerrors	If true, the control underlines spelling errors using a dotted line
$autocorrectspelling	If true, and the user types a separator (e.g. space or comma) when no text is selected, the control replaces a misspelled word immediately before the selection with a correctly spelt word. Note that Undo allows you to revert to the originally entered text, and then continue typing without correcting it again

These properties are kFalse in pre-Studio 11 (converted) apps to maintain previous behavior.

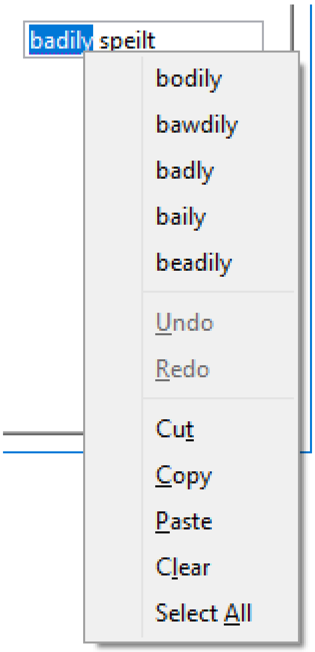
The dotted line used to underline spelling errors uses the color "colorspellingerror" in the system (and system.dark) section of appearance.json. The following screenshot shows an Entry field containing misspelled words:


Entry Field

badily spellt

Figure 26:

When $showspellingerrors is true, and the currently selected text in an Entry field is a misspelled word, the default context menu for the edit field includes up to 10 spelling suggestions, before the normal menu commands, such as Cut, Copy and Paste. Selecting one of these suggestions from the menu replaces the currently selected word.



badily speilt

Figure 27:

The context menu for Edit fields has the **Learn Spelling** menu item when the selected word is shown as a spelling error. If this menu item is selected the word will be added to the end user's custom dictionary and will no longer show as a spelling error. Conversely, if a word has been added to the custom dictionary, the context menu will show the **Unlearn Spelling** menu item, which when selected will remove the word from the custom dictionary and the word will be shown as a spelling error. These menu items are also available in the Code Editor context menu.

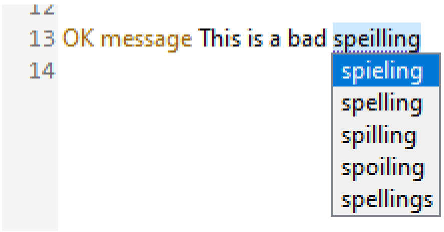
Code Editor

Spell checking is also enabled in the Code Editor (Method editor); there is a new **Show Spelling Errors** option in the View menu that is enabled by default.

Misspelled words in strings entered into code are underlined in the same way as edit fields underline spelling errors when $showspellingerrors is true. In addition, misspelled words outside *Square Brackets* are underlined for certain commands, including OK message, Yes/No message, No/Yes message, Prompt for input, Text:, Line:, and Send to trace log.

In addition, when **Show Spelling Errors** is enabled in the Code Editor, you can change the spelling of a *selected word* (which need not be misspelled) in either of two ways, described below (this applies to a string or outside square brackets in the commands as listed above).

You can select a word, and from the **Modify>>Selection** submenu you can select the **Change Spelling...** option: note that this command is only present if there are some possible suggestions for the selected word. You can also use the keyboard shortcut **Ctrl/Cmd+B** to change a word (specified in the **changeSpelling** key in the methodEditorAndRemoteDebugger section of keys.json). After selecting the command, a popup appears from which an alternative spelling can be selected to replace the word.



```
OK message This is a bad speilling
```

Figure 28:

Alternatively, you can select a word, Right-click on it, and the context menu contains a new **Change Spelling** hierarchical menu, with up to 10 suggestions, that can be used to replace the selected word.

Remote Debugger

The Remote Debugger also supports spell checking, enabled using the the **Show Spelling Errors** option. When this is true, for an edit session, the context menu for the editor includes up to 10 suggestions when the selected text is a misspelled word (that is, it

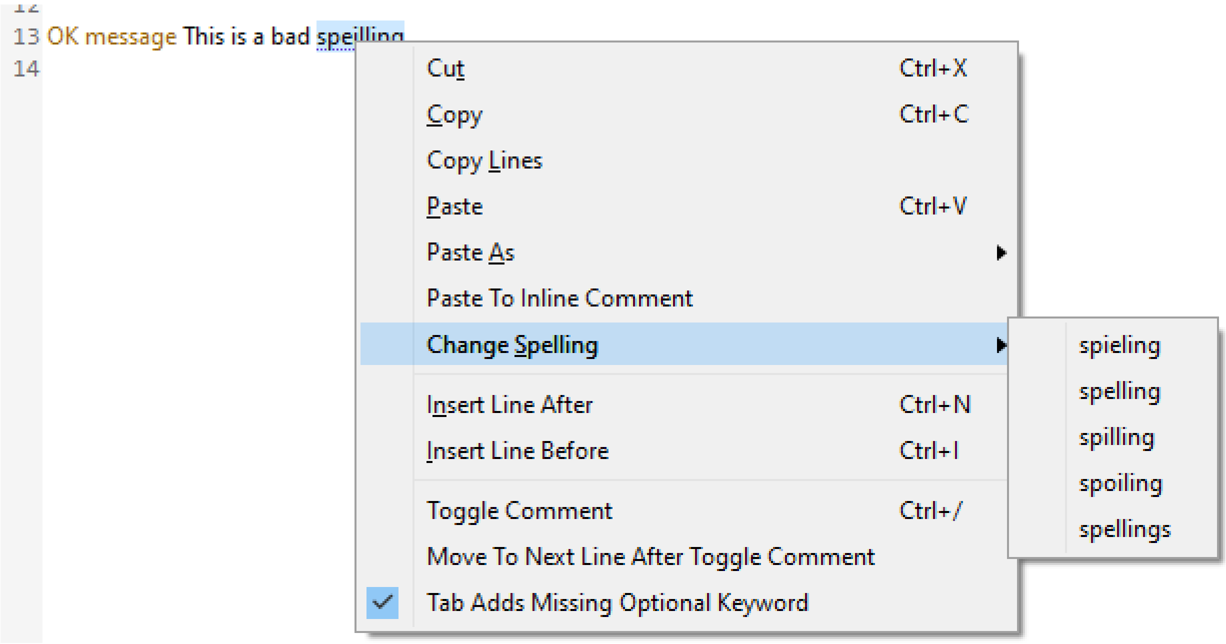


Figure 29:

behaves like a normal Entry field with $showspellingerrors set to true).

Component Store

The *Component Store* contains the objects and components you need to build the Remote forms and Reports in your web and mobile applications, plus Window and Toolbar components for desktop apps. When you create a new UI class or modify an existing one, the Component Store will open automatically. For example, when you create or modify a *Remote Form* in your application, the Component Store will display the JavaScript Components; the following screenshot shows a chart remote form (the JS Charts example app) with the Component Store docked to the left-hand side showing the **Buttons** group of components.

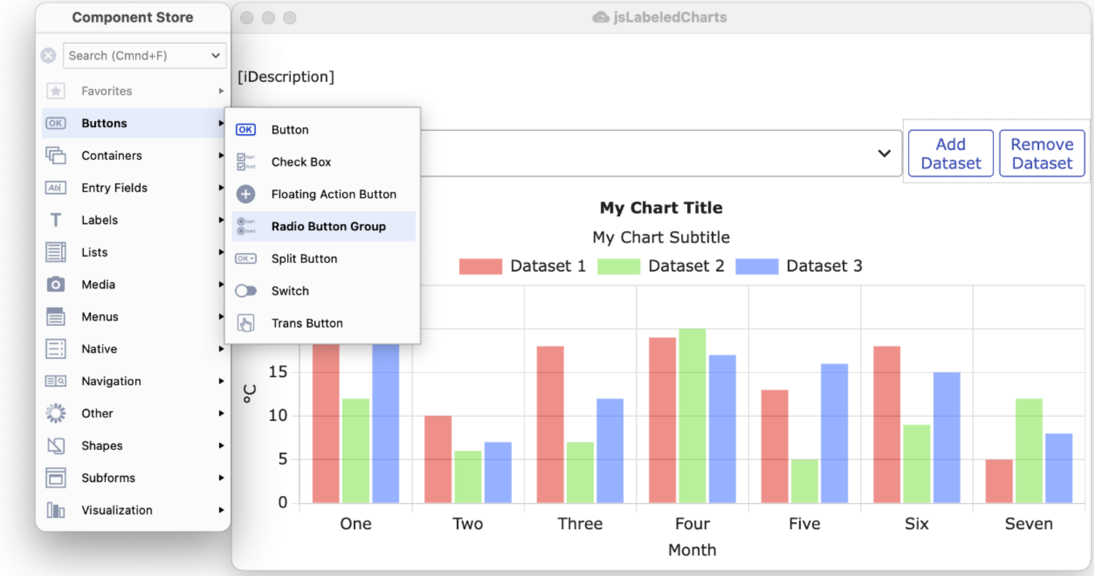


Figure 30:

There are a number of components in each group, shown in the sub-menu that pops out when you click on a group, such as the **Buttons** group, which contains the standard **Button, Check Box, Floating Action Button** (a new component), **Radio Button,** and other types of button components. The Component Store is displayed using the current IDE theme, including default (light) and dark themes.

On the Component Store context menu, the **Dock to Design Window** allows you to set where the Component Store is docked, either Auto, Left (the default), Right, and No (not docked, but floating). In Auto mode, the Component Store will dock to the left side of a design window, but if there is insufficient space on the left, the Component Store will dock on the right.

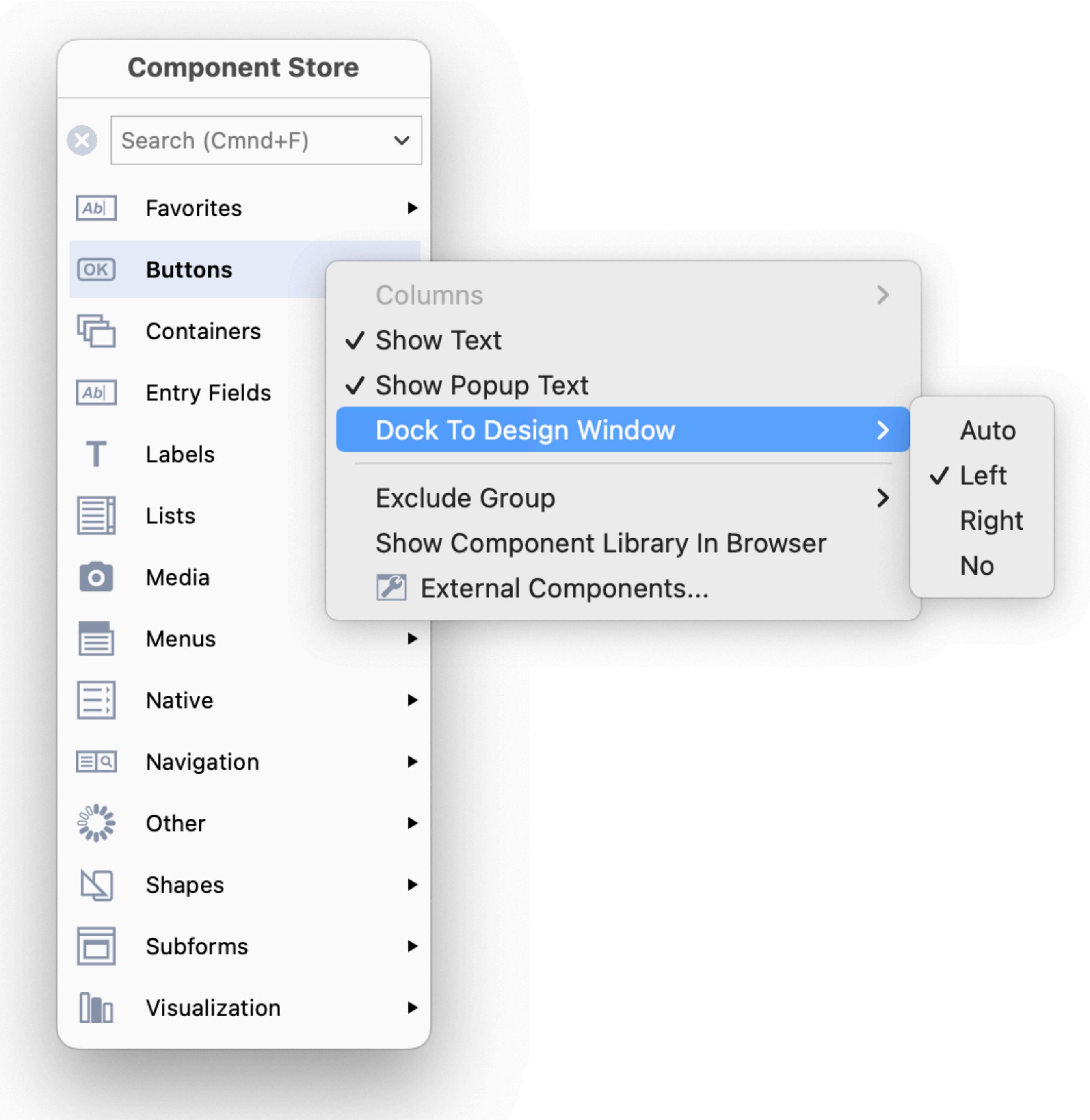

Component Store
Search (Cmnd+F)
Favorites
Buttons
Containers
Entry Fields
Labels
Lists
Media
Menus
Native
Navigation
Other
Shapes
Subforms
Visualization
Columns
✓ Show Text
✓ Show Popup Text
Dock To Design Window
Exclude Group
Show Component Library In Browser
External Components...
Auto
✓ Left
Right
No

Figure 31:

The initial view for the Component Store is to *Show Text labels* for the main groups and the components in the sub-menu popups, as shown above, but you can use its Context menu to change the appearance, e.g. hide the text or show it with 2 columns.

Searching for a component

You can use the **Search** box to locate a component or a group of similar components. As you type a search string, the contents of the Component Store is filtered, displaying only those components that *contain* the search string in their name, and the groups are hidden while the search is active. For example, you could enter "grid" to find all the grid components, as shown below. When the focus is on the Component Store, you can type **Ctrl/Cmnd-F** to put the focus into the Search box ready to type your search string.

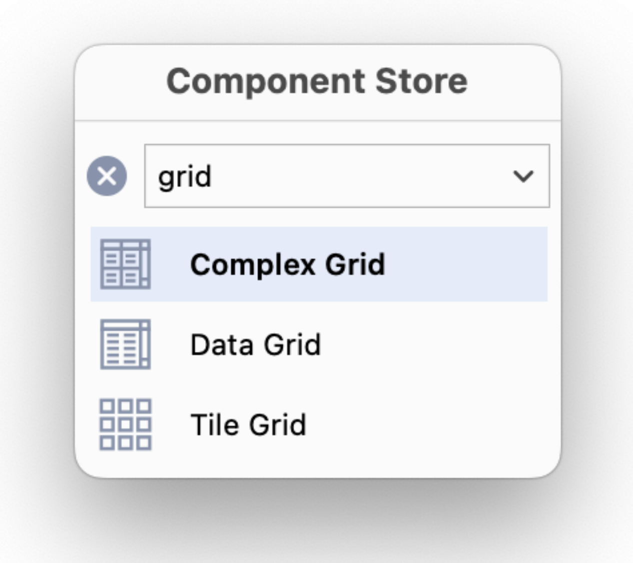


Figure 32:

Adding a Component to a form

To select a component, and add it to your Remote Form (or Report, Window, or Toolbar class), you can do one of the following:

- *Click on the main group icon* to open the sub-menu popup, then *click and drag a component icon* from the sub-menu, and drop it onto the form or window; as you drag the component out of the Component Store, the outline of the component is shown allowing you to place it precisely in the form or window.
- *Click and drag the icon shown in the main group* to create a component of that type; for example, you can drag the Button icon from the Buttons group to create a button, which is initially the default icon in that group (note the group icon/default component will change as you select different components).
- *Double-click an icon* in the main group or any sub-menu popup to add a component of that type; in this case, the component is *added to the center* of the form or window (double-clicking is not supported for report classes).
- *Press Return* to add the *currently selected component* to the design window (not supported for report classes).

Alternatively, you can use the keyboard to select a component:

- *To use the keyboard,* press **F3** to put the focus on the Component Store, use the **Up** or **Down** arrow keys to select a main group, press the ***Space*** key to open the sub-menu popup for the group, then *use the Arrow keys* to select a component, *and* press the ***Return*** key to add the component to the center of the form or window; you can use the ***Esc*** key to deselect/close a sub-menu popup.

The most recently selected group is highlighted in a color, while the icon for the most recently chosen component from any sub-menu popup is shown as the initial/default icon for the group; therefore, as you select different components from different groups, the initial or default icons will change. For example, if you previously chose a Combo box from the Lists group, the Combo box icon is shown in the main Lists group, and you can then drag or double-click the Combo box icon from the Lists group without opening the sub-menu to create a Combo box in your form.

If you create any Compound Objects for Remote forms they will appear in their own group in the Component Store: you can define compound objects by editing the Component Store library; see below. For example, Window classes have the Labeled Fields group containing the Labeled Entry Field and Labeled Masked Entry fields.

Changing the Appearance

You can change the appearance or layout of the Component Store using its Context menu. For example, you can Right-click/Ctrl-click anywhere on the Component Store and select or deselect the **Show Text** or **Show Popup Text** option to hide or show the text labels for the main groups or sub-menu popups, respectively.

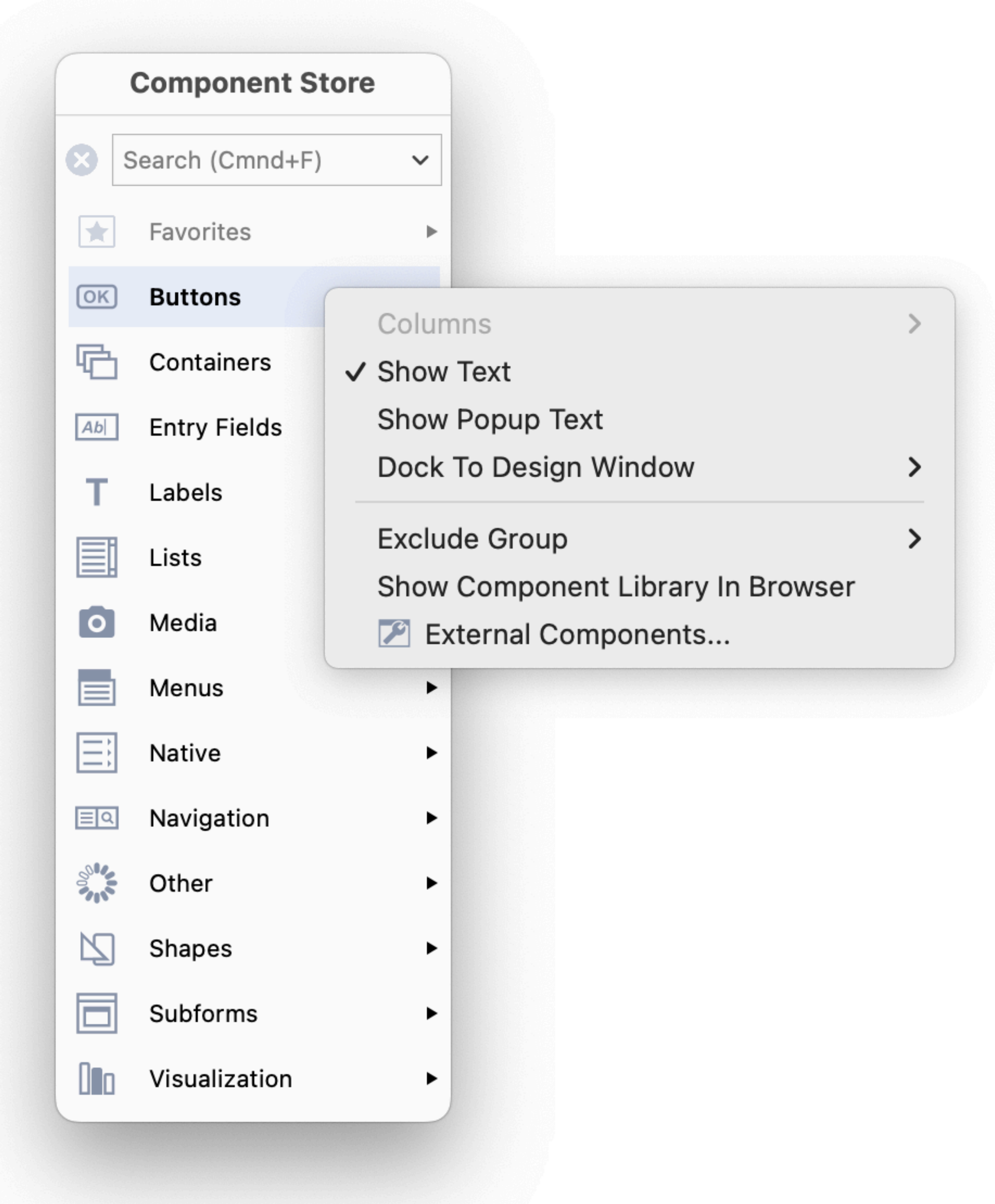
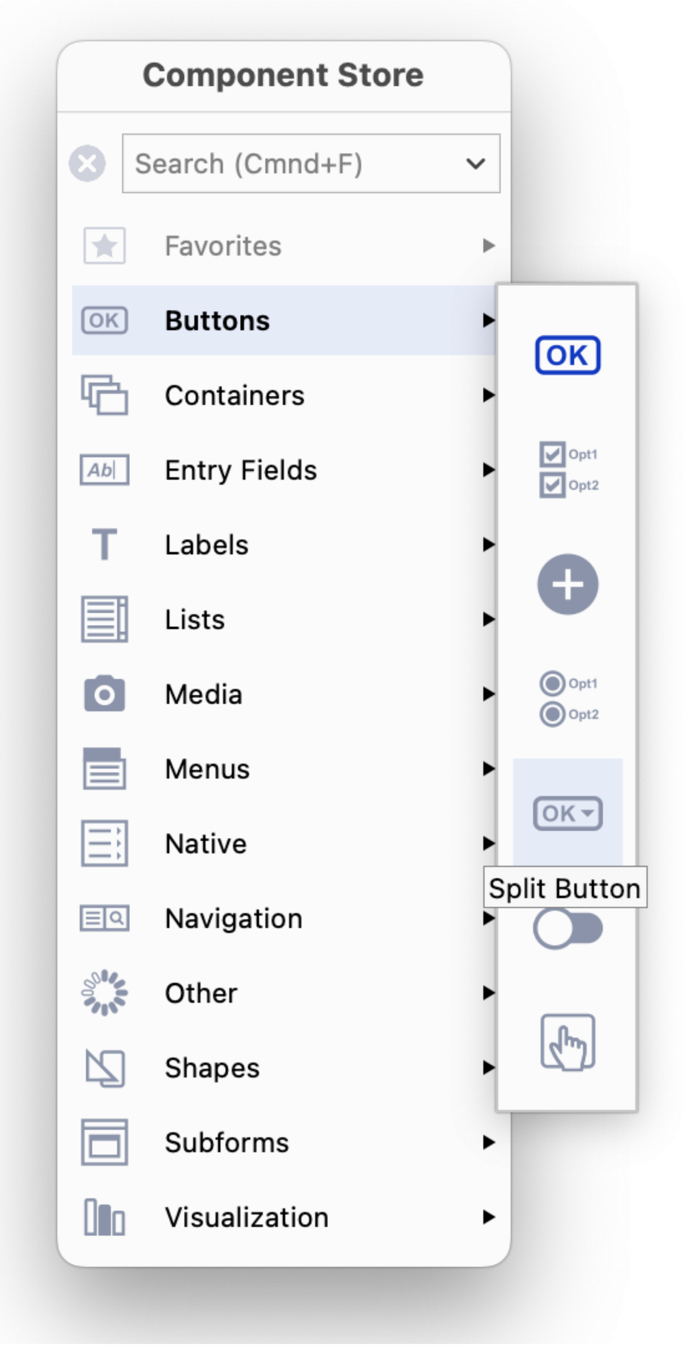


Figure 33:

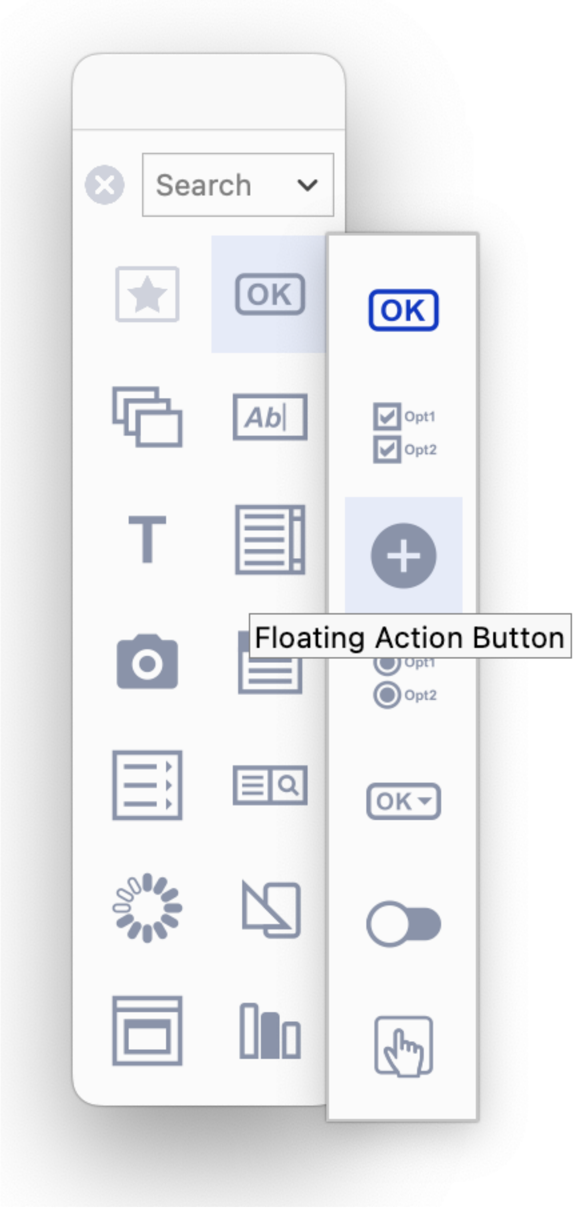
As you hide or show the Text labels, the icons will switch between *Large or Small* icons automatically (note the icons change automatically, so you cannot manually select large or small icons, as in previous versions). When the Text labels for the main groups or sub-groups are hidden, each icon has a tooltip that displays the component name or group name *as you hover over the icon*.

Note that there is no Save Window Setup command for the Component Store, since it saves its settings and current position automatically when it closes.

When the Show Popup Text optionis disabled, tooltips are displayedon the components in a sub-group



When both Show Text... options are disabled,large icons are shown and tooltips are displayedon the main groups and sub-groups



If the **Dock To Design Window** option is set to Auto, the Component Store is docked or “attached” to the left side of the current design window, or if there is not enough space to the left of the design window the Component Store is docked to the right side of the design window. If this option is set to No (not docked), you can drag or “tear” the Component Store from the design window and it will float within the Omnis application window, plus its last position is remembered automatically. The following image shows the Component Store floating next to a remote form:

When the **Dock To Design Window** option is to Auto, Left or Right, you can temporarily drag or “tear” the Component Store away from the design window by dragging its title bar, but it will *snap back and become docked again* when you move or reopen the design window, or when you change the docking options from the context menu.

Two Column mode

When the Text labels are hidden on the main groups (i.e. the **Show Text** option is unchecked), you can configure the main group icons in 1 or 2 columns using the **Columns** options on the context menu (the sub-menu text can be enabled or disabled in 2-column mode, as shown below). The Columns option is disabled (grayed) when the Text labels on the main component groups

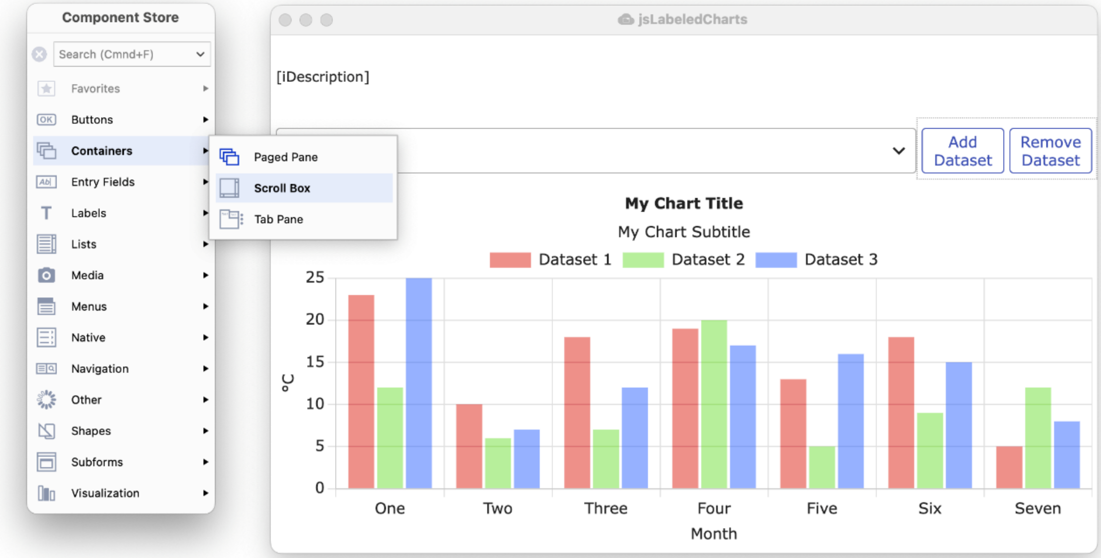
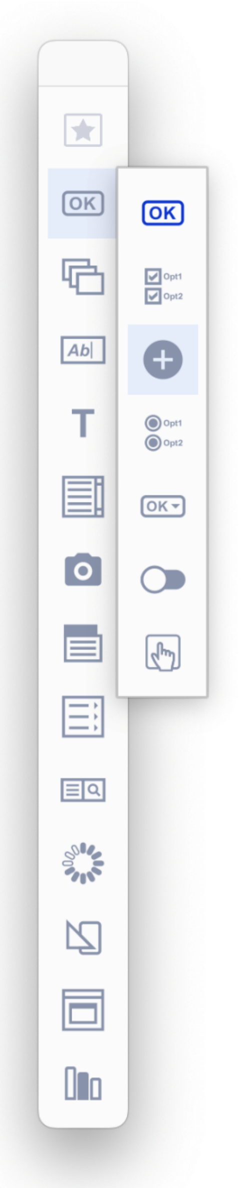
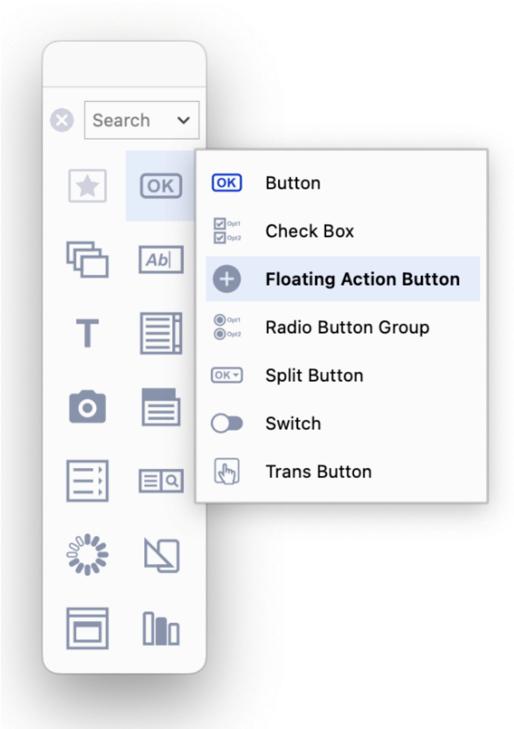


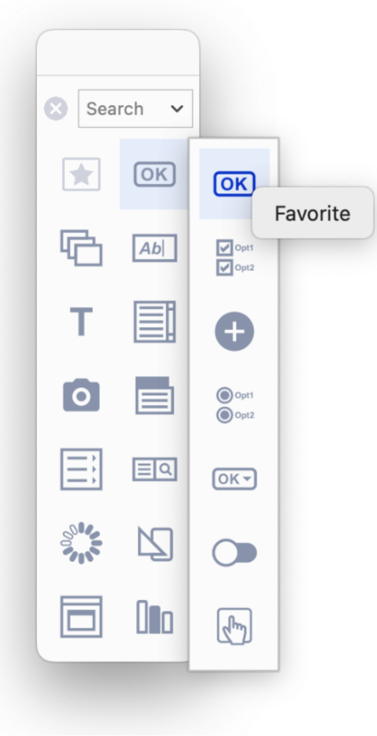
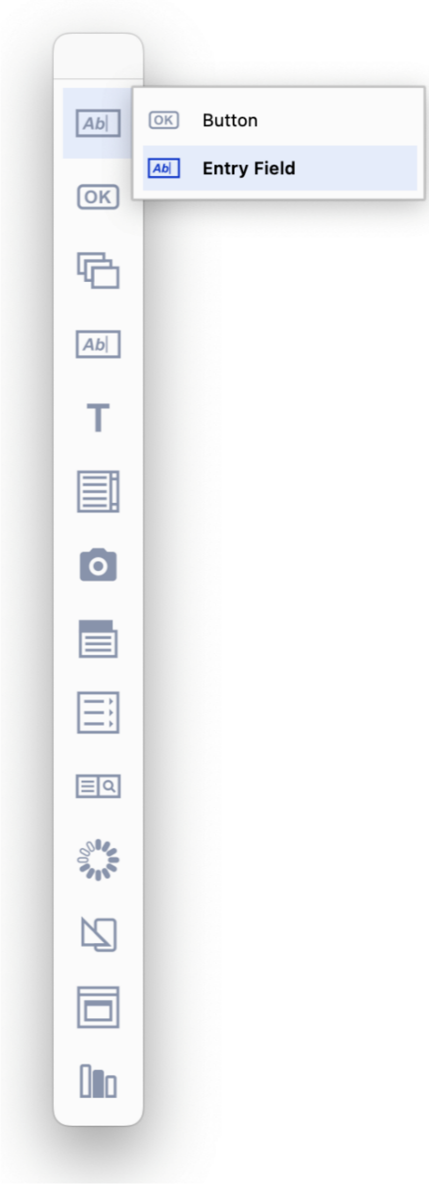
Figure 34:

are shown, and therefore you cannot enable 2-column mode in this case. Note the Search box is hidden when the Component Store is in single-column mode without text labels.

Single Column modePlus Show Text and Show PopupText are disabled	Two Column modeShow Text is disabled,Show popup text is enabled
	 Search OK Ab T Button Check Box Floating Action Button Radio Button Group Split Button Switch Trans Button

Favorites

You can add any single component to the **Favorites** group at the top of the Component Store window (shown initially with a Star icon). To add a favorite, Right-click on the icon for the component in a sub-menu and select the **Favorite** option. Adding components to the Favorites group makes it easier or quicker for you to select any controls that you use constantly. For example, in the following screenshot the Button and Entry fields have been selected as favorites and are now shown together in the Favorites group at the top of the Component Store.

Right-click a component,Select Favorite; in this case,Button is added to theFavorites group	You can add components fromdifferent groups to the Favorites group;in this case, the Button and Entry Field havebeen added to the Favorites group
	

To remove a component from the Favorites group, you need to right-click

Further Options

The options in the **Exclude Group** sub-menu in the Component Store context menu are checked by default, meaning that the **Deprecated** and **Internal** component groups are hidden or excluded by default; note that there are no Deprecated or Internal components for Remote forms, so these groups are only relevant for Window class controls at present. You are advised not to use the components in these groups, as they are included for backwards compatibility only, or for internal use, and should not be used for new applications.

The **Show Component Library In Browser** option allows you to change the contents of the Component Store and its groups; when selected, this option shows the Component Library (comps.lbs) in the Studio Browser, ready for you to edit it (as in previous versions). In general, you do not need to edit the Component Library, unless you want to add your own controls, compound objects, or class templates: see below.

The **External Components...** option opens the External Components dialog, allowing you to load external components (as in previous versions); this is only relevant for window and report classes, since all JavaScript components are loaded and displayed by default when designing remote forms. Note that all external components are shown in the new Component Store even if they have not been marked in the External Components dialog to be loaded.

Configuration

There are a number of options in the Omnis Configuration (config.json) and Appearance (appearance.json) files that control the behavior or appearance of the Component Store. The time taken for a group sub-menu to pop out can be set using the **componentStorePopupDelay** item in the 'ide' section of config.json, an integer specifying the popup delay in milliseconds. The default is -1 meaning that Omnis calculates the delay to be just longer than the double click time, which means you can double click on an entry to add the corresponding default component to the design window without the popup appearing briefly.

There is a new 'componentStore' group in appearance.json containing the item **colorgroupdefault** that allows you to set the icon color for the default component in a group in the Component Store.

Editing the Component Store Library

IMPORTANT: *You are advised not to change the properties of any of the existing components or class templates, but to duplicate an existing control and make any changes to the copy. In most cases, you do not need to edit the Component Store Library, except if you want to create your own class templates or compound objects.*

The content of the new Component Store window is driven by the classes in the **Component Store** library called 'comps.lbs' (as in previous versions). To open the component library, select the **Show Component Library In Browser** option from the Component Store context menu, or you can Right-click on the **Libraries** node in the Studio Browser and select the **Show Comps.lbs** option (the latter is useful if you do not have a library open). The $componenttype property for all classes and templates that appear in the Component Store is set to kCompStoreDesignObjects.

All controls in the Component Store library now have the property **$componentinfo,** which is a row of information that specifies which group the object appears under in the new Component Store window. The $componentinfo property is visible in the Property Manager when you are editing a component on a Remote Form, e.g. in the JSFormComponents remote form class (also for Window, Report, or Toolbar classes).

Click on the $componentinfo property in the Property Manager to edit it: it has three columns defining the group, icon, and default status for the object:

- **group**
  The name of the group to which the object belongs. Group names are case insensitive, for example: Lists, Buttons, Entry fields.
- **iconid**
  The icon used for the object in the Component Store, which should be an SVG image placed in the new icon set 'componenticons'. Icons are displayed at 20x20 or 28x28 and SVG images will scale to fit the current size.
- **default**
  A Boolean that indicates if the object is the initial default in its group (the default object will change once a different object is chosen). If default is set to true for more than one object in the same group, the initial default will be the first object according to the case-insensitive ascending sort order of objects within the group by their name.

Compound Objects

A Compound Object is comprised of two or more standard objects grouped together to make a single object that appears in the Component Store, such as the **Labeled Entry Field** available for Windows Classes. When you drag a compound object from the Component Store, all objects in the grouped object are created in the remote form (or window). The Tab Pane in the Containers group is a compound object, combining a Tab control and a Paged pane linked together.

You can create Compound Objects in the Component Store library, inside one of the Remote form or Window Component Store classes (or your own class but $componenttype must be set to kCompStoreDesignObjects). To create a Compound Object:

- Open the Component Store Library by right-clicking on the background of the Component Store and select the **Show Component Library In Browser** option; comps.lbs will be shown in the Studio Browser
- Open the Component Store class according to the type of Compound object (Remote form or Window class), then add the objects that will form the compound object, e.g. copy an Entry field and Label if you want to create a Labeled field; you are advised to create copies of the standard objects to form your compound objects. Note you cannot include line objects in a compound object
- Assign a name to the first object; this will be the name of the Compound object in the Component Store

For a Remote form, the first object is the object that occurs first in the field list window. For a Window class, the first object is either the first background object in the field list window, or if there are no background objects in the compound object, the first foreground object.

- Select all the objects that will form the Compound object, right click, and select **Group**
- Set the **$componentinfo** property of the group of objects, including the control name, group name and icon id (all members of the group should have the same value)
- Save the Component Store Library and close it

When the Component Store reloads in design mode, there will be a new Compound object with the specified name, group and icon id. The icon of a Compound object is shown in the Component Store with an additional … icon.

The dropped compound object has the same layout as its original objects, anchored at the top left of where you drop it.

You can use a responsive remote form to provide different layouts of the compound object for different breakpoints. Similarly, you can also set breakpoint-specific properties that will be set appropriately after dropping the compound object. Note that the Component Store Library may be using different breakpoints to your library, so the values used for each breakpoint in your library, after dropping a compound object, are the values for each nearest breakpoint, when comparing a Component Store breakpoint with a library breakpoint.

Container Compound Objects

The Component Store also allows you to create compound components using a Container field, such as a scrollbox or paged pane, with other objects inside the container. For example, you could create a compound object comprising a Scroll box (now available for remote forms) with a tab strip as its top toolbar component and a paged pane as its client component.

Class Templates

There are a number of Class Templates or Wizards that appear in the Studio Browser that are defined in the Component Store Library (such as Net Classes available in previous versions). You can create your own class templates, but the way you define these has changed (although the way you created class templates in previous versions is still supported for backwards compatibility).

Each class template in the Component Store Library has the new $componentinfo property, but for classes it has a single column named **group.** This allows a group to be specified for the class when it appears as a template or wizard in the Studio Browser. To use new $componentinfo property to define a class template:

- Select the class and set the group in $componentinfo for the class to the template name
- Set $componenticon to an Icon ID, preferably an SVG icon image
- Add a description for the class template in $desc

If you do not supply a group for a class template or wizard, it appears in a group named using its class type.

Property Manager

The **Property Manager** lets you display or change the **Properties** of the currently selected object in the Studio Browser or on a design window. This could be a library file or a Remote Form class selected in the Studio Browser, or a JavaScript component selected in a Remote Form. In addition, the Property Manager can be used to change the Omnis global preferences ($root.$prefs), and for specific classes or objects, the Property Manager will show the **Methods** for the currently selected object.

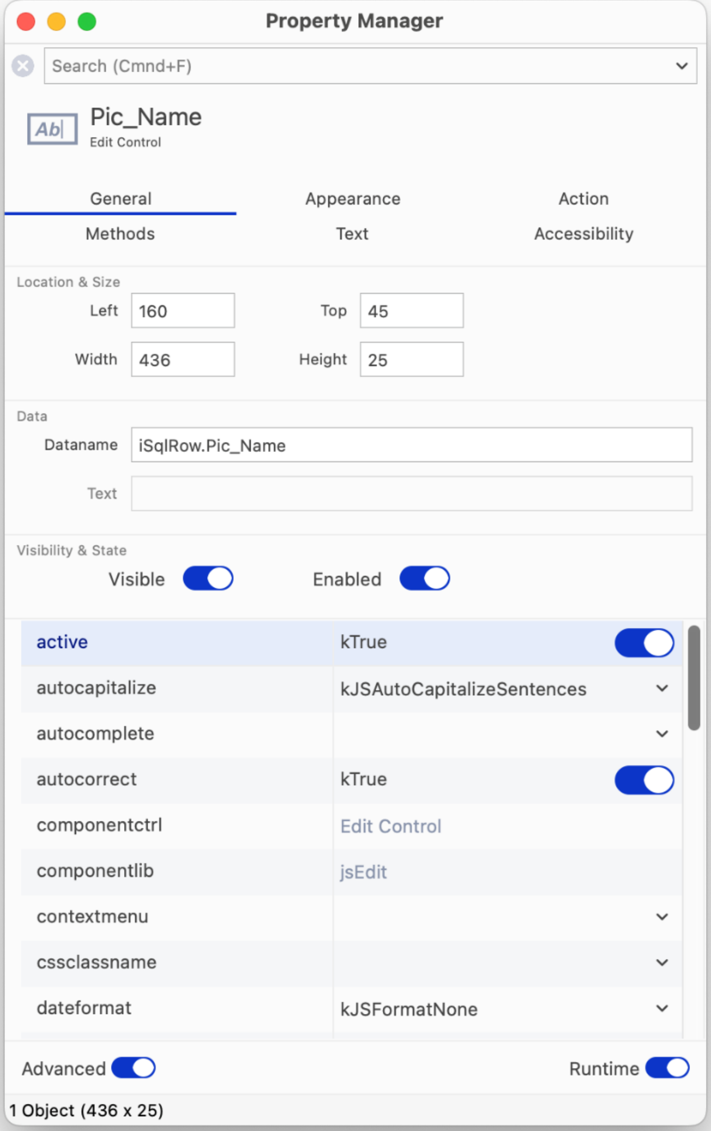
The Property Manager should appear automatically when needed, as soon as you click on a form or object that has properties. If it is not visible you can display it either by selecting the **View>>Property Manager** menu item from the main Omnis menu bar, or by pressing **F6/Cmnd-6,** in which case the properties for the class or currently selected object (library, class or object) will be shown in the Property Manager.

You can Right-click on a class or object, such as a remote form background or component, and select the **Properties** option from the context menu to display the properties for that class or currently selected object.

You can Right-click on a class or object, such as a remote form background or component, and select the **Class Properties** or **Field Properties** option from the context menu to display the properties for the current class or object under the mouse.

The properties for the current object are shown under a number of **tabs** (groups) such as *General, Appearance, Text,* and so on, unless you use the Search option (see below) or disable the **Advanced** filter, in which case the tabs are hidden. In addition, for some objects, the most common properties, such as $name and $dataname, are shown in a panel at the top. The following screenshot shows the Property Manager for a Remote form Entry field with the Advanced option enable (on).

The Property Manager also shows the size of the currently selected object (see above) as width x height coordinates in the status bar at the bottom of the window, plus the position of the pointer is shown as X-Y coordinates relative to the top-left corner of the design window. When a group of objects is selected, the width x height of the area occupied by the group of selected objects is shown.



Pic_Name
Edit Control

Location & Size

Left 160 Top 45

Width 436 Height 25

Data

Dataname iSqlRow.Pic_Name

Text

Visibility & State

Visible ● Enabled ●

active	kTrue	●
autocapitalize	kJSAutoCapitalizeSentences	⌄
autocomplete		⌄
autocorrect	kTrue	●
componentctrl	Edit Control	
componentlib	jsEdit	
contextmenu		⌄
cssclassname		⌄
dateformat	kJSFormatNone	⌄

Advanced ● Runtime ●

1 Object (436 x 25)

Figure 35:

Advanced Property Option

The **Property Manager** displays all the properties (and methods) for an object, or it can show a subset of properties. There is a switch at the bottom-left of the Property Manager window labelled **Advanced,** which either shows *all properties* for the current/selected object listed under separate tabs, or a single filtered list of "basic" properties (with no tabs) when the Advanced option is *disabled* (off): the latter is the default view for new installations of Omnis Studio and is intended for new users who may only need to access the basic properties.

When the Advanced option is *disabled* (off) the property list shows a basic subset of properties for the current object (selected library, form or component), or for the current context in the IDE, such as the Omnis preferences. For example, the following image shows the properties for a Remote Form when Advanced is disabled (off):

Note that if you use Find & Replace (on the Edit menu) and double-click on the find and replace log to select a property in the Property Manager, the Property Manager automatically switches to Advanced mode if the property is not part of the basic set.

Modifying the basic set of properties

The basic set of properties is defined in a file called basicproperties.json and stored in the Studio folder under the main Omnis folder. You can modify this file if you want to change the properties shown in the filtered state of the Property Manager. The file is in JSON format, and contains an array of property names which must be lower case, and include the :: prefix if the property name requires one (e.g. for some external component properties).

Omnis re-reads this file if it has changed when you uncheck the Advanced option: so checking and unchecking this box forces a re-read. If the file has invalid syntax and cannot be parsed, Omnis writes an error to the trace log, and no basic properties will be displayed.

Searching the Property Manager

There is a Search box at the top of the **Property Manager** window which allows you to search for a property: *note the search box is only visible when the Advanced option is enabled.* You can type a word or part of a word into the search box and the property list will update itself as you type.

The search results are property names that *contain the string* you entered, and they are shown in a single tab named 'Search'. The search results are always sorted by property name, irrespective of the sort list option on the context menu. You can click on a property in the property list and update or set its value.

For example, entering 'show' into the property search for a remote form will provide a subset of properties containing the word 'show'.

You can use the Backspace to clear a search string character by character, or you can click on the X icon to clear the whole string. The shortcut Ctrl/Cmnd+Shift+D moves the focus to the search box: you can press tab to return the focus to the property list. The 30 most recent searches are saved for re-use, which you can view by clicking on the drop arrow in the search box.

Each keystroke in the Search box performs a search, so there is a delay before a search is saved to the list: the delay defaults to 500ms, but you can change it in the config.json file in the "ide" group: "saveSearchDelay".

If you use Find & Replace (on the Edit menu) and double-click on the find and replace log to select a property in the Property Manager, the Property Manager clears the search before selecting the property. For both the Basic mode, and the Advanced mode when search results are being displayed, copy and paste properties are disabled on the context menu.

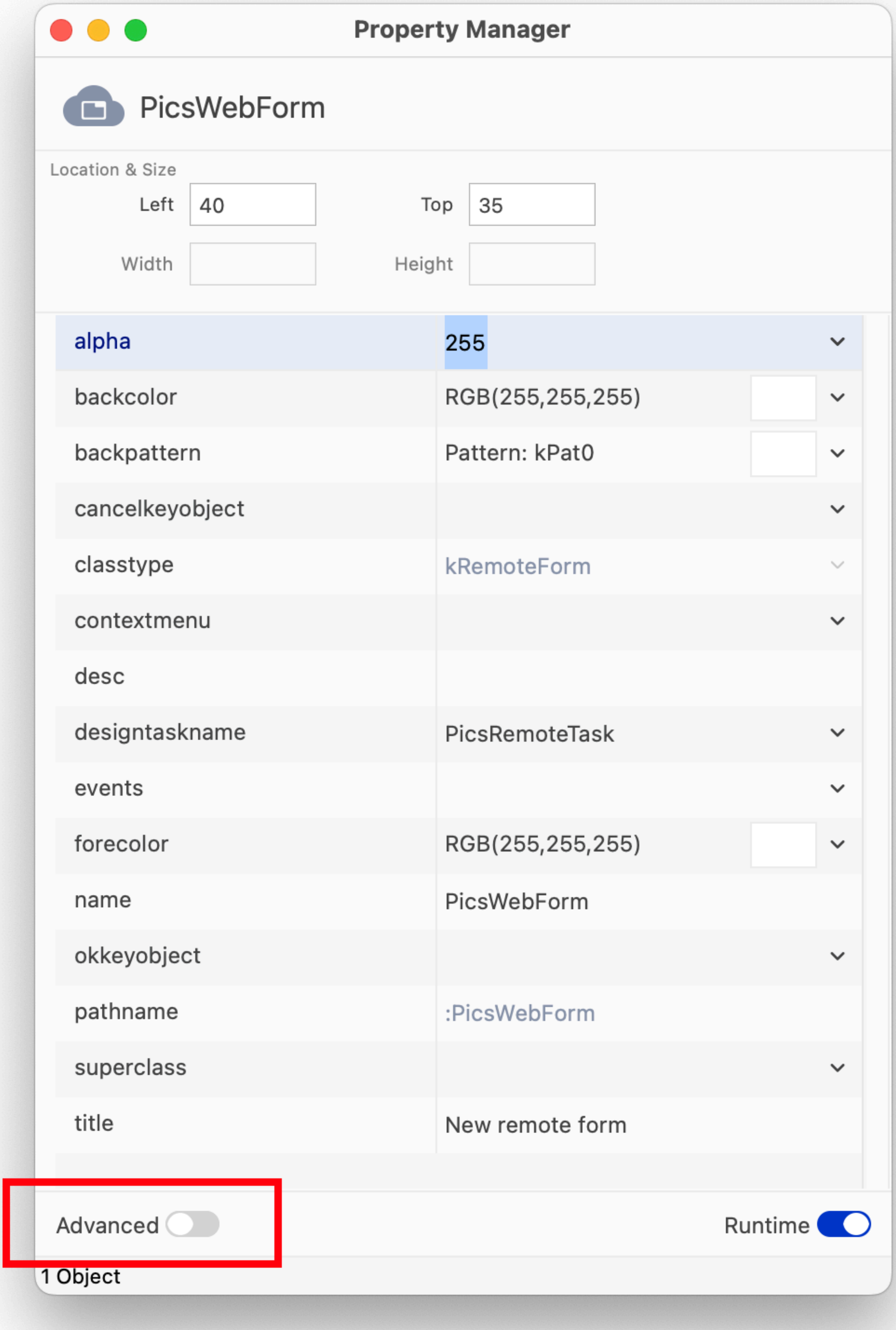
The 'search' item in the keys.json file allows you to hide or show the Search box in the Property Manager, Catalog, and Interface Manager.

Displaying all properties

You can display all the properties for a class or object on a single tab by entering * in the search box (in effect, this matches and displays all properties).

Property Tab

After you have searched the property list, you can then Right-click on a property, select the **Property Tab: <tab-name>** option to jump to that property on the specified tab. When there is no active search, the menu option is disabled, but still indicates the tab for the property.



Property Manager

PicsWebForm

Location & Size

Left 40 Top 35

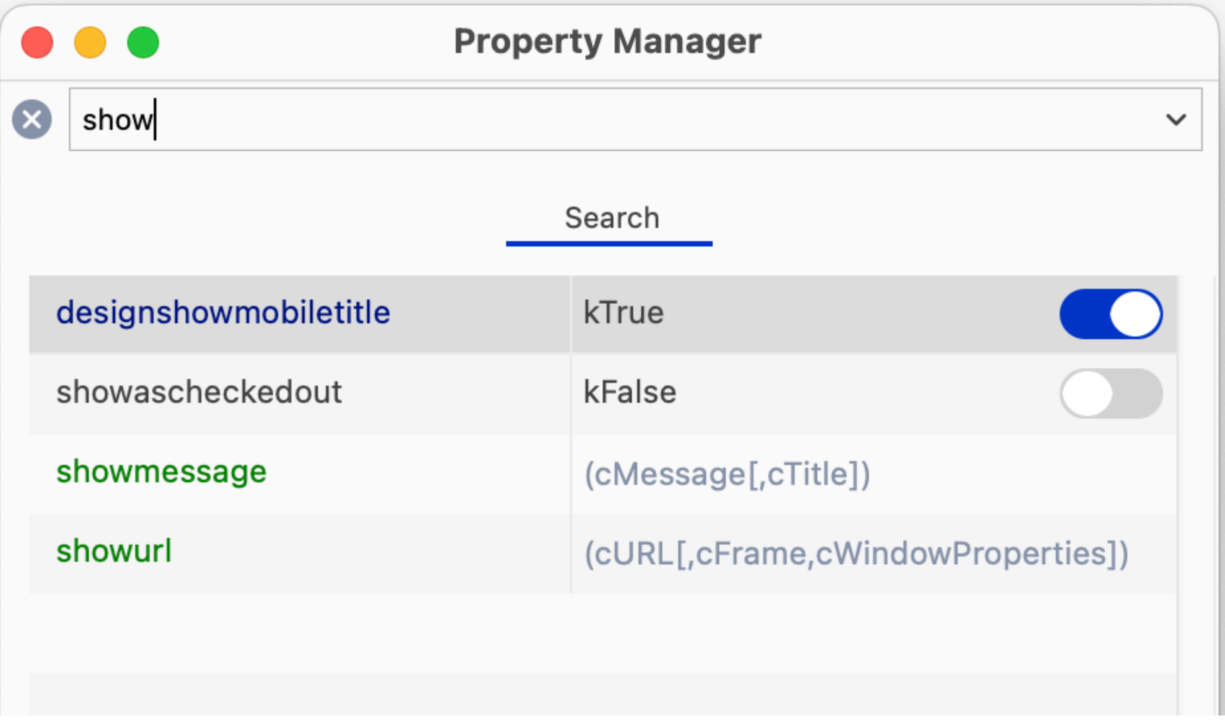
Width Height

alpha	255
backcolor	RGB(255,255,255)
backpattern	Pattern: kPat0
cancelkeyobject	
classtype	kRemoteForm
contextmenu	
desc	
designtaskname	PicsRemoteTask
events	
forecolor	RGB(255,255,255)
name	PicsWebForm
okkeyobject	
pathname	:PicsWebForm
superclass	
title	New remote form

Advanced ○ Runtime ●

1 Object

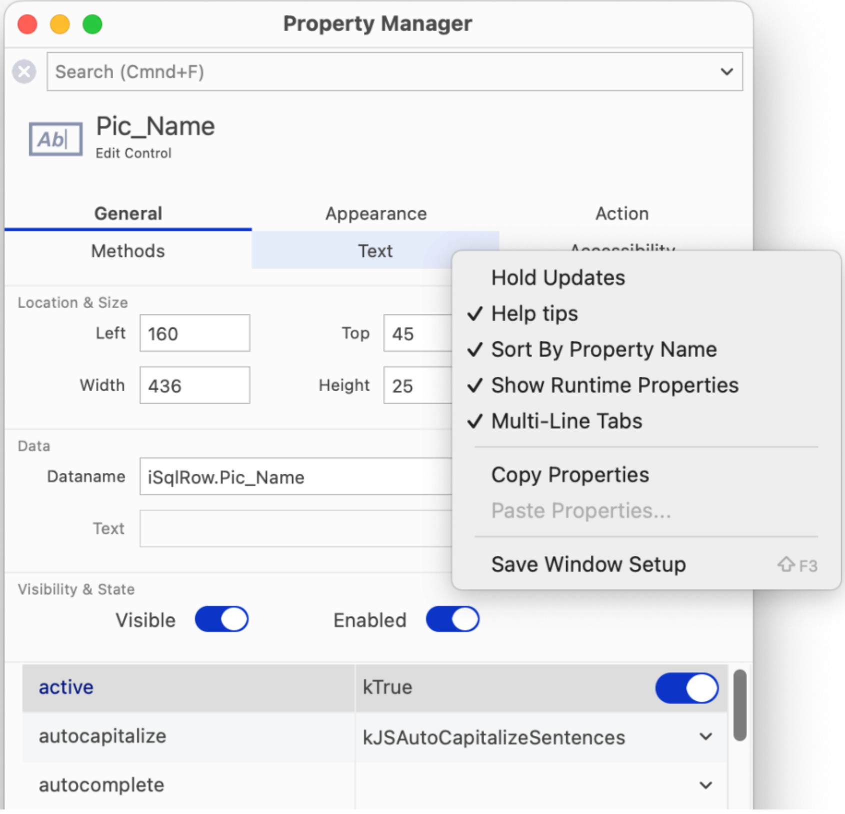
Figure 36:



show

designshowmobiletitle	kTrue
showascheckedout	kFalse
showmessage	(cMessage[,cTitle])
showurl	(cURL[,cFrame,cWindowProperties])

Figure 37:



Pic_Name
Edit Control

Location & Size

Left 160 Top 45

Width 436 Height 25

Data

Dataname iSqlRow.Pic_Name

Text

Visibility & State

Visible ● Enabled ●

active	kTrue
autocapitalize	kJSAutoCapitalizeSentences
autocomplete	

Figure 38:

Save Window Setup

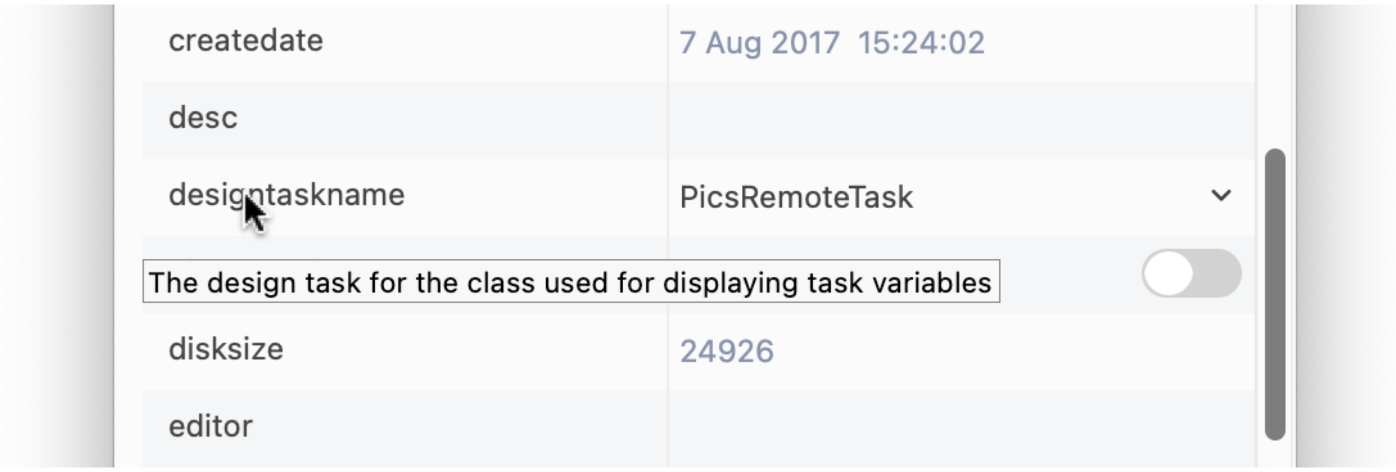
You can configure the Property Manager using its context menu, which you can open by right-clicking on its background. In addition, you can resize the window, and drag the column separator to resize the columns. When you have set up the Property Manager how you want it, you can save the layout using the **Save Window Setup** option in the context menu.

The properties of an object are shown in the Property Manager in alphabetical order by default, but you can list them functionally by unchecking the Sort by Property Name option in the Property Manager context menu.

The other options in the Property Manager context menu affect the behavior of the Property Manager. If set, the **Hold Updates** option stops the Property Manager updating its contents automatically when you click on another object. For example, you can click on a Remote form class to show its properties in the Property Manager, select the Hold Updates option, click on a field in the Remote form, and the Property Manager still displays the properties of the Remote form. Most of the time however you'll want this option turned off so the Property Manager updates itself automatically.

Property and Method Descriptions

The **Help tips** option in the Property Manager context menu (checked by default) displays short descriptive help messages for each property in the Property Manager: this is particularly useful for showing the parameters of a method. For example, the following screenshot shows the Help tip for the $designtaskname property for a remote form.



createdate	7 Aug 2017 15:24:02
desc	
designtaskname	PicsRemoteTask
disksize	24926
editor	

The design task for the class used for displaying task variables

Figure 39:

The **Show Runtime Properties** option lets you view properties that are normally available in runtime only, that is, properties of an instance rather than a design class. When runtime properties are visible in the Property Manager the methods for the instance are also shown. You cannot set runtime properties or run methods from the Property Manager itself, but you can drag references to them into your code in the method editor. The help tip for a method shows its parameters and description; for example, the following screenshot shows the Help tip for the $pushdata method for a remote form.

When methods are visible in the Property Manager, the help tips show a description and the parameters for each method.

Note you can drag a property or a method (and its parameters) from the Property Manager into the Method Editor when you are writing code: see 'Dragging Methods' below.

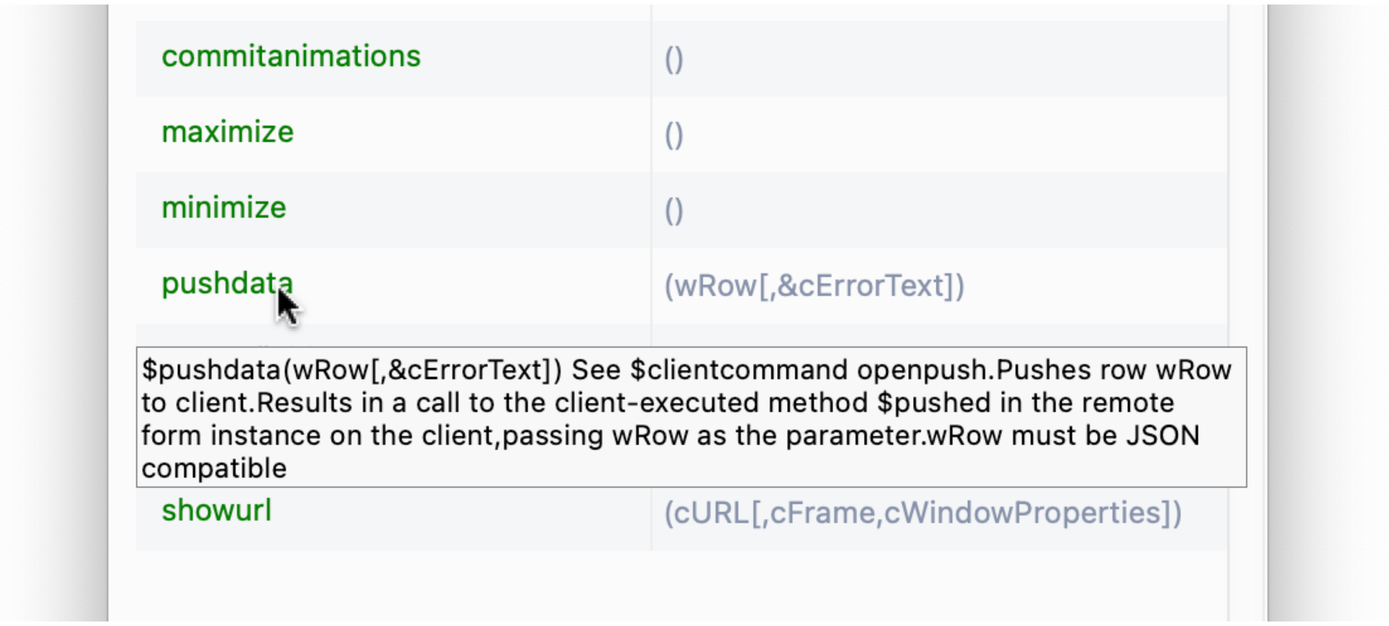
Copying Properties and Values

The **Copy Properties** option lets you copy all the properties of one object and paste or 'apply' them to another object of the same type: this is useful if you want to reproduce or duplicate a particular object, or even apply the properties of a built-in field to an external component. Only those generic properties that are contained in the source and target objects are copied.

The **Copy Value** option allows you to copy the value of a property, even if it is grayed out, such as when a class is not checked out of the VCS.

Show Property Context Menu

You can specify any built-in property to be displayed in the window or report field context menu by Right-clicking on the property in the Property Manager (when a field is selected) and selecting **Show property using editor context menu.** For example, the context menu for window and remote form editors shows 'Show $order', which allows you to display field ordering, but you can change it to most other properties. The field $ident is shown in the report editor context menu. This does not work for properties specific to external components, so for such properties the 'Show property…' option is grayed out.



commitanimations	()
maximize	()
minimize	()
pushdata	(wRow[,&cErrorText])
showurl	(cURL[,cFrame,cWindowProperties])

$pushdata(wRow[,&cErrorText]) See $clientcommand openpush.Pushes row wRow to client.Results in a call to the client-executed method $pushed in the remote form instance on the client,passing wRow as the parameter.wRow must be JSON compatible

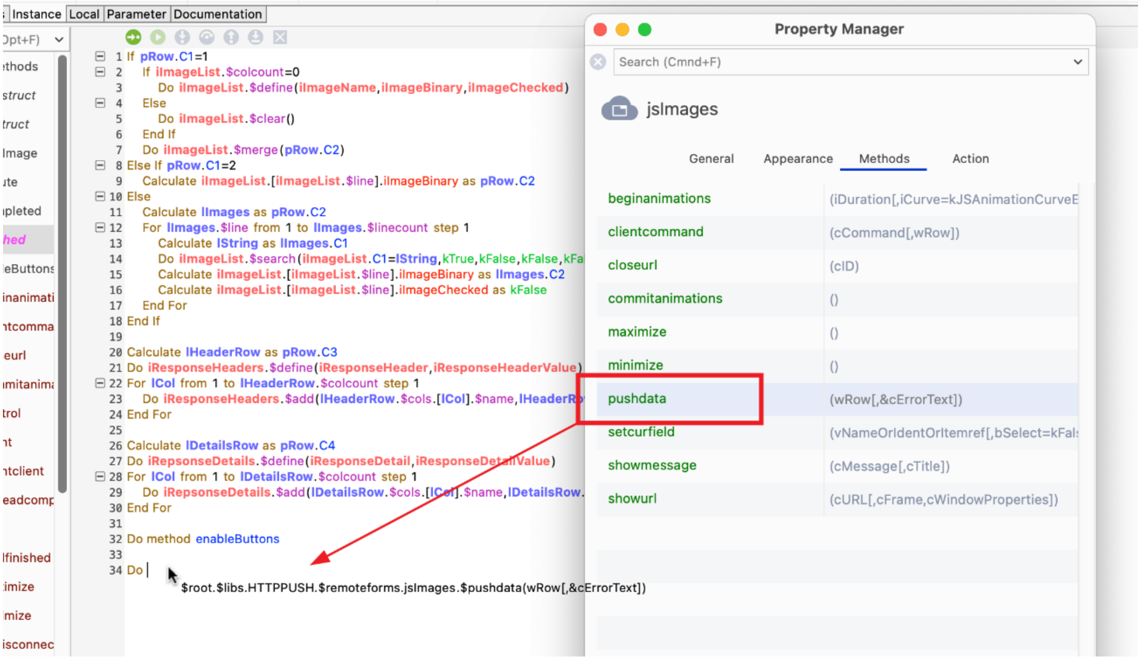
Figure 40:

The **Save Window Setup** option saves the selected property. A different property can be saved for each class editor.

The property value is displayed for background objects. If a property is not supported by an object, then nothing is displayed for that object.

Dragging methods from the Property Manager

Apart from displaying the properties and methods of classes and objects, the Property Manager lets you transfer a property or method and its notational path to your Omnis code in the Code Editor (Method Editor). You do this by dragging the property or method out of the Property Manager and dropping it onto your code in the method editor, for example, you could drag a method, such as the $pushdata() remote form method, and drop it onto the calculation part of the *Do* command.



```
If pRow.C1=1
   If iImageList.$colcount=0
      Do iImageList.$define(iImageName,iImageBinary,iImageChecked)
   Else
      Do iImageList.$clear()
   End If
   Do iImageList.$merge(pRow.C2)
Else If pRow.C1=2
   Calculate iImageList.[iImageList.$line].iImageBinary as pRow.C2
Else
   Calculate lImages as pRow.C2
   For lImages.$line from 1 to lImages.$linecount step 1
      Calculate lString as lImages.C1
      Do iImageList.$search(iImageList.C1=lString,kTrue,kFalse,kFalse,kFa
      Calculate iImageList.[iImageList.$line].iImageBinary as lImages.C2
      Calculate iImageList.[iImageList.$line].iImageChecked as kFalse
   End For
End If

Calculate lHeaderRow as pRow.C3
Do iResponseHeaders.$define(iResponseHeader,iResponseHeaderValue)
For lCol from 1 to lHeaderRow.$colcount step 1
   Do iResponseHeaders.$add(lHeaderRow.$cols.[lCol].$name,lHeaderRo
End For

Calculate lDetailsRow as pRow.C4
Do iRepsonseDetails.$define(iResponseDetail,iResponseDetailValue)
For lCol from 1 to lDetailsRow.$colcount step 1
   Do iRepsonseDetails.$add(lDetailsRow.$cols.[lCol].$name,lDetailsRow.
End For

Do method enableButtons

Do |
    $root.$libs.HTTPPUSH.$remoteforms.jsImages.$pushdata(wRow[,&cErrorText])
```

beginanimations	(iDuration[,iCurve=kJSAnimationCurveE
clientcommand	(cCommand[,wRow])
closeurl	(cID)
commitanimations	()
maximize	()
minimize	()
pushdata	(wRow[,&cErrorText])
setcurfield	(vNameOrIdentOrItemref[,bSelect=kFal:
showmessage	(cMessage[,cTitle])
showurl	(cURL[,cFrame,cWindowProperties])

Figure 41:

When you drag a method out of the Property Manager and drop it onto your code, its full path and parameters are copied as well. You can click in your code and edit the path and replace the parameters with your own variable names.

Setting Location and Size Properties

You can change the **Location** or **Size** of an object in the Property Manager using the +, -, *, or / keys plus a number of pixels, for example, you can enter +20 in the $left property in the Property Manager to move the object 20 pixels to the right. The location and size properties appear in the top panel of the Property Manager and include the $left, $top, $width, and $height properties. This also works for a group of selected objects where the property value is the same for all objects in the group (if the value is different among the selected objects in the group, the property value is blank).

Key	Description	Example for $left
+n	Adds n pixels to property value(s)	+20 moves object(s) 20 pixels to the right
-n	Subtracts n pixels from property value(s)	-20 moves object(s) 20 pixels to the left
*n	Multiplies property value(s) by n	*2 doubles the value, moves object(s) to the right
/n	Divides property value(s) by n	/2 halves the value, moves object(s) to the left

Selecting Integer Values

You can use the **Shift+Up** or **Shift+Down** Arrow keys to cycle through integer property values in the Property Manager, for example, when editing font sizes you can click into the property and use the Shift+Up/Down Arrow keys to increase or decrease the font size.

When increasing $fontsize in the Property Manager, labels and text objects in Remote forms and Window classes will increase their height, if necessary, in order to correctly display a single line of text using the increased font size.

Changing Boolean Properties

You can double-click on a Boolean (kTrue/kFalse) property value in the Property Manager to toggle its value (as well as clicking the switch). No other properties can be changed in this way by double-clicking.

Tab and Focus Selection

The Property Manager tries to restore (by searching for name) the last tab that you selected (this is reset when closing and re-opening the Property Manager window). In addition, you can use the tab key to give the tabbed pane the focus, and then use the Left and Right arrow keys to switch tabs. You can tab out of the tabbed pane using the tab key, and in the Property Manager you can also do this using the Up or Down arrow key.

Method Editor

The **Method Editor** is the main tool you use for programming or coding methods for the objects and classes in your application. It has a bulti-in free-type Code Editor, a powerful and comprehensive debugger that you can use to debug local and server code, as well as a useful method checker. See Method Editor in the Debugging Methods section.

Interface Manager

The **Interface Manager** displays the *public* methods and properties for objects in Omnis Studio, that is, any class that can contain methods and can be instantiated, including remote form, remote task, report, table, and object classes (not available for code classes since they cannot be instantiated). For each class or object, the Interface Manager displays all built-in methods, including those that are available in the instance of the class, as well as any custom methods you have added to the object. For each method in the class or object, the Interface Manager displays the method name, its parameters, return types, and a description, if any are present.

Private methods, namely methods with a name that does not begin with a dollar sign, are not included in the Interface Manager since these methods are confined to the class or instance.

You can view the Interface Manager for a class via its context menu or the method editor.

To view the Interface Manager

- Right-click on the class in the Browser and select Interface Manager from the context menu or from the method editor

- Open the method editor for the class
- Select View>>Interface Manager from the method editor menubar

The **Interface Manager** contains a list of objects in the class, that is, for windows and reports a list of window or report fields, for toolbars a list of toolbar controls, and for menus a list of menu lines. For other types of class or instance that do not contain objects, such as object classes, the Interface Manager contains the class methods only. You can click on each object or field in the left-hand list to view its methods. Built-in methods are shown in the color specified in the $nosetpropertycolor preference. Inherited methods are shown in the color specified in the $inheritedcolor preference. The properties tab similarly shows the object's properties.

The Details pane shows the parameters for the currently selected method. It also lets you add a description for your own custom methods. The status bar shows the return type for built-in methods, but not for your own methods since these can return any type.

The View menu on the Interface Manager menubar lets you open the method editor for the class, in addition to hiding or showing the built-in methods and details pane.

Dragging methods from the Interface Manager

You can drag a method or property from the method list and drop it on to an edit field in the method editor, or you can use copy and paste to do the same thing. The method name is prefixed by a variable name, such as "var_name.$methodname()" if you opened the Interface Manager by right-clicking on a variable of type Object. Otherwise the method name is prefixed by a dot, such as ".$methodname()", suitable to concatenate onto a variable name or some notation in the method editor. In all cases the parameters for the method are copied too, so they can be used as a template for the actual values you pass to the method.

Searching the Interface Manager

The search box on the **Interface Manager** allows you to search for methods and properties in the class. The search results are all matching methods and properties (still associated with their class or field), together with any names in the field name list which match the search string.

The search function allows you to cycle through the results using some new keys (in the IDE section of keys.json), **searchNext** and **searchPrev** which are mapped to Ctrl+G and Ctrl+Shift+G, respectively. Using search next/previous in the Catalog cycles through all tabs and fields identified by the search that contain at least one matching item. So, for example, if field *test* has matches in both methods and properties, Ctrl+G will go to the methods first, and the next Ctrl+G will go to the properties. If a control has no methods, but it does have properties, Ctrl+G will go directly to the properties tab, and vice versa.

The 'search' item in the keys.json file allows you to hide or show the Search box in the Interface Manager, Catalog, and Property Manager.

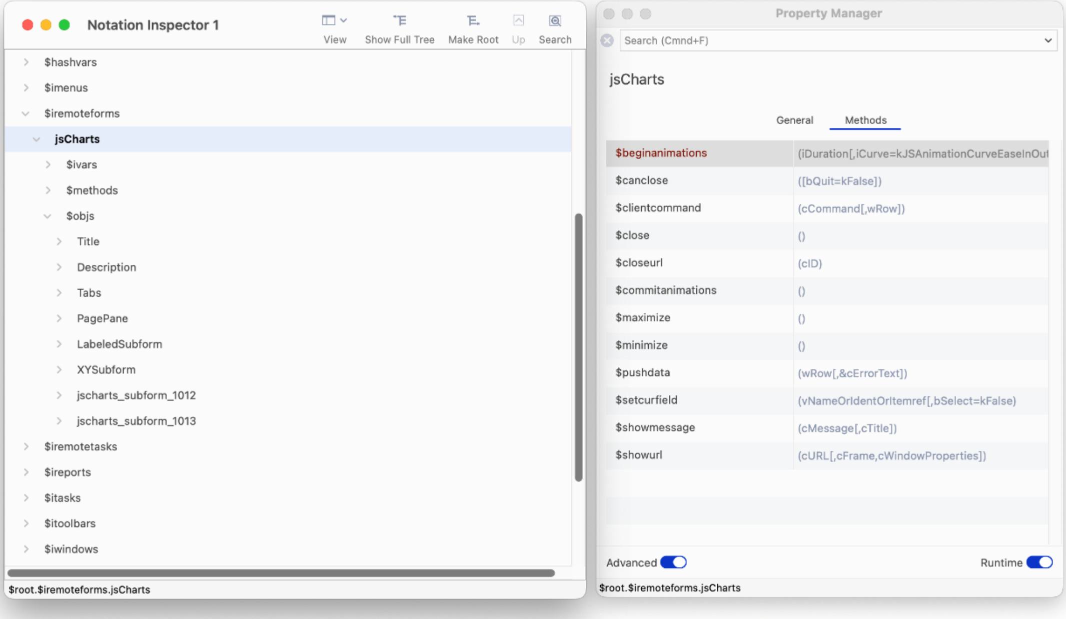
Notation Inspector

Omnis structures its objects in an object tree, or hierarchical arrangement of objects and groups that contain other objects. The complete tree contains all the objects in Omnis itself, together with your design libraries, classes, and other objects created at runtime, such as remote form instances. You can view the complete object tree in the **Notation Inspector,** which you can open from the **View** menu, or by clicking on the icon in the main Omnis toolbar, or by pressing F3/Cmnd-3.

Using the Notation Inspector

To facilitate a system of naming or referring to an object in the object tree, and its properties and methods, Omnis uses a system called the *notation*. The notation for an object is really the path to the object within the object tree. The full notation for an object is shown in the status bar of the Notation Inspector. You can use the notation to execute a method or to change the properties of an object in your Omnis code, and you can use a notation string anywhere you need to reference a variable or field name.

The *Notation Inspector* therefore lets you view the Omnis object tree from the $root object down. It is particularly useful for viewing the contents of your library and finding the right notation for a particular object or group of objects in your library. For example, you can get the notation for an object on a design form or remote form instance using the Notation Search tool on the Notation Inspector toolbar. When you click on an object or group in the Notation Inspector the Property Manager will display its properties and methods: note the Advanced option must be enabled to view all the properties for the current object.



```
$hashvars
$imenus
$iremoteforms
  jsCharts
    $ivars
    $methods
    $objs
      Title
      Description
      Tabs
      PagePane
      LabeledSubform
      XYSubform
      jscharts_subform_1012
      jscharts_subform_1013
$iremotetasks
$ireports
$itasks
$itoolbars
$iwindows
```

$root.$iremoteforms.jsCharts

jsCharts

$beginanimations	(iDuration[,iCurve=kJSAnimationCurveEaseInOut
$canclose	([bQuit=kFalse])
$clientcommand	(cCommand[,wRow])
$close	()
$closeurl	(cID)
$commitanimations	()
$maximize	()
$minimize	()
$pushdata	(wRow[,&cErrorText])
$setcurfield	(vNameOrIdentOrItemref[,bSelect=kFalse)
$showmessage	(cMessage[,cTitle])
$showurl	(cURL[,cFrame,cWindowProperties])

$root.$iremoteforms.jsCharts

Figure 42:

The Notation Inspector lets you drill down the hierarchy of objects within the Omnis tree: for example, you can view the object tree for an open remote form in the $iremoteforms group, shown above: when you click on an open remote form name, jsCharts in this case, the Property Manager displays the runtime properties and methods of the form.

When the Notation Inspector opens it shows the $root object which contains all the objects in the Omnis object tree including all your open libraries and their contents. It also includes all the objects and groups created at runtime when you run your application. You can expand each branch of the tree to show the contents of an object or group.

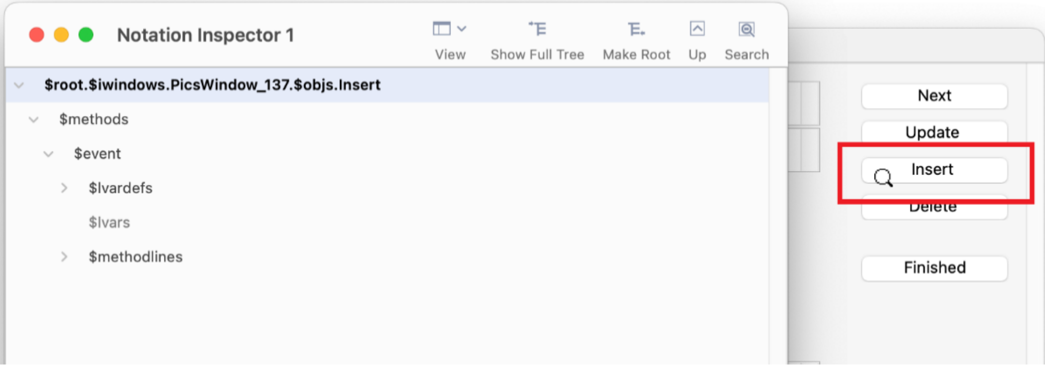
All the different types of class in your library are shown in Notation Inspector within their respective object group. For example, all the remote form classes in your library are contained in the $remoteforms group, while the $classes group contains *all* the classes in your library. Also note that a group may be empty if your library does not contain any classes of that type.

Dragging notation from the Notation Inspector

Having found the object you're interested in, you can copy its full notation to the clipboard and paste anywhere in your library, or you can drag an item from the Notation Inspector and drop it onto your code in the method editor. For example, you can drag the item into the calculation part of the *Do* command in the Code Editor.

Finding the notation for an object

You can find the notation for a window or toolbar object in an instance of one of these classes. To do this, click on the Notation Search button (spyglass icon) on the Notation Inspector toolbar then click on the object in your open window or toolbar (this works for desktop apps only, not web-based remote forms). The Notation Inspector will refresh itself showing the notation for the object you clicked on. The object becomes the root of the tree, so you can expand the tree to view its contents.



```
$root.$iwindows.PicsWindow_137.$objs.Insert
  $methods
    $event
      $lvardefs
      $lvars
      $methodlines
```

Figure 43:

You can click the Search button, then click on a button or other object on an open window and view its notation in the Notation Inspector: the tree is redrawn and you can drill down to the view the contents of the object.

Catalog

The **Catalog** lists all the Functions, Constants, Event codes, and Hash variables in Omnis, in addition to listing all the *Variables, Schema class columns, Query class columns*, *User constants,* and *String tables* in your library. The Catalog lists all the *variables* for the current object including Task, Class, Instance, Local, and Parameter variables, as well as Event parameters. For example, if you are working in a Remote Form class, the Catalog will show the class and instance variables for that class; for example, the screenshot below shows the Instance variables for a Remote form instance. In addition, when you select a particular method in the Method Editor, the Catalog will list the Local and Parameter variables for that method.

The values column is available for the Variables, Constants, Events, and Hash variable group tabs. For example, you can view the values for all instance variables under the Variables tab (assuming there is an instance open), as above. The values column is displayed as a third column on the right-hand side of the Catalog window, under each tab, and will show the current value of the variables or other items.

Searching the Catalog

The **Search** box at the top of the Catalog window allows you to search for an item *on the current tab*. You can type a word or part of a word into the search box and the Catalog list will update itself as you type. For example, you can search for a function or any functions containing the word 'text' under the Functions tab; in this case, there are several functions under different sub-groups on the main Functions tab, as follows:

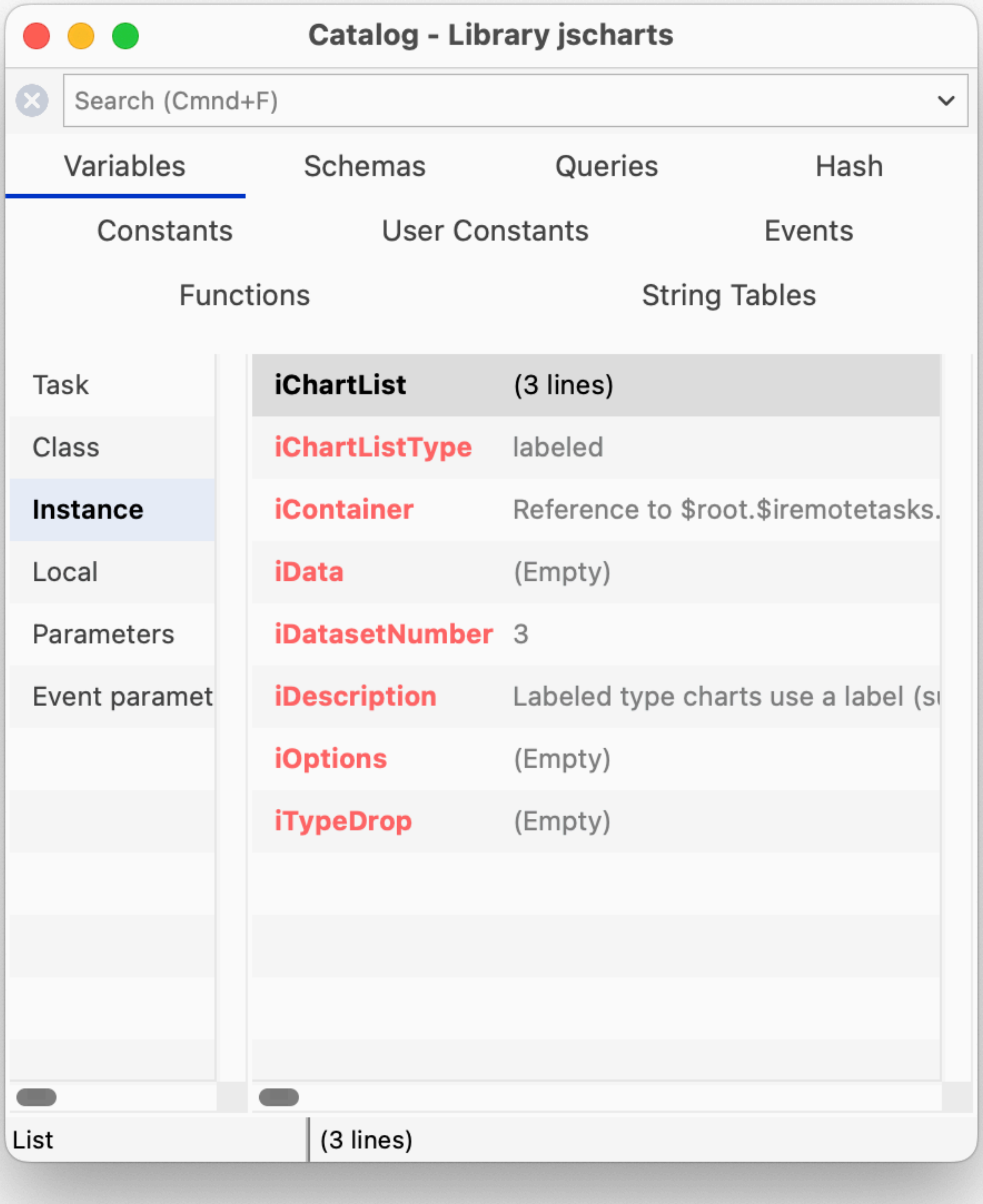
The search results are all matching items (still in their groups), together with groups that have no match where the group name matches.

You can cycle through the results using Ctrl+G and Ctrl+Shift+G for Next and Previous, respectively (these are stored as **searchNext** and **searchPrev** in the "ide" section of keys.json). Using search next/previous in the Catalog cycles through all tabs and fields identified by the search that contain at least one matching item.

The "search" item in the keys.json file allows you to hide or show the Search box in the Catalog, Property Manager, and Interface Manager.

Catalog Context Menu

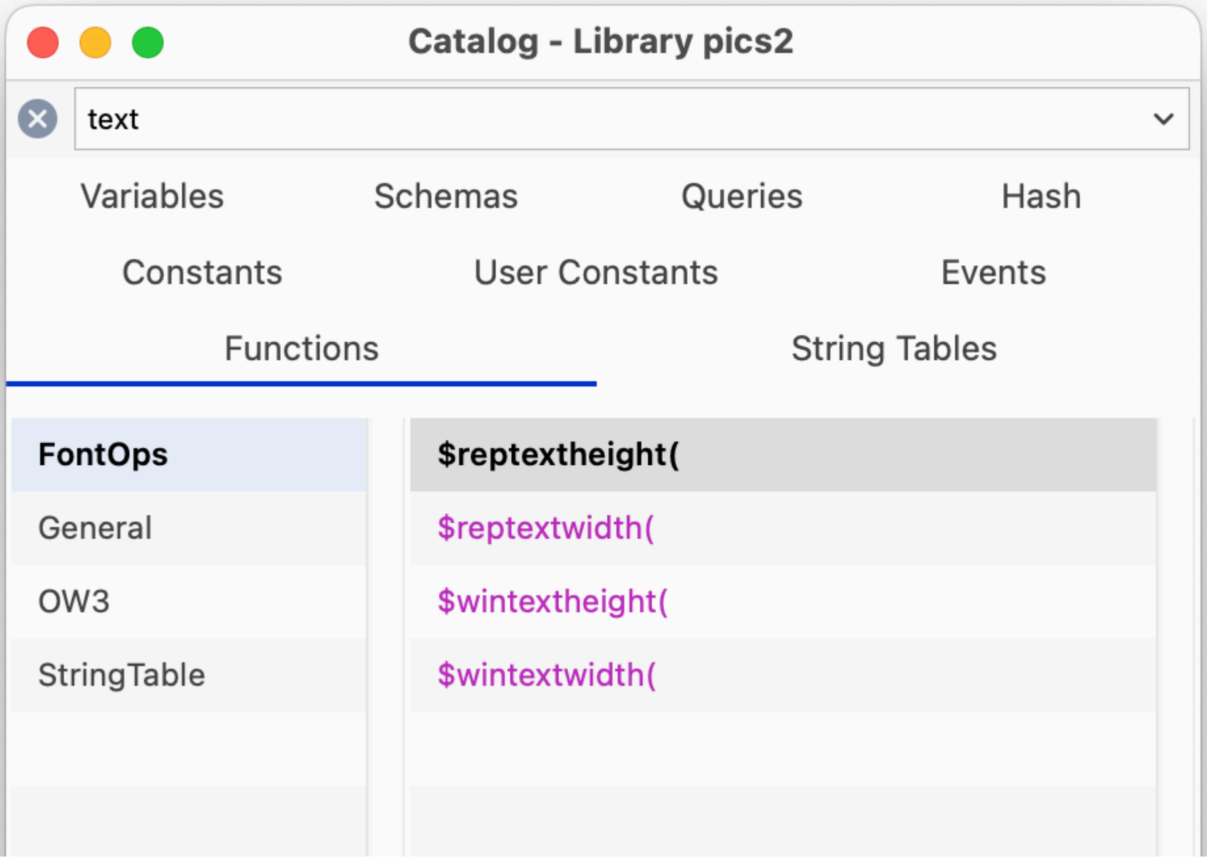
The context menu on the Catalog lets you change its appearance and style of tabs. The **Show status bar** option hides or shows the Catalog status bar. The **Show Values** option in the context menu hides or shows the values column (default is on). The **Help Tips** option enables or disables popup Help tips for Functions, Events and Constants.



iChartList	(3 lines)
iChartListType	labeled
iContainer	Reference to $root.$iremotetasks.
iData	(Empty)
iDatasetNumber	3
iDescription	Labeled type charts use a label (s
iOptions	(Empty)
iTypeDrop	(Empty)

List (3 lines)

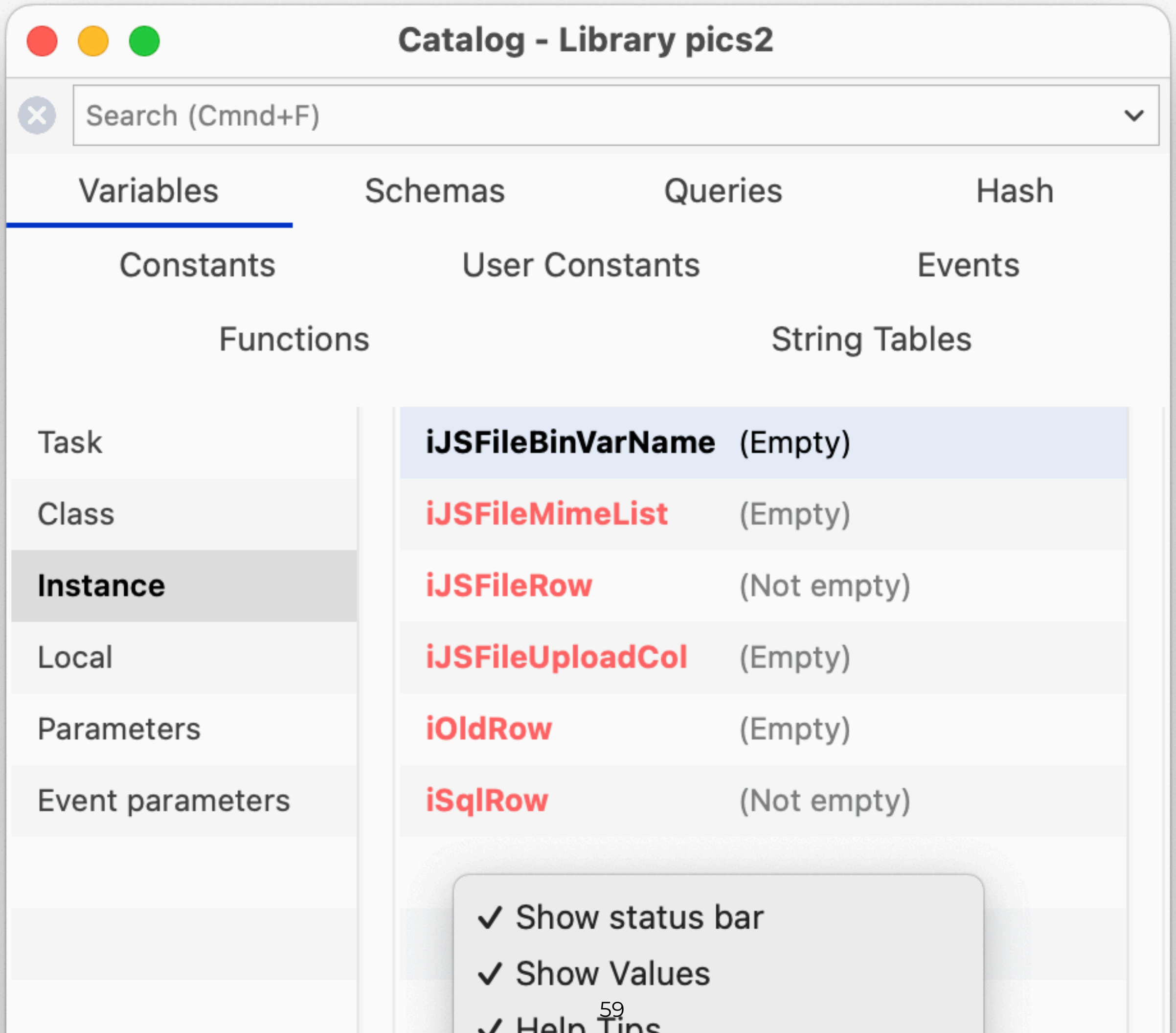
Figure 44:



FontOps
General
OW3
StringTable

$reptextheight(
$reptextwidth(
$wintextheight(
$wintextwidth(

Figure 45:



Task
Class
Instance
Local
Parameters
Event parameters

iJSFileBinVarName	(Empty)
iJSFileMimeList	(Empty)
iJSFileRow	(Not empty)
iJSFileUploadCol	(Empty)
iOldRow	(Empty)
iSqlRow	(Not empty)

Multi-Line Tabs option allows the tabs in the Catalog to wrap onto several lines; with this option unchecked, tabs are shown in a horizontal scrolling list. The **Show** option allows you to show or hide individual tabs in the Catalog. All these options are saved in the **Save Window Setup** option.

Syntax Colors

Items in the right-hand list of the Catalog are shown using the relevant syntax color, for variable types, constants, etc. The **catalogUsesSyntaxColors** item in the "ide" section of config.json can be used to control this behavior; the default is true.

Variable Context Menu

You can Right-click on any Variable (or field name) to view its type, current value and other information (to view an instance variable, the instance has to be open); the first Variable option open a separate window showing the value for the variable, e.g. a list variable displays a grid of list values. The following screenshot shows the context menu for an instance variable in a remote form instance.

Dragging items from the Catalog

When you have found an item in the Catalog, such as a Variable name, or a Schema coumn name, you can enter its name into your code in the Code Editor by double-clicking on it (assuming the cursor is in the appropriate place) or by dragging it out of the Catalog and dropping the item in the appropriate place into your code.

When using the Code Editor, you can drag items from the Catalog to the *Initial value* and the *Description* fields in the Variable pane: for this to work, the focus must be on the initial value or description field *before switching to the Catalog* to select the item.

SQL Browser

The SQL Browser lets you access and interact with many leading *server databases* or DBMSs, however specific database support will depend on the edition of Omnis Studio you are running: all versions give you access to **PostgreSQL** and **SQLite,** including the *Community Edition*, while other editions, including the *Professional Edition*, provide access to Oracle, MySQL, Sybase, and many other databases and data sources via ODBC.

Omnis Studio provides a separate ***Data Access Module*** (or DAM), or *Object DAM,* to connect directly to each server database, plus there is an ODBC DAM that allows you to access ODBC-compliant databases or data sources, such as MS SQL Server and SAP HANA.

Under the **Session Manager** option in the SQL Browser there will be a session template for each database supported in your version of Omnis Studio, and/or if you have the correct clientware installed on your computer; all versions should contain a template for **PostgreSQL** and **SQLite.**

Your version of Omnis may also contain an Omnis SQL DAM that lets you access an Omnis data file using SQL methods rather than the Omnis Data Manipulation Language (DML), but this is only provided for backwards compatibility and for running legacy applications and should not be used for new applications.

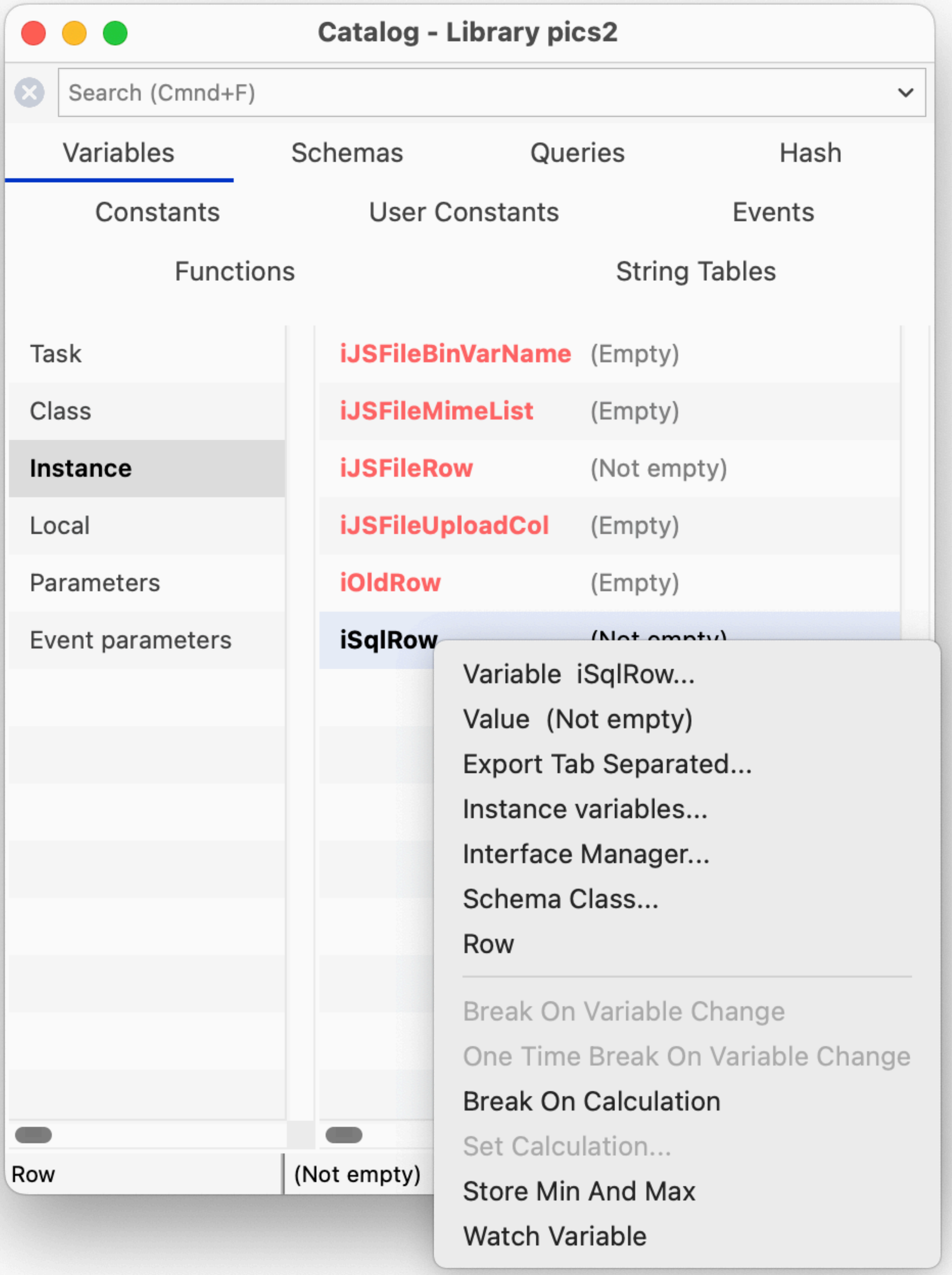
Creating and Editing a Session

In the SQL Browser, under the Session Manager option, you can create a new session, or duplicate an existing one and modify it adding your own connection parameters. For most purposes, the quickest and easiest way to create a new session is to duplicate an existing template and modify it.

To create a new session to your database

- Click on the **SQL Browser** option in the Studio Browser, then click on the **Session Manager** option
- Select the type of database you want to connect to and click the **Duplicate Session** option
- Rename the session using a suitable name for your database or project
- Double-click the session template and modify the parameters in the **Modify Session** dialog

When you create or modify a database session you need to select the **DBMS vendor,** the **Data Access Module** (DAM) and version, as well as specifying the **Username** and **Password** for the session. The connection parameters required will vary depending on your database.



iJSFileBinVarName	(Empty)
iJSFileMimeList	(Empty)
iJSFileRow	(Not empty)
iJSFileUploadCol	(Empty)
iOldRow	(Empty)
iSqlRow	(Not empty)

Row	(Not empty)

Figure 46:

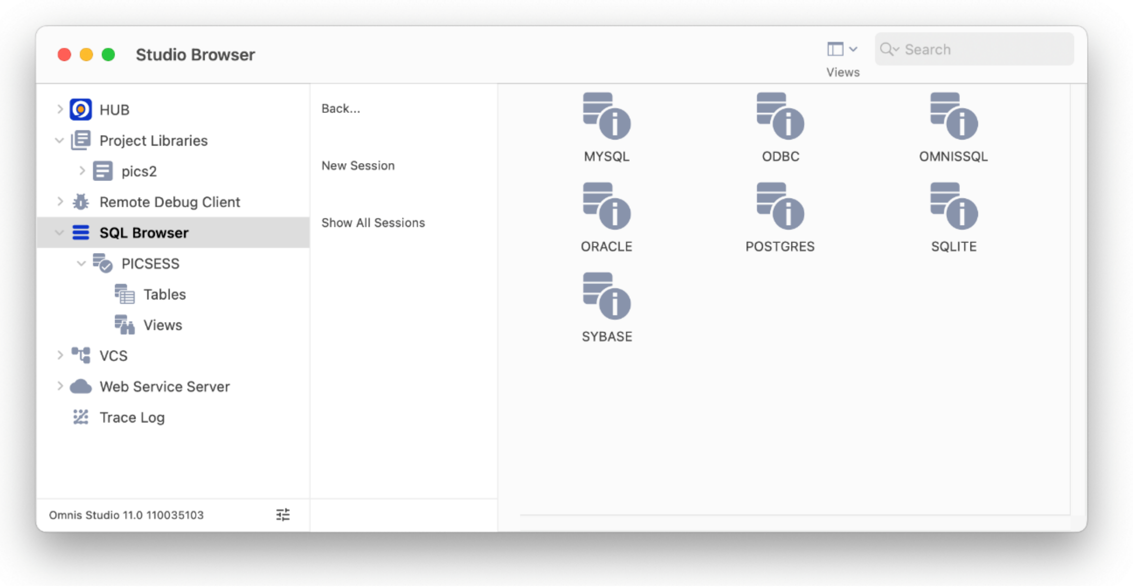
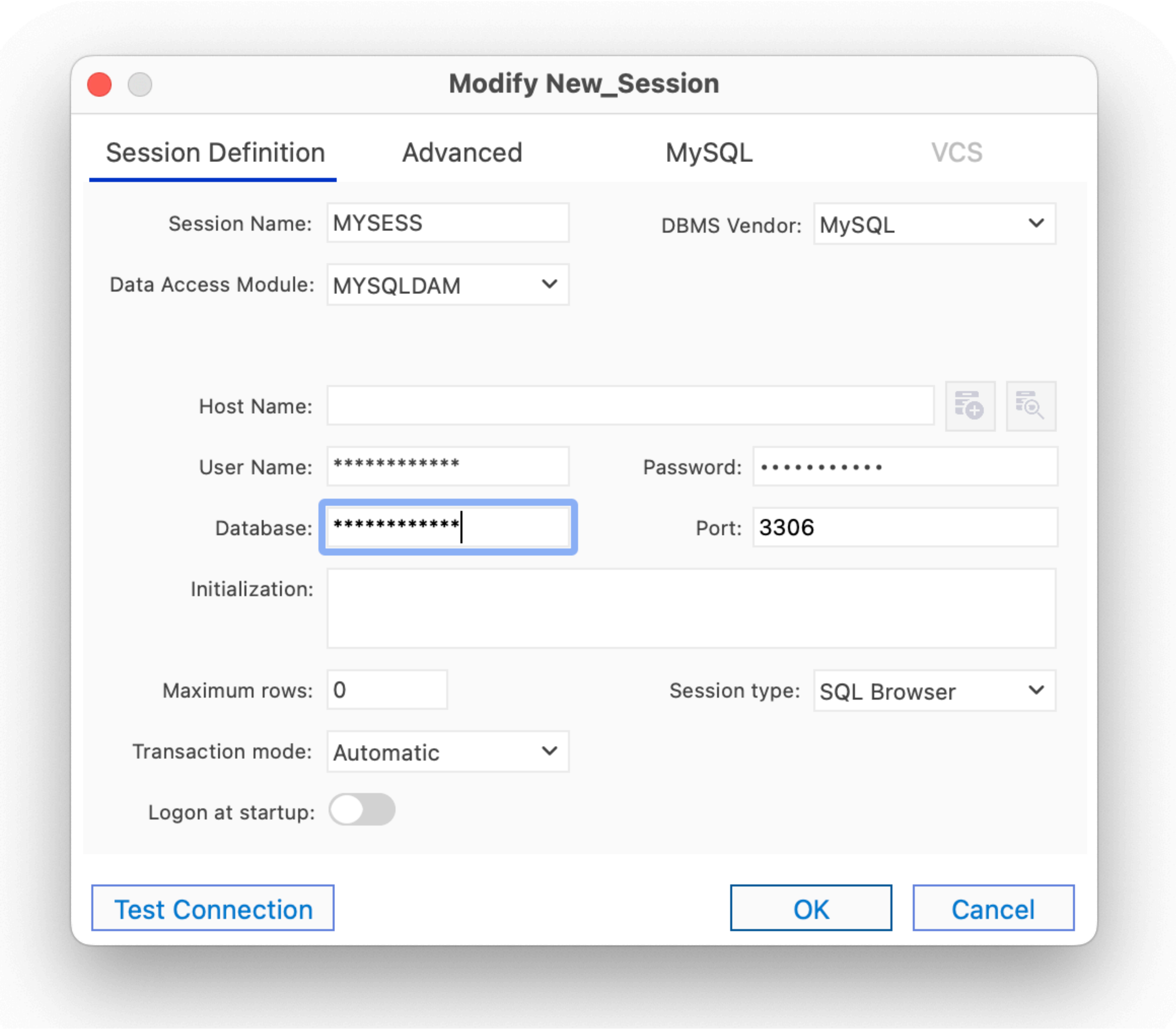


Figure 47:



Session Name:	MYSESS	DBMS Vendor:	MySQL
Data Access Module:	MYSQLDAM		
Host Name:			
User Name:	************	Password:	•••••••••••
Database:	************	Port:	3306
Initialization:			
Maximum rows:	0	Session type:	SQL Browser
Transaction mode:	Automatic		
Logon at startup:	○		

Figure 48:

Opening a session and accessing your data

Having defined or modified your session template, you can open or log onto your database session in the SQL Browser.

To open a database session

- Click on the **SQL Browser** option in the Studio Browser
- Click on the **Open Session** option and then the session name, e.g. PICSESS
- Double-click on the **Tables** object to view the tables on your database
- Right-click on a table name (e.g. MyPictures) and select the **Show Data** option

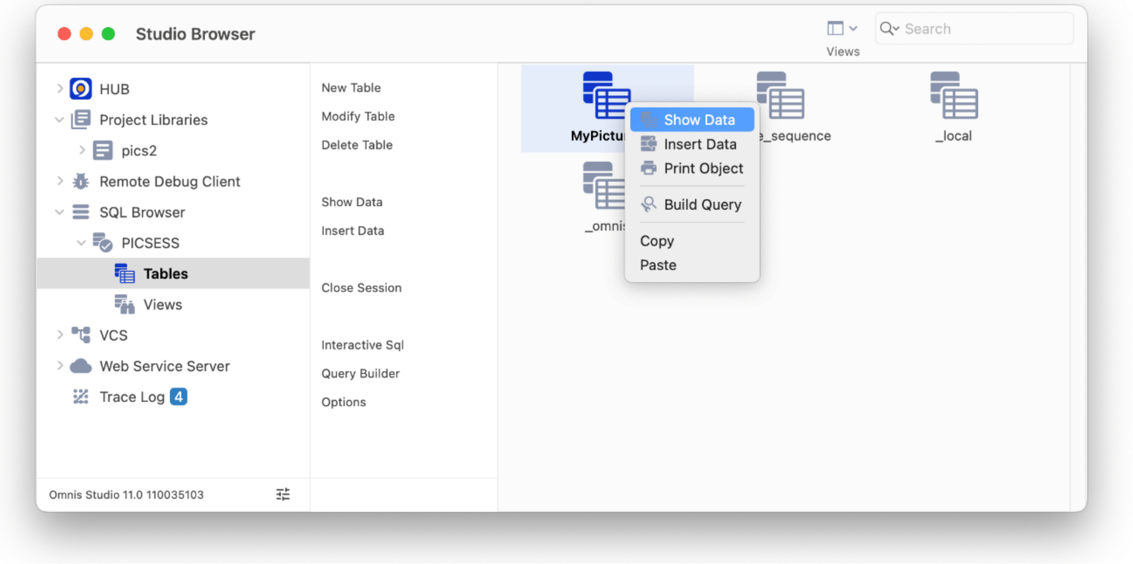
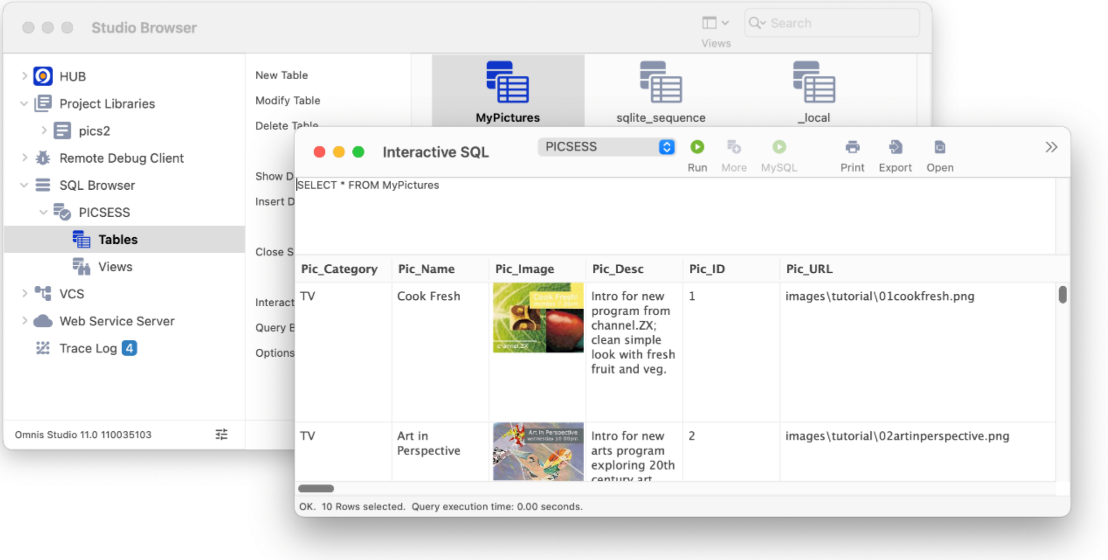


Figure 49:

The **Show Data** option opens the **Interactive SQL** (iSQL) tool which submits a basic SELECT on your table to display the data.



```
SELECT * FROM MyPictures
```

Pic_Category	Pic_Name	Pic_Image	Pic_Desc	Pic_ID	Pic_URL
TV	Cook Fresh		Intro for new program from channel.ZX; clean simple look with fresh fruit and veg.	1	images\tutorial\01cookfresh.png
TV	Art in Perspective		Intro for new arts program exploring 20th century art	2	images\tutorial\02artinperspective.png

OK. 10 Rows selected. Query execution time: 0.00 seconds.

Figure 50:

If you use the Show Data option on the SQLite database used in the tutorial, you can view the text and images in the database.

You can change the max line height in the result window using the context menu on the results grid; the setting is saved in the Save Window Setup.

Once you have created your database session you can view your data using the Interactive SQL tool, as above, or create windows and forms based on the session using the SQL Form Wizard. Note the Tutorial shows how you can build a SQL form to access data via a web browser on a desktop PC or mobile device.

You can resize the font in the iSQL (results pane) using the Ctrl+/Ctrl- shortcut keys.

To learn more about connecting to and interacting with a SQL database, see the SQL Programming chapter.

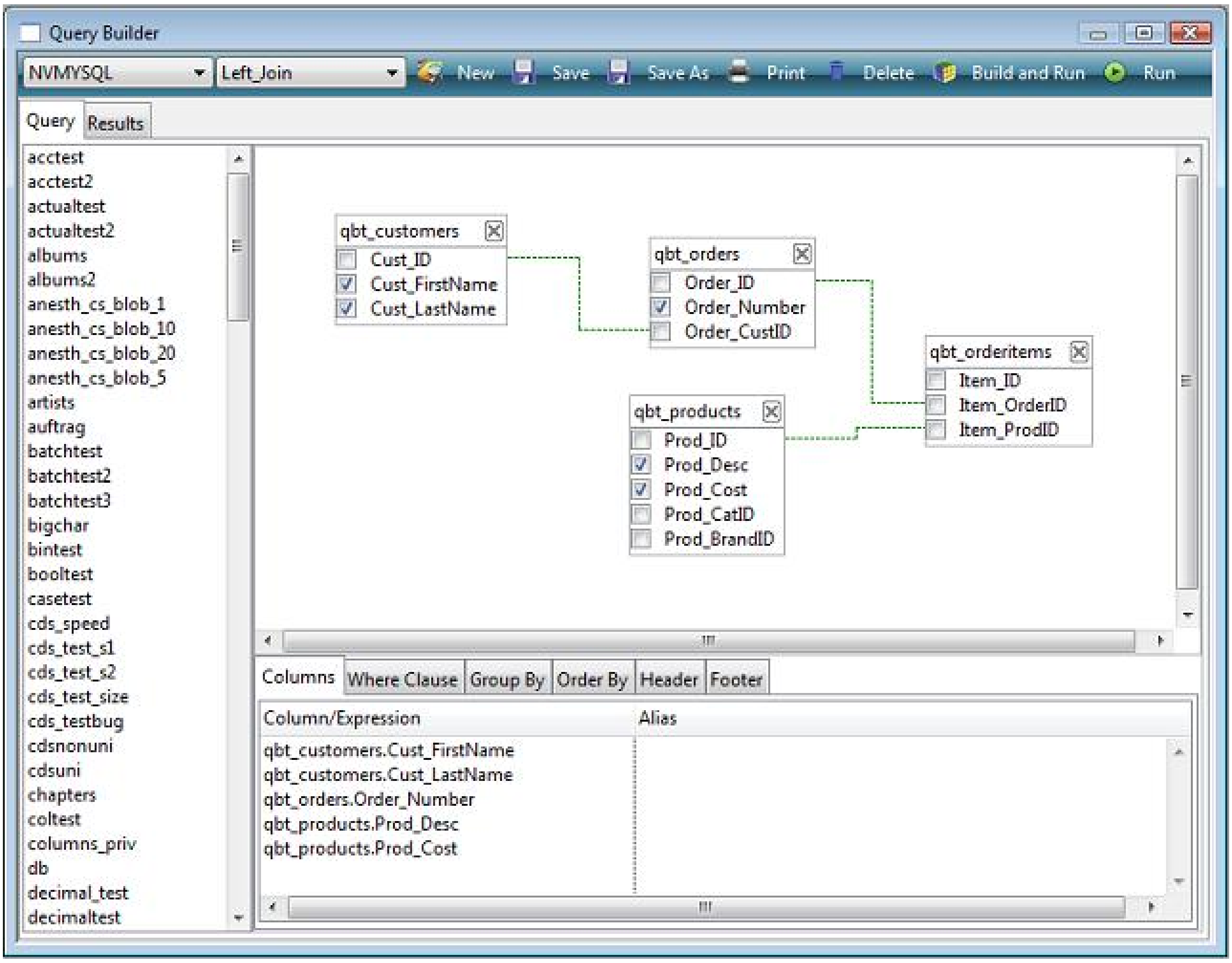
SQL Query Builder

The SQL **Query Builder** lets you build, run and store SQL Queries quickly and easily using a graphical interface. The Query Builder is built into the Omnis Studio IDE and is easily accessed via the SQL Browser. The Query Builder supports the generation of both simple queries requiring little SQL knowledge and more advanced queries using different join types and clauses. You can save queries for later use and you can create Omnis query classes based on your queries to allow you to take advantage of the queries you build in the Query Builder in your applications.

Note that like the SQL Browser, the databases supported in the SQL Query Builder may depend on the version of Omnis Studio you are running.

SQL Query Builder window

You can open the SQL Query Builder by clicking on the 'Query Builder' option when the SQL Browser option is selected in the main Studio Browser window. The Query Builder window has three main panes: the table pane on the left showing all available tables, the main design area at the top-right showing the query in graphical format, and the lower tab pane for defining aliases, SQL clauses and expressions.



Column/Expression	Alias
qbt_customers.Cust_FirstName	
qbt_customers.Cust_LastName	
qbt_orders.Order_Number	
qbt_products.Prod_Desc	
qbt_products.Prod_Cost	

Figure 51:

All panes in the Omnis Query Builder are resizable and can be saved via 'Save Window Setup' of the main context menu. The list of available tables may also be refreshed at any time via the table list context menu.

Creating a Query

The SQL Query Builder allows you to construct most types of query simply using drag and drop and the context menus available as appropriate in each pane of the main Query Builder window. In order to construct a query, you must be logged on to a SQL session via the SQL Browser.

The first time you open the Query Builder the query list will be empty allowing you to create a new query. If an existing query is currently displayed pressing either the 'New' toolbar button or selecting '<new query>' from the query dropdown list will clear the query from the screen.

Selecting a SQL Session

All open sessions are shown in the droplist at the top-left of the Query Builder window. You can select a session to build your query by dropping down the session list and selecting a session.

Adding a Table

Once a SQL session has been selected a list of tables for the current session is shown in the left hand pane. To include a table in a SQL Query simply drag the table into the main design area on the right.

Removing a Table

You can remove a table from a query either by clicking on the 'X' icon for the table or by opening the context menu for the table and selecting 'Remove Table'. You have to confirm if you want to remove a table from the current query.

Refreshing a Table

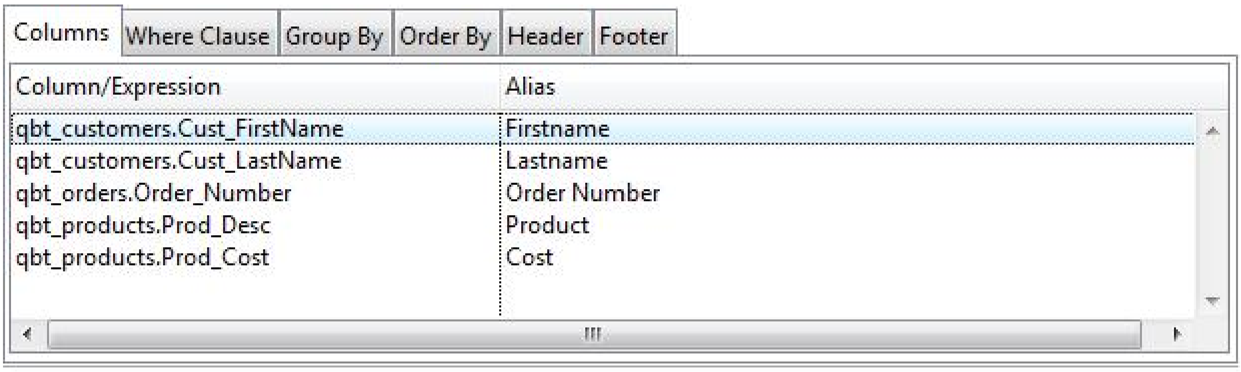
If a table has been modified outside the Query Builder, you can refresh the table in the Query Builder by right-clicking on the table and selecting 'Refresh Table' from the context menu. This option will add any new columns or remove any deleted columns to reflect the current state of the server table.

Selecting Columns

Once a table has been added, you can select columns by selecting their names individually or by right-clicking on the table you can check or uncheck all the columns in the table. Alternatively, you can specify that the Query Builder checks all columns automatically when a table is added using the 'Select all Columns on drop' option within the 'Options' dialog (see the Options section).

Adding Column Aliases

You can add an alias for each selected column in the 'Columns' tab of the lower tab pane by clicking in the 'Alias' column of the current line.



Column/Expression	Alias
qbt_customers.Cust_FirstName	Firstname
qbt_customers.Cust_LastName	Lastname
qbt_orders.Order_Number	Order Number
qbt_products.Prod_Desc	Product
qbt_products.Prod_Cost	Cost

Figure 52:

You can reorder columns in this pane by dragging and dropping column names in the list.

Creating Joins

To construct your queries, you can create joins between tables either using drag and drop in the main Query Builder design area or using the Table Joins dialog. To create a join, drag a column name from one table and drop it onto the column name within the table you wish to join with. Alternatively, you can right-click on a table to open the 'Table Joins' dialog from the context menu.

The 'Table Joins' dialog lets you modify joins, set the operator and type, re-order using drag and drop and switch columns using the context menu.

Joins may also be deleted via the context menu of the joined column in the main design area and Line and style preferences can be set via the 'Options' dialog.

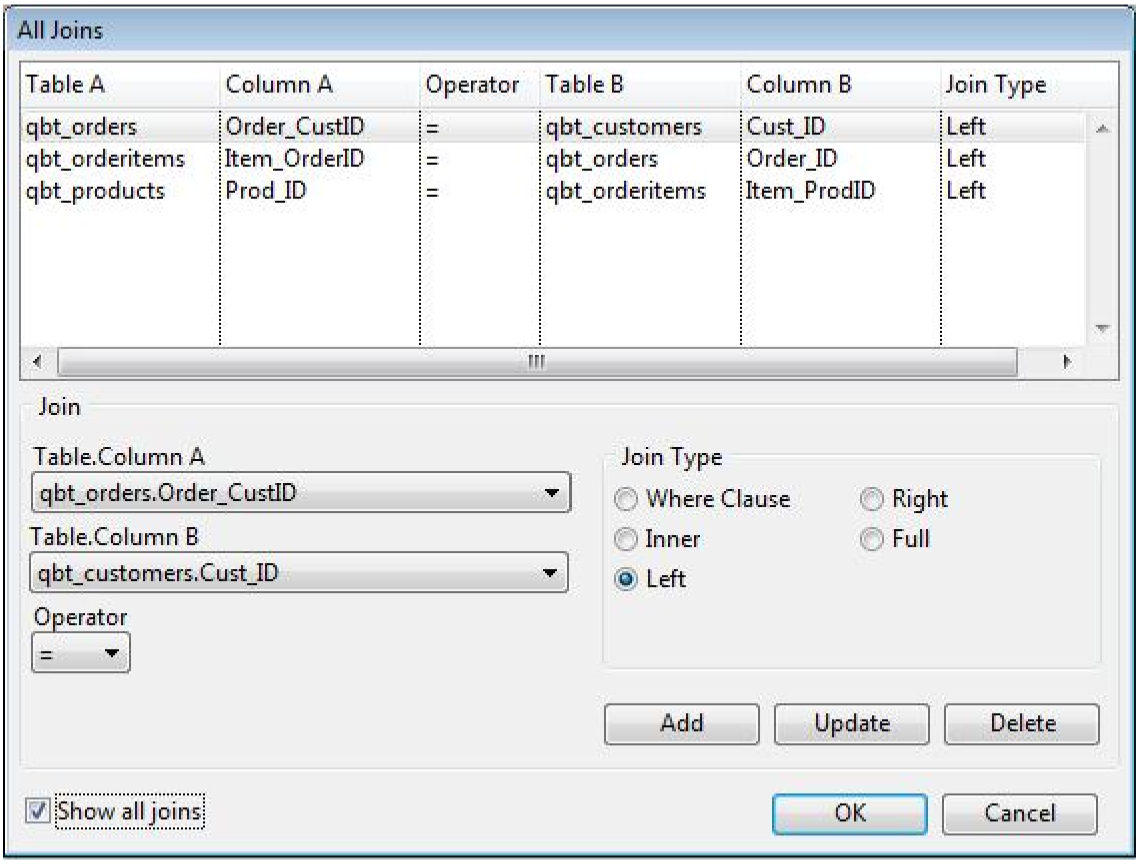


Table A	Column A	Operator	Table B	Column B	Join Type
qbt_orders	Order_CustID	=	qbt_customers	Cust_ID	Left
qbt_orderitems	Item_OrderID	=	qbt_orders	Order_ID	Left
qbt_products	Prod_ID	=	qbt_orderitems	Item_ProdID	Left

Table.Column A: qbt_orders.Order_CustID

Table.Column B: qbt_customers.Cust_ID

Operator: =

Join Type: ○ Where Clause ○ Right ○ Inner ○ Full ● Left

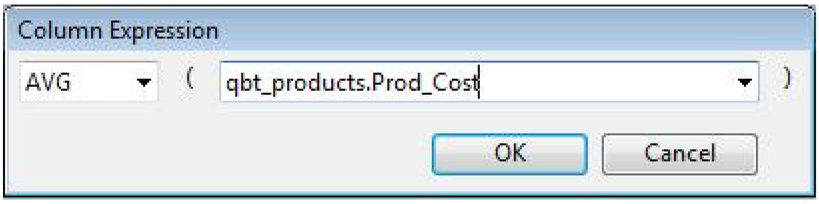
☑ Show all joins

Figure 53:

You can set the default Join type for queries using ‘Join Type’ option in the Joins window, which can be opened by right-clicking on a Join and selecting ‘Joins’.

Adding Column Expressions

In addition to specifying aliases, the ‘Columns’ tab of the lower tab pane lets you add expressions to a query via the context menu. Selecting ‘Add Expression’ from the context menu opens a dialog where you can add common SQL expressions such as AVG, COUNT, MAX, MIN and SUM.



AVG (qbt_products.Prod_Cost)

Figure 54:

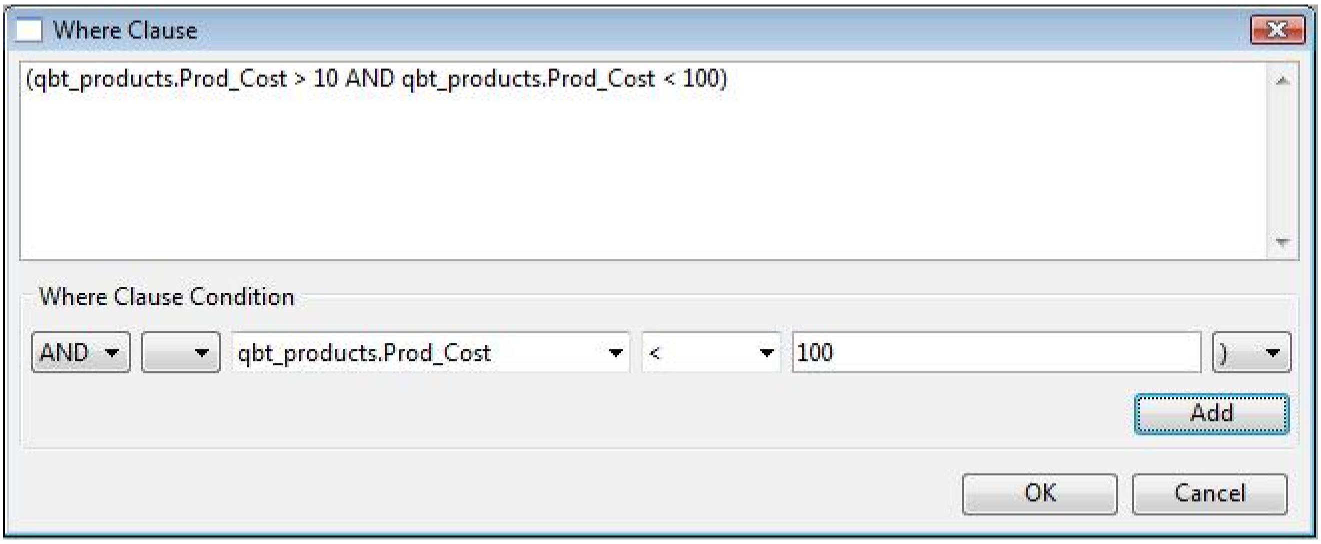
Adding a Where Clause

You can add a Where clause condition by dropping a column from a table onto the ‘Where Clause’ pane of the lower tab pane. When you drop a column the ‘Where Clause’ dialog is opened automatically and the column is pre-selected. You can also right-click on the Where Clause pane to Edit the column conditions for the current query.

Adding a Group by Clause

You can add a Group By clause, to group selected rows together to return a summary of information, by dropping a column name onto the ‘Group By’ pane of the lower tab pane. You can also add a Having Clause to restrict the rows used by the Group By clause either by dropping a column from a table or via the context menu.

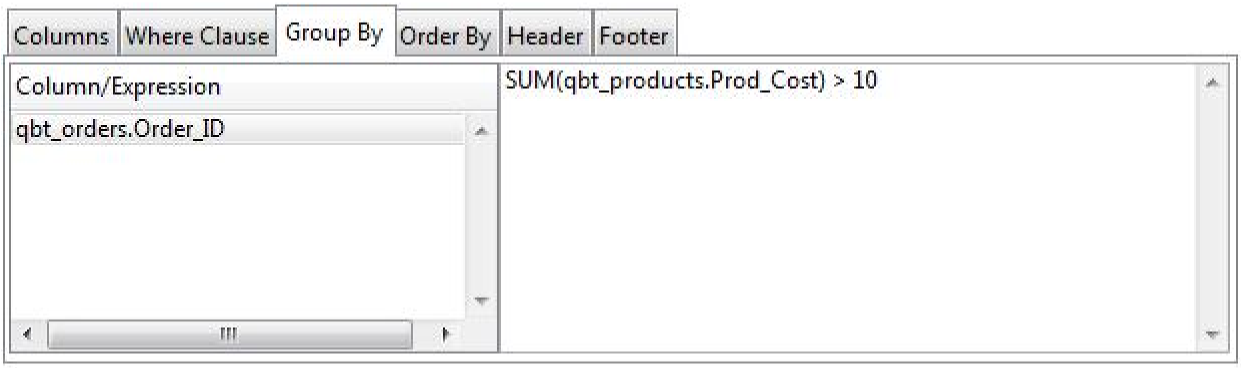
Adding an Order by Clause



```
(qbt_products.Prod_Cost > 10 AND qbt_products.Prod_Cost < 100)
```

AND | | qbt_products.Prod_Cost | < | 100 |)

Figure 55:

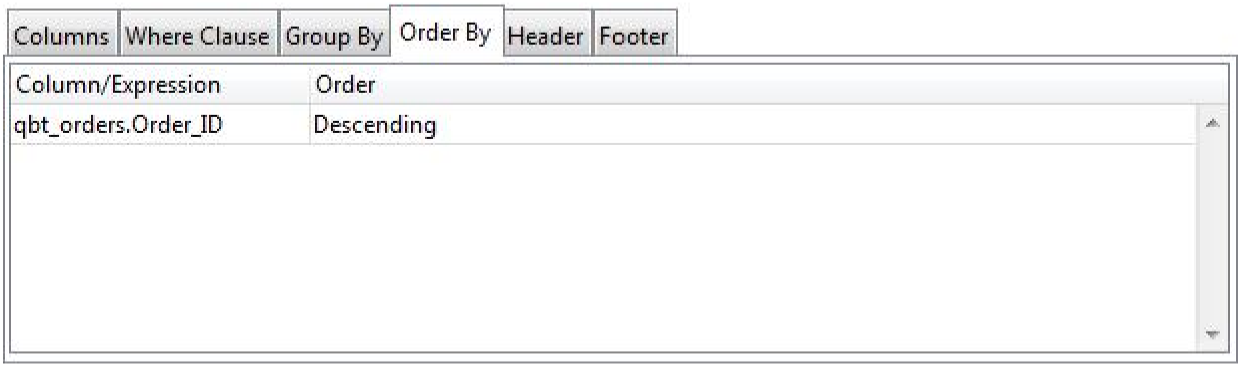


Column/Expression
qbt_orders.Order_ID

```
SUM(qbt_products.Prod_Cost) > 10
```

Figure 56:

You can add an Order By clause by dropping columns onto the ‘Order By’ tab of the lower tab pane: when you drop a column Descending order is selected by default but you can change it to Ascending. You can add Expressions to the Order By clause using the context menu by right-clicking on the Order By pane.

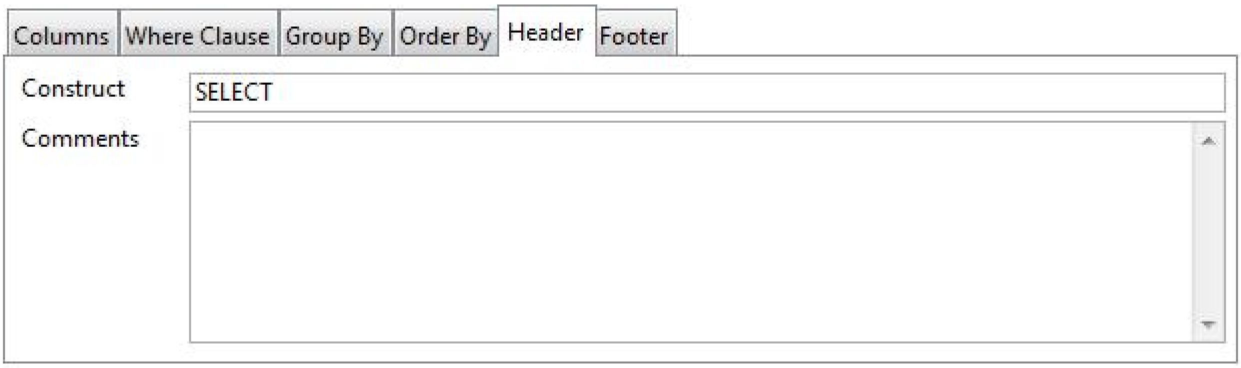


Column/Expression	Order
qbt_orders.Order_ID	Descending

Figure 57:

Modifying the SELECT Construct

The ‘Header Tab’ in the lower tab pane lets you enter an alternative construct, such as ‘SELECT DISTINCT’ and where supported ‘SELECT TOP 100’. This tab also lets you add comments to precede the generated SQL Statement.



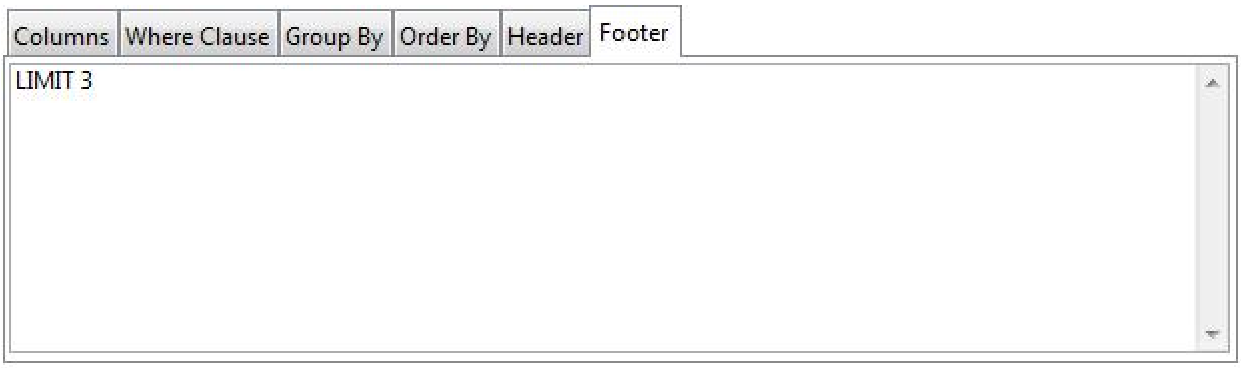
Construct: SELECT

Comments

Figure 58:

Adding Extra Query Text

The ‘Footer Tab’ in the lower tab pane lets you add any additional query text to be appended to the generated SQL Statement.

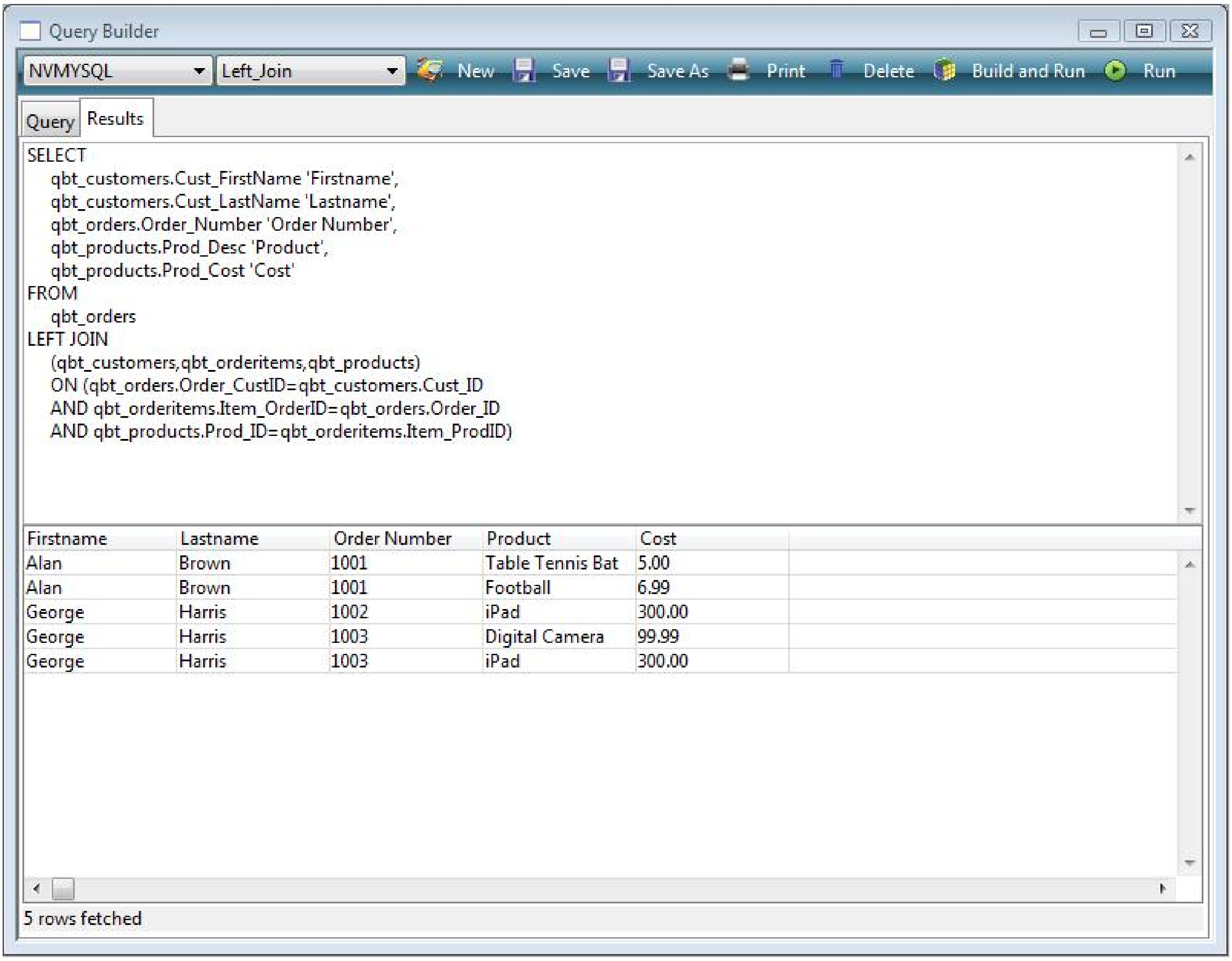


```
LIMIT 3
```

Figure 59:

Running a Query

You can run the current query from the main toolbar in the Query Builder window using the ‘Build and Run’ or ‘Run’ option. Both buttons switch the main pane to the ‘Results’ pane showing the SQL script and results generated by the Query. You can make changes to the SQL script generated if required and the query can be executed again using the ‘Run’ button. Note that using the



```
SELECT
    qbt_customers.Cust_FirstName 'Firstname',
    qbt_customers.Cust_LastName 'Lastname',
    qbt_orders.Order_Number 'Order Number',
    qbt_products.Prod_Desc 'Product',
    qbt_products.Prod_Cost 'Cost'
FROM
    qbt_orders
LEFT JOIN
    (qbt_customers,qbt_orderitems,qbt_products)
    ON (qbt_orders.Order_CustID=qbt_customers.Cust_ID
    AND qbt_orderitems.Item_OrderID=qbt_orders.Order_ID
    AND qbt_products.Prod_ID=qbt_orderitems.Item_ProdID)
```

Firstname	Lastname	Order Number	Product	Cost
Alan	Brown	1001	Table Tennis Bat	5.00
Alan	Brown	1001	Football	6.99
George	Harris	1002	iPad	300.00
George	Harris	1003	Digital Camera	99.99
George	Harris	1003	iPad	300.00

5 rows fetched

Figure 60:

‘Build and Run’ option will rebuild the statement from the saved query text and therefore overwrite any changes you may have made.

Any errors which occur are reported in the status bar. If the full error text is not displayed, you can click on the status bar to open a dialog showing the full error text.

You can resize the font in the Query Builder results pane using the Ctrl+/Ctrl- shortcut keys.

You can change the max line height in the result window using the context menu on the results grid; the setting is saved in the Save Window Setup.

Saving a Query

You can save a query using the ‘Save’ or ‘Save As’ toolbar option. When using ‘Save’ option for the first time, or the ‘Save As’ option, you can add a description for the query. The description is also available to view/edit via the ‘Query Info’ option of the main context menu. Once saved, a Query is added to the list of Queries available in the dropdown list in the main Query Builder toolbar.

Deleting a Query

The Delete button in the Query Builder toolbar button lets you delete the currently selected query (you must confirm the deletion).

Query Reports

There are two types of report available via the Print button on the main toolbar or the context menu opened by right-clicking on the Query design pane: both these options open the Print Query dialog that allows you to print either the Structure or the Results of the current query. In addition, you can include the generated SQL Script from the last executed SQL query in both reports.

Query Structure Report

The ’Query Structure’ report shows the tables and their joins where the selected columns are represented by an astrix ‘*’.

Query Results Report

The ‘Query Results’ report shows the results of the last executed SQL query. Column widths reflect those of the results pane and may therefore be adjusted prior to printing.

Query Info

The ‘Query Info’ dialog available from the main query context menu displays information about the current query. You can change the name and description of query.

Options

You can set various options for the SQL Query Builder in the ‘Options’ dialog which is available from the context menu on the main query window. You can specify the line and color styles for joins, and you can set preferences for dragging and dropping during table creation.

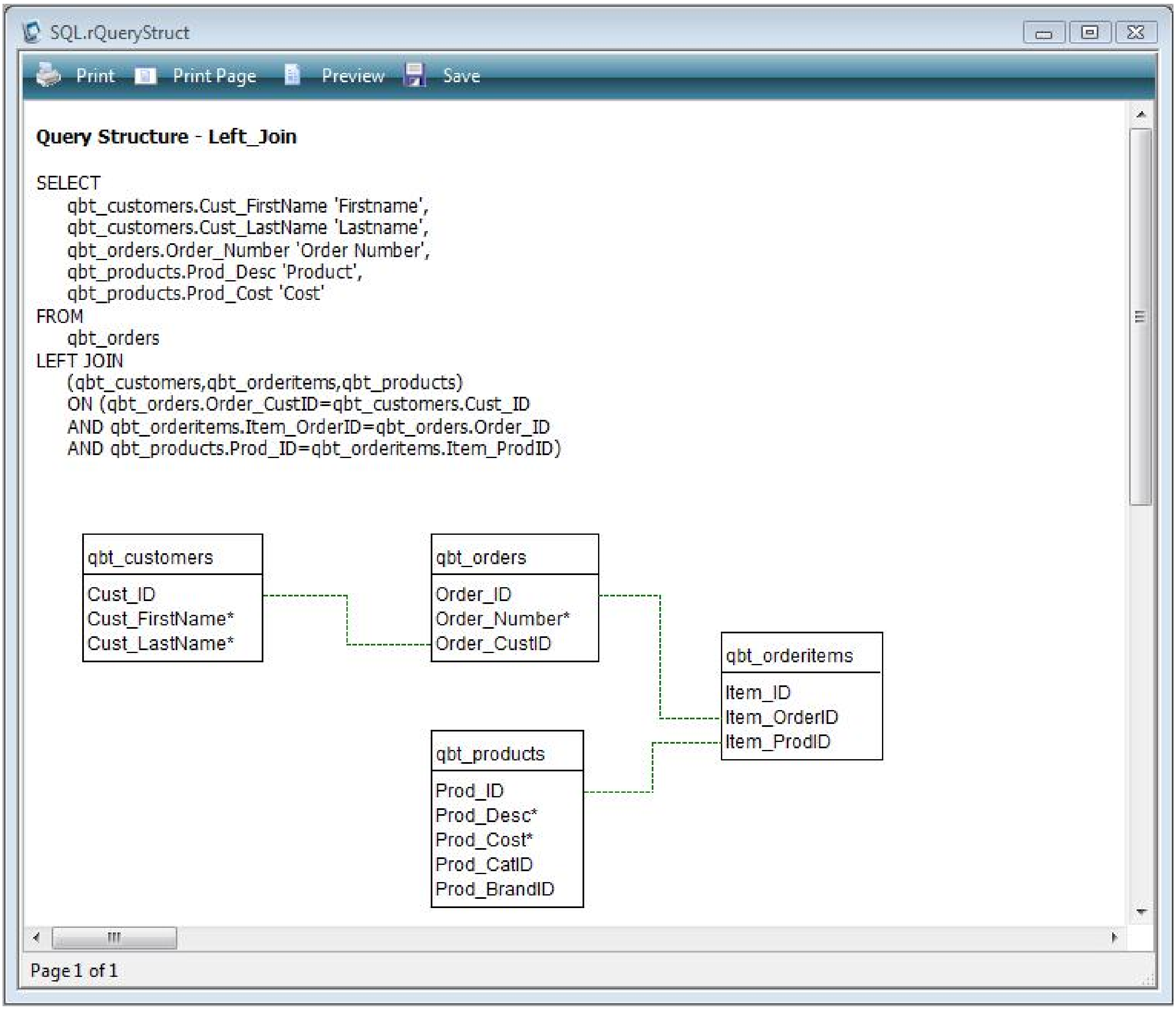
For databases where table names are prefixed by a username, the Omit Username option is particularly useful when running the same query against different databases where the username is different, but the table and column names are the same.

Creating a Query Class

You can create an Omnis Query class based on the current query. To do this, you can drag the current query from the main query design window on the right and drop it on to an open library in the main Studio Browser. When you create a query class in this way, all the additional query text is added to the class, together with any associated Omnis Schema classes in order to map to the server tables in the query. Note that this feature is restricted by the same limitations as a query class and therefore only supports ‘Where Clause’ joins.

Creating a Table class

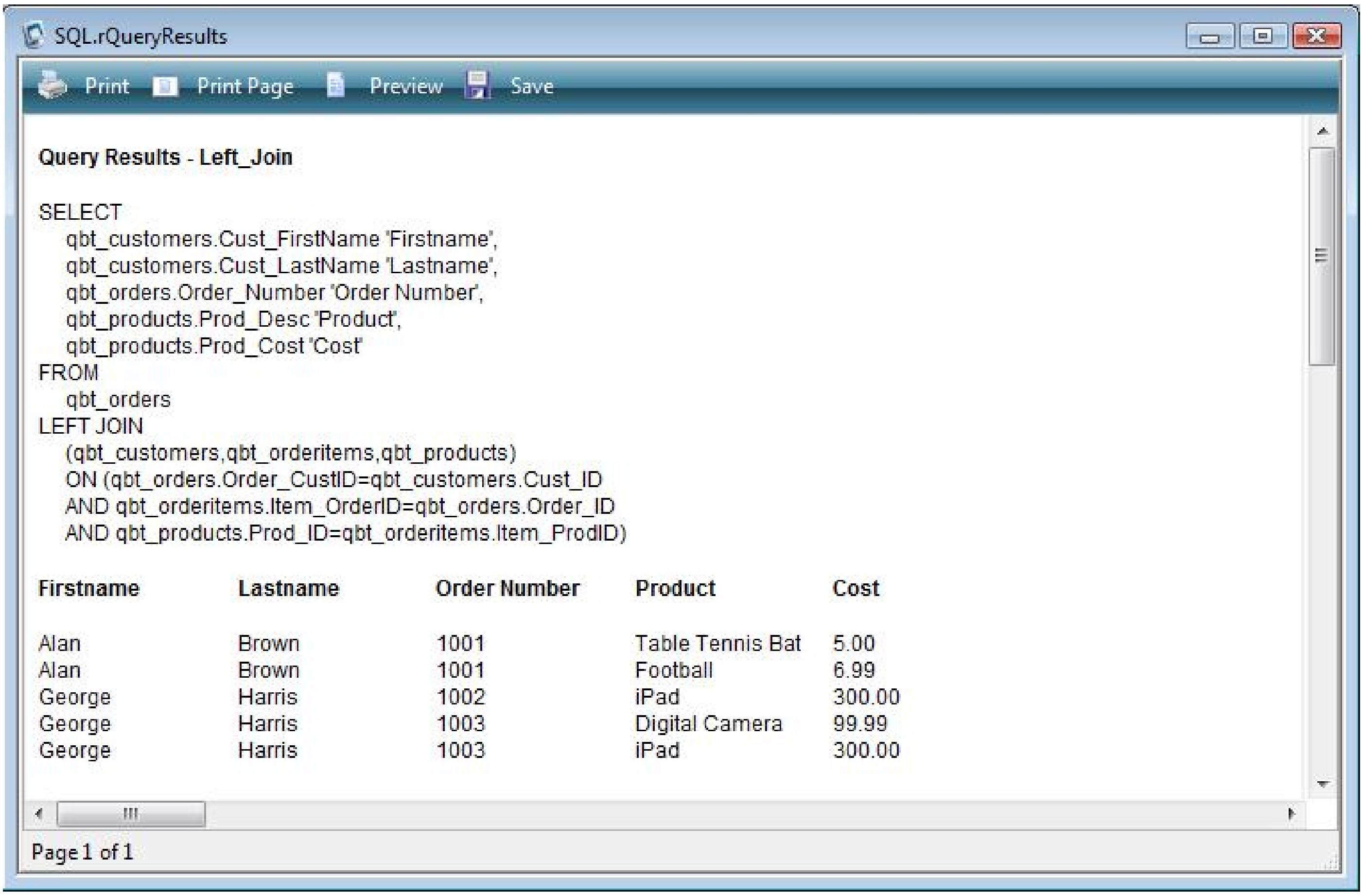
You can create a table class from the current query using the ‘Create table class’ option which is available on the ‘Other’ toolbar menu option: the option also gives you the option to create a window class and/or a remote form for viewing the data via the new table class.



Query Structure - Left_Join

```
SELECT
     qbt_customers.Cust_FirstName 'Firstname',
     qbt_customers.Cust_LastName 'Lastname',
     qbt_orders.Order_Number 'Order Number',
     qbt_products.Prod_Desc 'Product',
     qbt_products.Prod_Cost 'Cost'
FROM
     qbt_orders
LEFT JOIN
     (qbt_customers,qbt_orderitems,qbt_products)
     ON (qbt_orders.Order_CustID=qbt_customers.Cust_ID
     AND qbt_orderitems.Item_OrderID=qbt_orders.Order_ID
     AND qbt_products.Prod_ID=qbt_orderitems.Item_ProdID)
```

Figure 61:



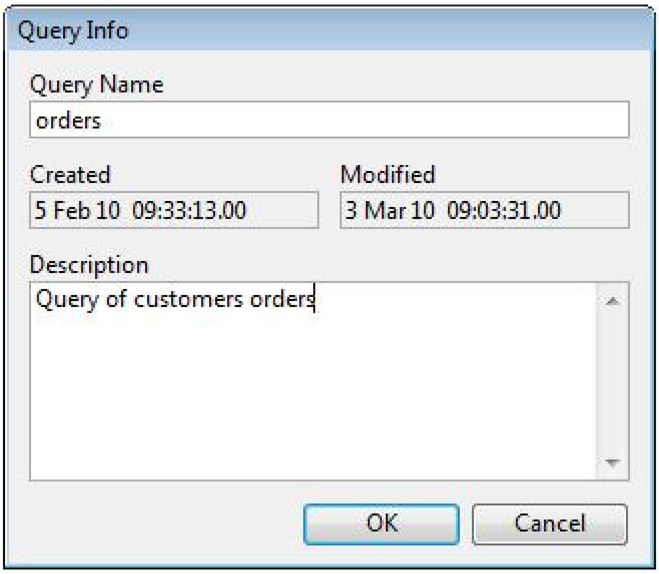
Query Results - Left_Join

```
SELECT
    qbt_customers.Cust_FirstName 'Firstname',
    qbt_customers.Cust_LastName 'Lastname',
    qbt_orders.Order_Number 'Order Number',
    qbt_products.Prod_Desc 'Product',
    qbt_products.Prod_Cost 'Cost'
FROM
    qbt_orders
LEFT JOIN
    (qbt_customers,qbt_orderitems,qbt_products)
    ON (qbt_orders.Order_CustID=qbt_customers.Cust_ID
    AND qbt_orderitems.Item_OrderID=qbt_orders.Order_ID
    AND qbt_products.Prod_ID=qbt_orderitems.Item_ProdID)
```

Firstname	Lastname	Order Number	Product	Cost
Alan	Brown	1001	Table Tennis Bat	5.00
Alan	Brown	1001	Football	6.99
George	Harris	1002	iPad	300.00
George	Harris	1003	Digital Camera	99.99
George	Harris	1003	iPad	300.00

Page 1 of 1

Figure 62:



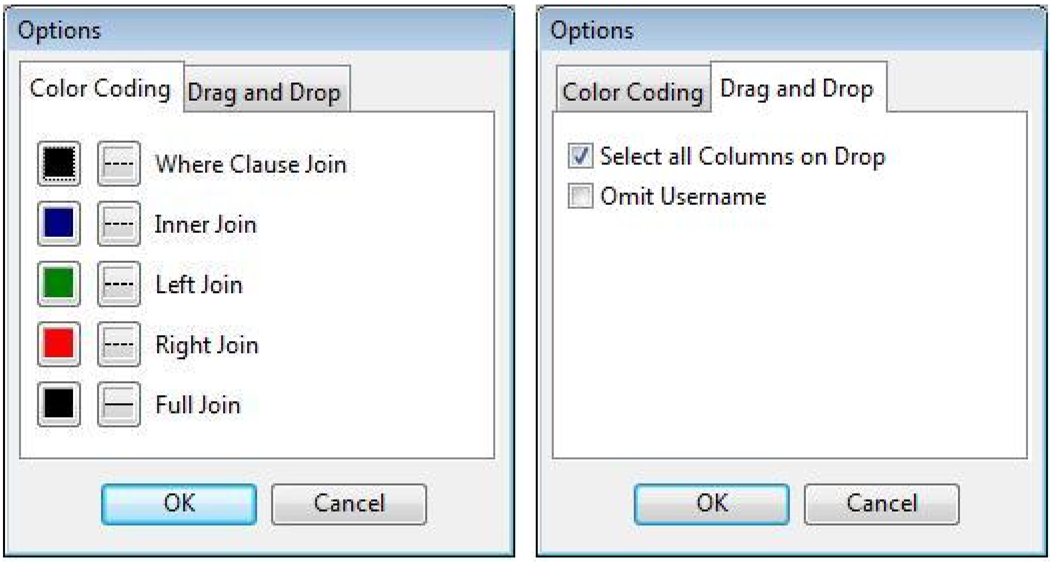
Query Name: orders

Created: 5 Feb 10 09:33:13.00

Modified: 3 Mar 10 09:03:31.00

Description: Query of customers orders

Figure 63:



Where Clause Join

Inner Join

Left Join

Right Join

Full Join

☑ Select all Columns on Drop

□ Omit Username

Figure 64:

Creating a DB View

You can create a DB View from the 'Other' menu.

Exporting Data

You can export the results data using the 'Export Data' option available on the 'Other' toolbar menu option.

Creating a Statement Block

You can copy the generated SQL Script to the clipboard in a 'Begin Statement', 'Sta:', 'End Statement' block by right-clicking on the 'Results' pane and selecting 'Create Statement' from the context menu. This option is also available on the Other menu option.

You can paste the statement into an Omnis method, providing an alternative method of executing your query in an Omnis Studio library where a query class is not appropriate.

SQL Query Builder App

A runtime or end user version of the SQL Query Builder application is available on request. Please contact your local Omnis sales or support office for details.

Version Control System

Note that some editions of Omnis Studio do not allow access to the Omnis VCS.

The Omnis *Version Control System* (VCS) lets you manage and revise Omnis libraries and other application components systematically. In a team environment, with several people working on the same application at the same time, you need to ensure that only one person can change a particular component at a time.

Using the Omnis VCS you can control the development of your Omnis applications, or any other project involving many different files such as Internet or Intranet applications. Specifically, the VCS can manage Omnis libraries or their classes, external components, DLLs or Code Fragments, Omnis data files, text or word processor files, Html and web server files, or any other types of file required in your Omnis application.

The Omnis VCS has an easy-to-use environment that allows you to check-in and check-out components, plus it had a useful tool that lets you compare different versions of the same library or different revisions of the same component.

The Omnis VCS is described in detail in a later section.

Auto Updates

You can perform updates or any other changes to your Omnis desktop application or folder structure upon restarting Omnis by adding a script to the Omnis data folder. You can use the Auto Update feature to update any file in the Omnis Studio tree, including the Omnis executable or program file itself: however, the studiorg.exe file cannot be updated under Windows.

To enable the Auto Update feature, write a batch file under Windows called **update.bat,** or on macOS or Linux create a bash script called **update.sh,** and add it to the Omnis data folder, i.e. the folder containing the Studio, Startup, and Welcome folders.

When Omnis starts up it will execute the update script automcatically at startup, before loading any external components, externals or libraries. If the call to run the script is successful, Omnis then deletes the update.bat/sh file.

When running on Windows, Omnis incorporates a request to run this as part of the existing UAC support implemented via studiorg.exe. In this case, you will get a UAC prompt if the update script needs to run, or if the registry needs updating for some reason, or if both updates and registry updates are required.

The Windows batch file or Unix script must have Execute permissions set in order to run. You can do this in the Properties of the file or via the file system. To do this in your code on macOS or Linux you can use the $setunixpermissions() fileops function:

```
If sys(6)= 'U' ## macOS or Linux
     Do fileops.$setunixpermissions( scriptPathName,'-rwxr--r--' ) ## set file to execute
End if
```

Update Feedback

When running the auto update script some feedback that the script is running can be provided in a console window. To enable this, you must place a file (which can be empty) named 'showconsole.txt' in the same directory as update.bat. When this file is present, a console window is displayed while update.bat is running.

Example

The following example shows typical commands that could be used in a batch script: the commands download two new xcomps from a server (xcomp1.dll and xcomp2.dll), and store them in a folder specified by con(sys(115),pathsep(),updates,pathsep(),xcomp):

```
copy /y <studio data folder path>\updates\xcomp\xcomp1.dll
<studio program folder path>\xcomp\xcomp1.dll
del <studio data folder path>\updates\xcomp\xcomp1.dll
copy /y <studio data folder path>\updates\xcomp\xcomp2.dll
<studio program folder path>\xcomp\xcomp2.dll
del <studio data folder path>\updates\xcomp\xcomp2.dll
```

When a path has a space or spaces in it, e.g. Program files, the path should be enclosed in quotes:

```
"C:\Program Files\Omnis Software\OS11\xcomp\Dummy.dll"
```

External Class Editor

$editor and $editordata are properties of all class types except system tables – in previous versions these properties were only available for object classes. The properties allow you to specify your own editor and to access the data for an Omnis class. The definitions of these properties are:

- **$editor**
  The name of the add-in tool library used to edit the class
- **$editordata**
  Editor data stored with the class. Typically used by the library identified by $editor

By default, these properties are not visible in the Property Manager, therefore to make them visible, edit the show_editor item under "properties" in the config.json file:

```
  "properties": {
    "show_editor": true
  },
```

In addition, you should note that $editordata will only appear in the Property Manager when used in conjunction with the Notation Inspector.

$editor overrides the default editor for a class. Omnis calls $exectool for the specified add-in library, passing it a single parameter which is an item reference to the class.

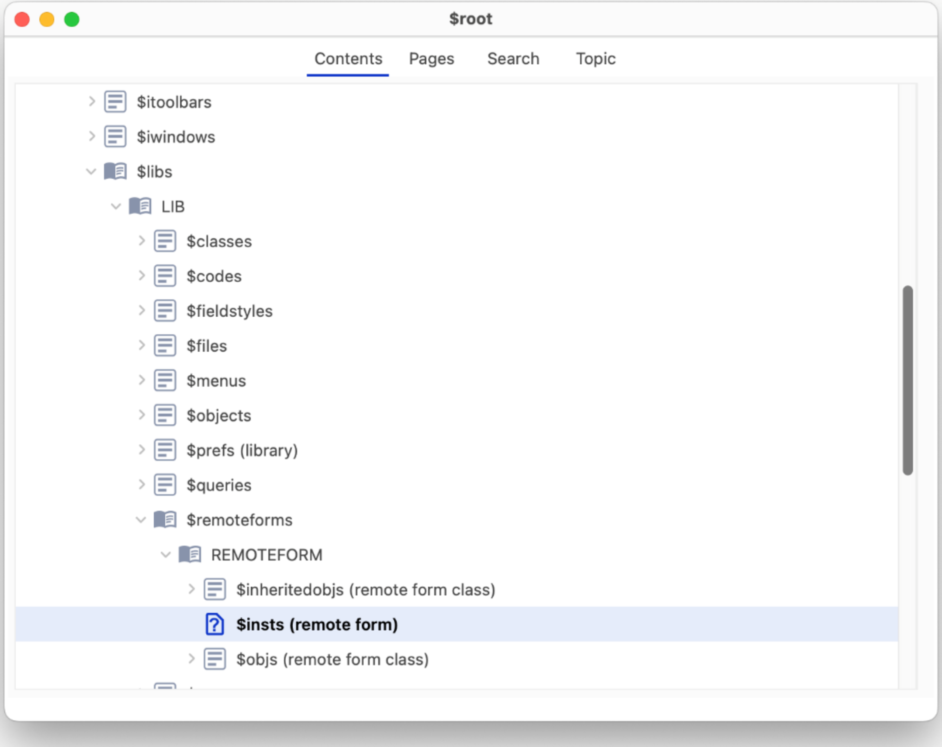
Note that the specified editor is not used when using find and replace – instead, the normal editor for the class opens.

There is a Tech Note TNID0007 on the Omnis website that shows how you can create an alternative schema editor.

Omnis Help

In design mode, Omnis provides many different types of help: tooltips and helptips over toolbar controls: help text for the main menus shown in the status bar at the bottom of the Omnis application window: plus when you're writing a method the Code Assistant will popup automatically to display the appropriate variable name, notation (property or method name), and the correct parameters for the current object or context in your code.

In addition to these types of help, there is a fully context-sensitive **Help system,** available by pressing F1 or via the Help menu. Omnis also provides 'What's This?' help for individual functions and commands while you're working in the Method Editor: 'What's This' type help is *not* provided for the main tools in the Omnis IDE.



- $itoolbars
- $iwindows
- $libs
  - LIB
    - $classes
    - $codes
    - $fieldstyles
    - $files
    - $menus
    - $objects
    - $prefs (library)
    - $queries
    - $remoteforms
      - REMOTEFORM
        - $inheritedobjs (remote form class)
        - **$insts (remote form)**
        - $objs (remote form class)

Figure 65:

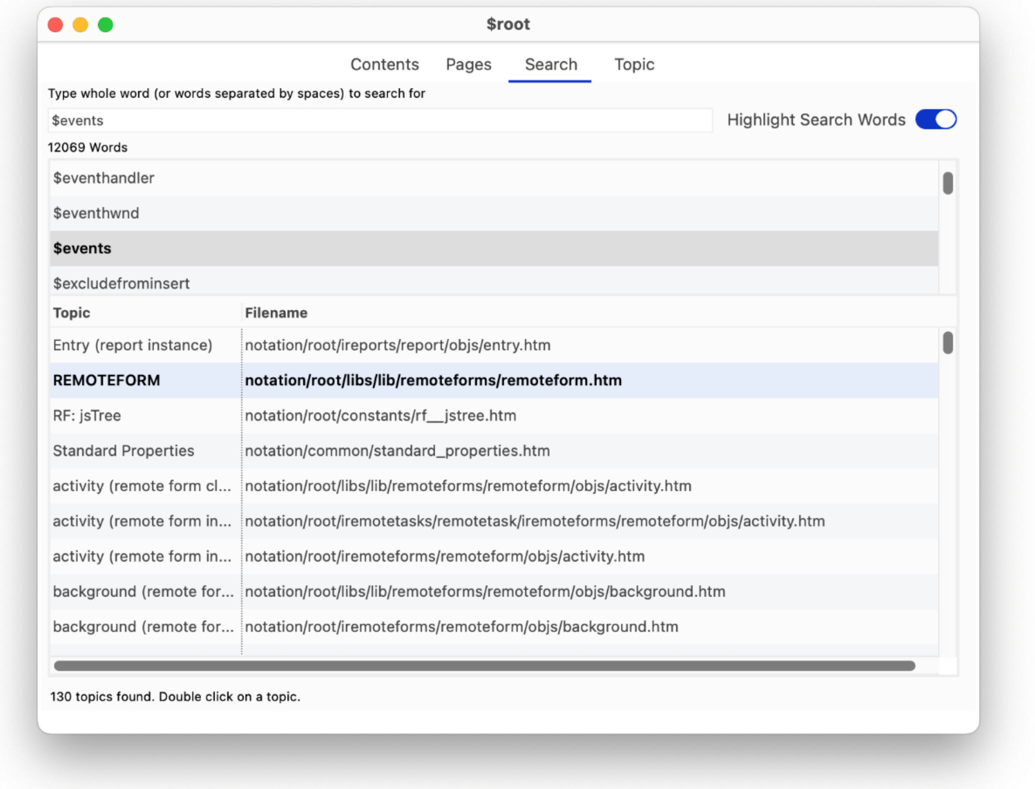
Under the Contents tab you can drill down to the topic or object type you're looking for: you can double-click on an item in the tree to load the page.

Under the Search tab you can type a command, function, or method name and Omnis will provide a list of matching topics: you can double-click an item in the Topic list to load the page under the Topic tab (as shown below).

You can create your own Help system and add it to your own application using the Help Project Manager under the Tools menu. Note that the Omnis Help available in the development version of Omnis is located in the omnis\idehelp folder.

System Notifications

Omnis can send notifications to the operating system on the end user's computer, on both Windows 10/11 and macOS. You have control over the content of notifications and when they are sent via your Omnis code using a new **ONOTIFY** object. When sent, a notification will pop up on the end user's screen and will be added to the **Notification Center** for the current operating system.



Type whole word (or words separated by spaces) to search for

$events

12069 Words

$eventhandler

$eventhwnd

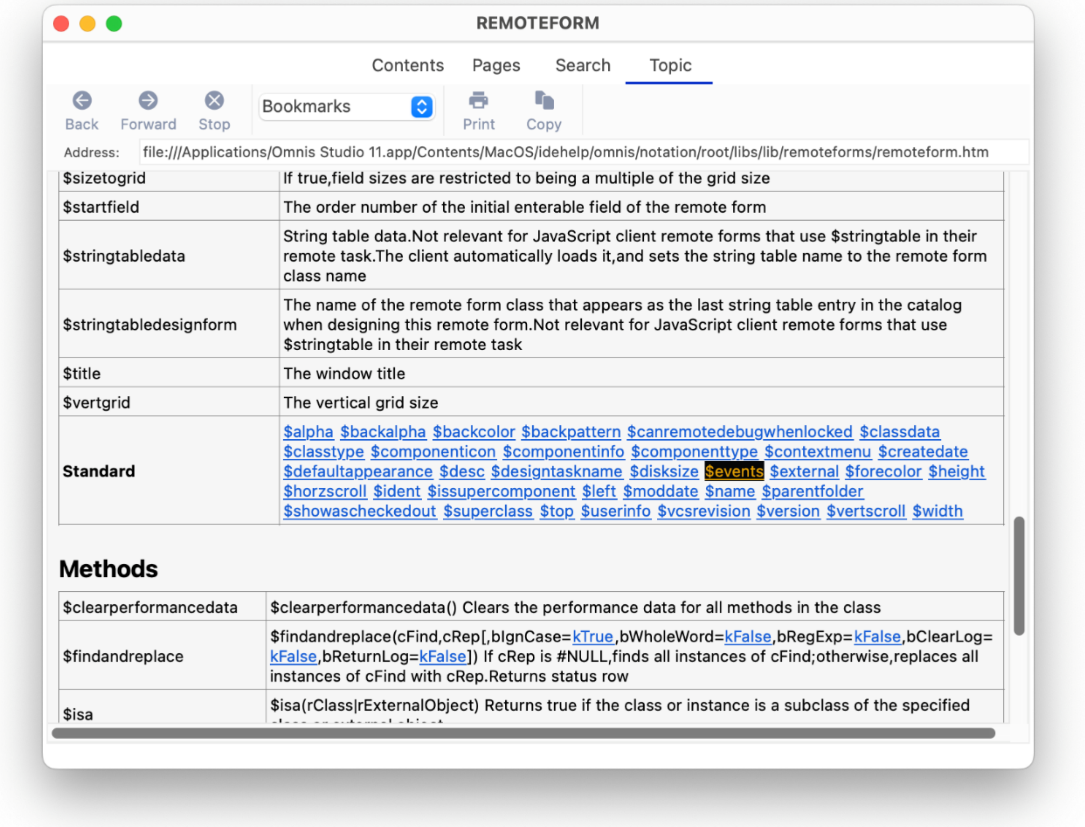
$events

$excludefrominsert

Topic	Filename
Entry (report instance)	notation/root/ireports/report/objs/entry.htm
REMOTEFORM	**notation/root/libs/lib/remoteforms/remoteform.htm**
RF: jsTree	notation/root/constants/rf__jstree.htm
Standard Properties	notation/common/standard_properties.htm
activity (remote form cl...	notation/root/libs/lib/remoteforms/remoteform/objs/activity.htm
activity (remote form in...	notation/root/iremotetasks/remotetask/iremoteforms/remoteform/objs/activity.htm
activity (remote form in...	notation/root/iremoteforms/remoteform/objs/activity.htm
background (remote for...	notation/root/libs/lib/remoteforms/remoteform/objs/background.htm
background (remote for...	notation/root/iremoteforms/remoteform/objs/background.htm

130 topics found. Double click on a topic.

Figure 66:



Address: file:///Applications/Omnis Studio 11.app/Contents/MacOS/idehelp/omnis/notation/root/libs/lib/remoteforms/remoteform.htm

$sizetogrid	If true,field sizes are restricted to being a multiple of the grid size
$startfield	The order number of the initial enterable field of the remote form
$stringtabledata	String table data.Not relevant for JavaScript client remote forms that use $stringtable in their remote task.The client automatically loads it,and sets the string table name to the remote form class name
$stringtabledesignform	The name of the remote form class that appears as the last string table entry in the catalog when designing this remote form.Not relevant for JavaScript client remote forms that use $stringtable in their remote task
$title	The window title
$vertgrid	The vertical grid size
Standard	$alpha $backalpha $backcolor $backpattern $canremotedebugwhenlocked $classdata $classtype $componenticon $componentinfo $componenttype $contextmenu $createdate $defaultappearance $desc $designtaskname $disksize $events $external $forecolor $height $horzscroll $ident $issupercomponent $left $moddate $name $parentfolder $showascheckedout $superclass $top $userinfo $vcsrevision $version $vertscroll $width

Methods

$clearperformancedata	$clearperformancedata() Clears the performance data for all methods in the class
$findandreplace	$findandreplace(cFind,cRep[,bIgnCase=kTrue,bWholeWord=kFalse,bRegExp=kFalse,bClearLog=kFalse,bReturnLog=kFalse]) If cRep is #NULL,finds all instances of cFind;otherwise,replaces all instances of cFind with cRep.Returns status row
$isa	$isa(rClass|rExternalObject) Returns true if the class or instance is a subclass of the specified

Figure 67:

The end user can click on a notification and either start Omnis, or if Omnis is already running, bring Omnis to the front. In both of these cases, the method **$localnotify()** in the **Startup_Task** (in the library that sent the notification) receives parameters specific to the notification and the method can then process the click, or call another method, for example.

As well as sending notifications, there are additional functions that allow you to add a badge to the application icon to alert the end user about the notifications.

There are two interfaces provided to send a system notification:

- **An object,** providing a way to encapsulate notification parameters.
- **A function,** providing a simple interface to send a notification with a single line of Omnis code.

As with many features in Omnis, these interfaces provide a single, cross-platform method to interact with system notifications on both Windows and macOS.

Notification Object

The **Notification Object** provides a way to encapsulate notification parameters. To use the object, you need to set the *Subtype* of an *Object variable* to the **LocalNotify** external object. The object has the following properties:

Property	Description
$action	A value that specifies up to 2 optional actions that are to be included in the notification; on Windows, this is via one or two buttons; on macOS, this is either via a button for a single action, or via an options popup for two actions. A 'Specifying Actions' section
$delay	The delay in seconds between the call to $sendlocal() and the notification being delivered. Omnis can quit before the notification is delivered, as the operating system takes care of deferred delivery
$messageimage	Image(s) to be displayed with the notification. See the 'Specifying Images' section
$messagetext	The text of the notification. This is the main notification message, displayed in a plain font. The operating system will truncate this if it occupies more than 4 lines, either due to word wrapping, or the presence of newline characters (kCr, lLf or kCr kLf)

Property	Description
$notifylib	See section ‘Handling Notification Clicks’ for details about this property
$title	The title of the notification. Some text, displayed in bold font above the main notification text. The operating system will truncate this if it is too long. Windows allows this to occupy two lines, if you separate the lines using either kCr, lLf or kCr kLf. macOS only allows a single line
$userinfo	A row containing user information that is passed to the $localnotify() method when the user clicks on the notification or a notification action. It must be possible to convert $userinfo to JSON. See section ‘Handling Notification Clicks’

To send a notification, created using the current property values, use the $send() method of the object.

```
Do Object.$send([&cErrorText])
```

Sends a local operating system notification using the current property values. The parameters are as follows:

Parameter	Description
cErrorText	A character variable that receives text describing an error if $send() fails

If the call to $send() fails, it returns the value #NULL, and sets the cErrorText parameter if it is provided.

If the call to $send() succeeds, it returns a character string. This is a string that uniquely identifies the notification. You can use this string to remove the notification from the system Notification Center, if for example the notification is no longer relevant.

Notification Functions

The **ONOTIFY.$sendlocal()** function sends a system notification.

```
Do ONOTIFY.$sendlocal(cTitle,cMessage,vImage,iAction,wUserInfo,[iDelay=0,&cErrorText])
```

The parameters are as follows:

Parameter	Description
cTitle	The title of the notification. Some text, displayed in bold font above the main notification text. The operating system will truncate this if it is too long. Windows allows this to occupy two lines, if you separate the lines using either kCr, lLf or kCr kLf. macOS only allows a single line
cMessage	The text of the notification. This is the main notification message, displayed in a plain font. The operating system will truncate this if it occupies more than 4 lines, either due to word wrapping, or the presence of newline characters (kCr, lLf or kCr kLf)
vImage	Image(s) to be displayed with the notification. See the 'Specifying Images' section
iAction	A value that specifies up to 2 optional actions that are to be included in the notification; on Windows, this is via one or two buttons; on macOS, this is either via a button for a single action, or via an options popup for two actions. A 'Specifying Actions' section
wUserInfo	A row containing user information that is passed to the $localnotify() method when the user clicks on the notification or a notification action. It must be possible to convert $userinfo to JSON. See section 'Handling Notification Clicks'
iDelay	The delay in seconds between the call to $sendlocal() and the notification being delivered (optional). Omnis can quit before the notification is delivered, as the operating system takes care of deferred delivery
cErrorText	A character variable that receives text describing an error if $sendlocal() fails

If the call to $sendlocal() fails, it returns #NULL, and sets the cErrorText parameter if it is provided.

If the call to $sendlocal() succeeds, it returns a character string. This is a string that uniquely identifies the notification. You can use this string to remove the notification from the system Notification Center, if for example the notification is no longer relevant.

Specifying Images

You can specify an image for the notification via the $messageimage property of the object, or vImage parameter of the function. macOS only allows a single image, whereas Windows allows up to three. The Windows images must each have an associated type, and there can only be one image of each type. The image types are identified by constants:

Constant	Description
kONOTIFYimageTypeNormal	The image is to be displayed below the notification.
kONOTIFYimageTypeLogo	The image is to be used as the application logo.
kONOTIFYimageTypeHero	The image is to be used as the hero image (this is Windows terminology). This is an image displayed across the top of the notification, and it must have the size 364x180 (728x360 retina) to look good, otherwise the system resizes it and crops it.

Images can be specified either using a character variable, or by using a list. To include no image in the notification, either use an empty character variable or value, or use a list with no lines and the correct number of columns (see below).

If you use a character variable, with a non-empty value, the notification has a single image; the character variable must contain the full pathname of an image file (typically PNG or JPEG), and on Windows it will have the type kONOTIFYimageTypeLogo.

If you use a list variable, then the list must have at least one column on macOS, and at least 2 columns on Windows. The number of rows is limited to 1 on macOS, and 3 on Windows (one for each type). Column 1 of the list contains the full pathname of an image file (typically PNG or JPEG), and column 2 contains a kONOTIFYimageType… constant.

The system is responsible for laying out the notification content (i.e. you have no control over layout), and you should avoid using very large images in a notification.

Specifying Actions

You can specify up to 2 actions to be included with the notification. To specify no actions, the action value can be either zero or kONOTIFYactionNone.

The actions are pre-defined, as macOS requires actions to be pre-defined. To specify one or more actions, use the following constants, which can be added together when specifying 2 actions:

Constant	Description
kONOTIFYactionAccept	The notification displays the Accept action.
kONOTIFYactionClose	The notification displays the Close action.
kONOTIFYactionDecline	The notification displays the Decline action.
kONOTIFYactionDelete	The notification displays the Delete action.
kONOTIFYactionNo	The notification displays the No action.
kONOTIFYactionOpen	The notification displays the Open action.
kONOTIFYactionPrint	The notification displays the Print action.
kONOTIFYactionYes	The notification displays the Yes action.

Handling Notification Clicks

By default, when the user clicks on either a notification, or a notification action, Omnis executes the method **$localnotify()** in the **Startup_Task** of the library containing the code calling ONOTIFY.$sendlocal() or object.$send().

When using a LocalNotify object to send the notification, you can override the library using the **$notifylib** property; this property is the internal name of the library whose startup task is to receive the $localnotify() call. If you do not assign $notifylib, or set it to empty, the default behavior applies.

If Omnis is not running when the user clicks on either a notification or a notification action, the system starts Omnis. Omnis defers calling $localnotify() until the startup task has completed, to allow startup libraries to be opened and their initialization to complete.

If Omnis is running when the user clicks on either a notification or a notification action, the system brings Omnis to the front.

When the system calls Omnis to tell it about a notification, and the library in which $localnotify() is to be called is not open (after waiting for startup to complete if necessary), Omnis ignores the call.

$localnotify appears in the built-in methods of a task class, so you can override it. It has 2 parameters:

Parameter	Description
pAction	A kONOTIFYaction… constant that identifies the action pressed by the user. kONOTIFYactionNone (zero) if the user clicks directly on the notification, rather than a button or popup.
pUserInfo	A row. The user info value that was supplied when sending the notification.

$localnotify() is not required to return a value.

Removing Notifications

The notification object and function send methods (Object.$send() and ONOTIFY.$sendlocal()) both return a **unique id** to identify the notification that was sent. If you want to remove the notification from the Notification Center at some point later (possibly after restarting Omnis), you need to save the id somewhere, e.g. in a local SQLite database.

To remove one or more (or even all) notifications sent by Omnis, use the method:

```
Do ONOTIFY.$removelocal([vIDs,&cErrorText])
```

The parameters are as follows:

Parameter	Description
vIDs	Either a single character id, or a single column list of ids, to remove. To remove all local notifications sent by Omnis, pass an empty character string, a list with no lines, or omit the vIDs parameter.
cErrorText	A character variable that receives text describing an error, if $removelocal() fails.

If the call to $removelocal() fails, it returns the Boolean value false, and sets the cErrorText parameter if it is provided. If the call to $removelocal() succeeds, it returns the Boolean value true.

Badges

ONOTIFY provides functions that allow a badge to be added to, or removed from, the application icon.

On Windows, this applies to the application icon in the taskbar. On macOS, this applies to the application icon in both the dock, and the task switcher. The two operating systems behave differently, because of the way their APIs work.

The badge is only displayed while Omnis is running.

$setbadgecount

ONOTIFY.$setbadgecount(iCount[,&cErrorText,iBadgeColor,iBadgeTextColor])

Sets the application icon badge to the specified count. The parameters are as follows:

Parameter	Description
iCount	The count to display as the badge. Must be greater than zero. When running on Windows, a value greater than 99 is displayed as 99+.
cErrorText	A character variable that receives text describing an error, if $setbadgecount() fails
iBadgeColor	Windows only. The background color of the count badge. Defaults to styledbadgebackgroundcolor in the system section of appearance.json.
iBadgeTextColor	Windows only. The text color of the count badge. Defaults to styledbadgetextcolor in the system section of appearance.json.

Note that the appearance.json items styledbadgebackgroundcolor and styledbadgetextcolor have been moved to the ‘system’ section of appearance.json.

If the call to $setbadgecount() fails, it returns the Boolean value false, and sets the cErrorText parameter if it is provided. If the call to $setbadgecount() succeeds, it returns the Boolean value true.

$setbadgeicon

ONOTIFY.$setbadgeicon(vIconId[,&cErrorText,iBadgeColor=kColorHilight])

Sets the badge on the application icon to be the specified icon. Note this is available on Windows only. The parameters are as follows:

Parameter	Description
vIconId	The icon id of the icon to display as the badge. The size component is ignored, as badges are always 16x16.
cErrorText	A character variable that receives text describing an error, if $setbadgeicon() fails
iBadgeColor	The color to be applied to the themed SVG; only applies if the icon is a themed SVG. Default kColorHilight.

If the call to $setbadgeicon() fails, it returns the Boolean value false, and sets the cErrorText parameter if it is provided. If the call to $setbadgeicon() succeeds, it returns the Boolean value true.

$removebadge

ONOTIFY.$removebadge([&cErrorText])

Removes the badge from the application icon. The parameters are as follows:

Parameter	Description
cErrorText	A character variable that receives text describing an error, if $setbadgecount() fails.

If the call to $removebadge() fails, it returns the Boolean value false, and sets the cErrorText parameter if it is provided. If the call to $removebadge() succeeds, it returns the Boolean value true.

Enabling Notifications

To receive notifications from Omnis, notifications have to be enabled for Omnis in the respective system settings. On **Windows,** you can enable System Notifications via the Settings >> System dialog, then the **Notifications & Actions** option. On **macOS,** you can use the System Preferences >> **Notifications & Focus** option. (See the Enabling Notifications section for platform considerations.)

The following describes how Omnis is identified by each operating system in order to initialize system notifications.

macOS

The macOS operating system identifies applications using their application bundle identifier, so if you install multiple versions of Omnis on the same macOS system, notification settings, such as those in the **Notifications & Focus** section of System Preferences, apply to all applications with that bundle identifier.

For Studio 11.0, the application bundle identifier now includes the version, that is, net.omnis.omnisStudio.11.0. In addition, the Development, Server, or Runtime versions of Omnis are identified by type. Therefore, the application bundle identifier is net.omnis.omnisStudio.<type>.11.0 where <type> can be Dev, Server, or Run, so these three executables can co-exist on the same macOS system. In addition, the deployment tool caters to the different types.

Windows

For notifications to work on Windows, and in particular to allow clicks on notifications to be passed to Omnis, Omnis needs to register an AppUserModelID and store the AppUserModelID in a shortcut to Omnis in the system Start menu.

There are two configuration entries in the 'windows' section of config.json:

Entry	Description
initLocalNotifications	Boolean. Default true. If true, Omnis initialises the interfaces required to send notifications to the local Notification Center.
createShortcut	Boolean. Default true. If true, and there is no shortcut to itself in the Start menu, Omnis creates a shortcut to itself in the Start menu. It then modifies the shortcut to contain the AppUserModelID required for local notifications to work.

Omnis uses core resource string 9 as the template for its AppUserModelID. This defaults to "OmnisSoftware.OmnisStudio.$.11". To create the AppUserModelID, Omnis replaces $ with Dev, Server or Run to identify the Development, Server or Runtime version of Omnis.

The deployment tool (Windows only) allows you to customize resource 9. Note that if there is no $ placeholder in the resource, the resource value is not changed by the attempt to insert Dev, Server or Run.

Power Management Notifications

Omnis Studio can receive sleep and wake notifications from the operating system to indicate power management changes: the following applies to macOS and Windows.

Requests from the system to go into idle sleep, when there is no user activity, can be denied on macOS or disabled on both macOS and Windows.

This allows the system to remain awake if Omnis Studio is busy.

Power Management Methods

Each task has a set of power management methods which can be overridden.

$systemcansleep (only sent on macOS)

All library task instances receive a call to the **$systemcansleep** method when the system is requesting permission to go into idle sleep.

If all instances of this method return kTrue then sleep will be allowed to continue and there will be a subsequent call to **$systemwillsleep**.

If any instance returns kFalse from this method then sleep will be aborted.

The total time taken to return from all calls to this method must not exceed 30 seconds or the sleep will continue.

$systemwillsleep

All library task instances receive a call to the **$systemwillsleep** method when the system is starting a sleep which cannot be cancelled, e.g. low battery or laptop lid is closed. This is delivered before any hardware is powered off.

The total time taken to return from all calls to this method must not exceed 30 seconds on macOS or 2 seconds on Windows otherwise the sleep will continue.

This call can be used by an application to save the state before the system sleeps.

Operations can be performed such as saving data to disk or disconnecting from databases.

$systemwillwake

All library task instances receive a call to the **$systemwillwake** method when the system is beginning to power on, i.e. most hardware has not been powered on. Attempts to access disk, network, the display, etc. may result in errors or blocking the process until those resources become available.

On Windows once user interaction is detected, e.g. mouse or keyboard input, then the system will send **$systemdidwake**.

$systemdidwake

All library task instances receive a call to the **$systemdidwake** method when wakeup has completed and the system is powered on. This call can be used by an application to resume the state which was saved when the system went to sleep. Operations can be performed such as loading data from disk or reconnecting to databases.

Disabling idle sleep

Typically the system will be setup to sleep after a set period of inactivity. An Omnis application can disable this by using the **$disablesystemidlesleep** root preference. If set to kTrue the system will be prevented from going into idle sleep.

An application will still receive a call to the method **$systemwillsleep** if the system is starting a sleep which cannot be cancelled.

On macOS the system will log a message to the system log to indicate the reason why the system is blocked from going into idle sleep.

A Studio application can set this log message by using the $disablesystemidlesleepreason root preference.

The default for this message is set to ‘Omnis Studio is busy’ but can be altered by editing the string for resource number 1835.

The message should describe the name of the application and the activity blocking the sleep, e.g. “MyApp is searching appointments“.

Chapter 2—Libraries and Classes

The components for your Omnis application or project are stored in an Omnis ***library file.*** This file contains all the class definitions that define the data and UI objects in your application. The class types available in Omnis include remote forms (for web or mobile apps), remote tasks for handling web communications, schemas for defining your data structures, reports for presenting your data, and so on.

Omnis Classes are predefined structures that control the appearance and behavior of the objects in your application, and therefore classes are the main components or building blocks in your application. You can create classes using the wizards provided in the Studio Browser, or you can create them from scratch using class templates. You can create any number of classes in each library file and modify them at any time while you develop your application.

You can manage the classes in your application or project in the **Studio Browser,** or for a multi-library project, involving multiple developers, you may like to use the Omnis VCS and work on different parts of your library on a collaborative basis (depending on the developer license you have).

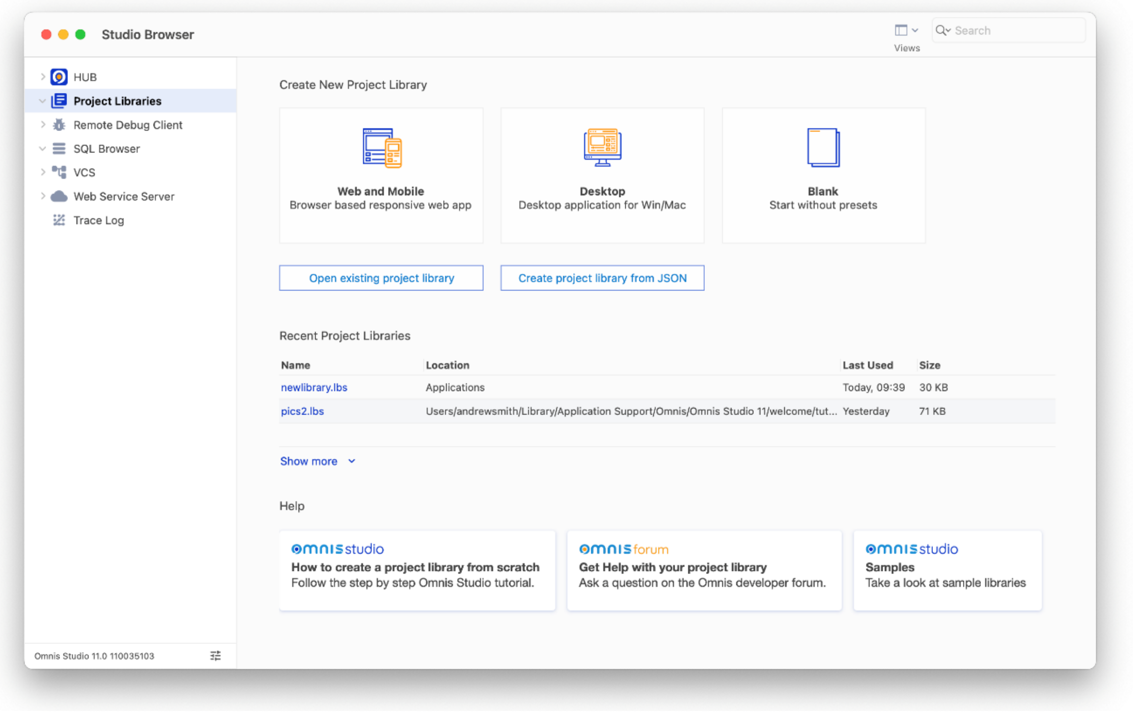
Omnis Libraries

Each library file contains a number of system classes and preferences that control the behavior of your library and its contents. A library has certain properties (preferences) too, which you can examine and change using the Property Manager.

You can create and open any number of library files in the **Studio Browser,** and each library can contain any number of classes. In practice, you may want to split your whole application into one or more libraries and store different types of objects in different libraries.

Creating a New Library

- Start Omnis and open the **Studio Browser:** the Studio Browser should open by default, but if it is not open you can press F2/Cmnd-2 to open it or bring it to the top, or you can select the **Browser** option from the View menu, or in Windows you can click on the **Browser** button on the main Omnis toolbar (you can enable the toolbar via the View menu).
- Click on the **Project Libraries** option to create a new library or open an existing library



Create New Project Library

Web and Mobile
Browser based responsive web app

Desktop
Desktop application for Win/Mac

Blank
Start without presets

Recent Project Libraries

Name	Location	Last Used	Size
newlibrary.lbs	Applications	Today, 09:39	30 KB
pics2.lbs	Users/andrewsmith/Library/Application Support/Omnis/Omnis Studio 11/welcome/tut...	Yesterday	71 KB

Help

omnis studio
How to create a project library from scratch
Follow the step by step Omnis Studio tutorial.

omnis forum
Get Help with your project library
Ask a question on the Omnis developer forum.

omnis studio
Samples
Take a look at sample libraries

Figure 68:

- To create a new library, click on one of the options under **Create New Project Library,** enter a name for the new library, including the **.lbs** file extension and click on Save or press Return

The New Project Library options are:

- **Web and Mobile**
  creates a new library containing a **NewRemoteForm** template ready for you to start designing & coding your first web and mobile form layout; the library also contains a **Remote_Task,** a **Startup_Task,** and a folder containing various **System Classes** You can click on the Test option, or Press Ctrl-T, to open the form in a web browser, or you could double-click on the button (in design mode) to view its code in the Method Editor.

See JavaScript Remote Forms for more information about creating Remote forms and Remote tasks, or JavaScript Components for information about adding components and other objects to your remote form.

- **Desktop** (hidden in the Community edition)
  creates a new library containing a **NewWindow** template ready for you to start designing a form for Windows or macOS desktop use only; the library also contains a **Startup_Task,** and a folder containing various **System Classes**
- **Blank**
  creates an empty library containing only a **Startup_Task** and the **System Classes**

See Startup Task and System Classes for more information.

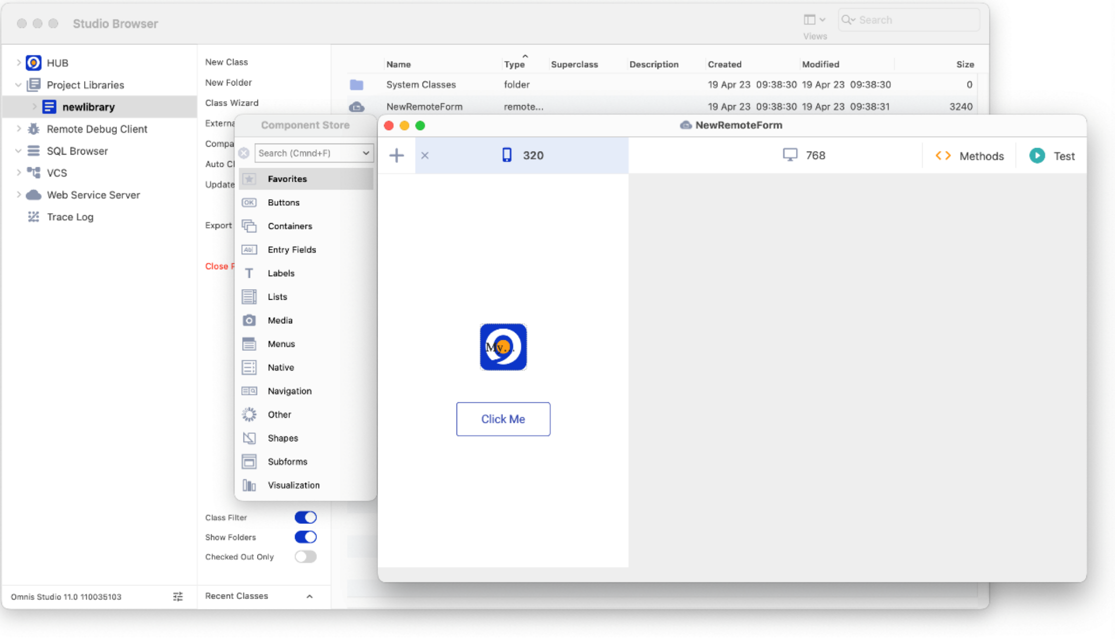


Figure 69:

Library Name

When you name a new library, you can use the file naming standards for the current operating system. The .LBS file extension is not obligatory, i.e. on macOS, but it will help you distinguish library files from other types of file in your file system. The new library is opened in the Studio Browser under the **Project Libraries** option and shown as a single icon in icon view, or as a single line in details view.

You can also create a new library by importing a JSON tree using the **Create Project Library from JSON** option in the Studio Browser (not available in some editions): the JSON tree must previously be exported from Omnis Studio using the export to JSON option: see Importing Libraries.

Note that libraries created in the Community Edition cannot be opened in the Professional Edition of Omnis Studio, and vice versa.

Restoring Open Libraries & Classes

When Omnis starts up, it opens any **libraries** that were open at shutdown, together with any class editors that were open in design mode (this applies to the development version only). This is controlled by the **restoreOpenLibsAtStartup** item in the “ide” section of the config.json file, that defaults to true.

When the development version of Omnis shuts down successfully, it saves a list of libraries to re-open. The library list saved excludes all libraries in the startup and studio folders, and all private libraries (these libraries will typically re-open anyway). In addition, Omnis will only run the startup task of a library that it re-opens, if the startup task was open when Omnis last shut down successfully. Running the startup task is controlled by the **openStartupTaskWhenRestoringOpenLibrary** item in the “ide” section of config.json. If this is true (the default), Omnis will run the startup task of each library that had an open startup task when Omnis closed. Set this to false if you do not want the startup tasks of libraries to be run.

In addition to libraries, Omnis opens any **class editors** that were open at shutdown when it next starts up (this applies to the development version only). This is controlled by the **restoreOpenClassEditorsAtStartup** item in the “ide” section of the config.json file (the value of this configuration entry is ignored, and treated as false, if the restoreOpenLibsAtStartup entry is false). If true (the default), after completing startup, the development version of Omnis tries to re-open class editors that were open when Omnis last shut down successfully. Note the system table editors are not reopened.

For class editors other than remote form and window editors (which automatically save their last position), the restored editors open in their last screen position, provided that the screen configuration has not changed since Omnis was shut down; if the screen configuration has changed, then the editors open at their default position for the new screen configuration.

The method editor attempts to restore the selection to the method line being edited. The remote form, report and window editors attempt to restore their current selection. These attempts will work unless the class has been changed, that is, by replacing the library (or class) with a modified copy before restarting, e.g. from the VCS.

The restored class editors open behind any user windows opened by either startup libraries or libraries opened due to the restoreOpenLibsAtStartup config.json entry.

Opening a Library

To open an existing library, select the **Project Libraries** option in the Studio Browser and click on the **Open existing project library** button and navigate to the library file to open it. If you have already opened the library, you can select its name from the **Recent Project Libraries** list. Alternatively, you can double-click on a library file (icon) on your desktop to open it in Omnis Studio.

If you open a library created in a previous version of Omnis Studio it will need to be converted: see Library Conversion.

Opening a Library in code

You can open a library in your code, which can be useful if your application needs to open additional libraries at runtime. You can use the $add() method which has the following syntax:

```
Do $libs.$add(cPath [,bCreate=kFalse, cIname, cPword, &iErrCode, &cErrText, iFlags=kLibFlagNone, startupPar
```

Opens or creates library at cPath and returns an item reference to the library. The parameter cIname overrides the default internal name, and cPword is the library password (if required).

If an error occurs, the returned item reference is NULL; if passed, the iErrorCode and cErrorText parameters identify the error.

The iFlags parameter allows different options to be set when the Omnis Runtime opens a library. The iFlags parameter is a sum of one or more of the following kLibFlag… constants:

Constant	Description
kLibFlagNone	If specified, has no effect
kLibFlagDoNotOpenStartupTask	If specified, do not open the Startup_Task
kLibFlagEnableConversionByRuntime	The Omnis runtime version will offer to convert the library
kLibFlagConvertWithoutUserPrompts	If specified, and conversion is allowed, Omnis will immediately perform conversion without giving the user any prompts that require a response; note the user cannot cancel the conversion in this case

Any further parameters are passed to the $construct method of the Startup_Task (in the case where the Startup_Task is allowed to run).

If you prefix cPath with “!!!”, then Omnis does not open the startup task of the new library.

Closing a Library

To close a library, select the library in the Studio Browser and click on the **Close Project Library** option, or you can right-click on the library icon in the Studio Browser and select **Close Project Library** from the context menu.

Closing All Libraries

In the *Runtime (desktop)* and *Server* versions of Omnis Studio only, you can close all open libraries in a single command using the **Close All Libraries** option in the **File** menu; *note this option is not available in the Development version.*

Library Properties

A library has a set of *Properties* and *Preferences* that control its settings and behavior. You can use the Property Manager to display or change the properties of a library, or the properties of a library can be accessed in your code using the notation: $clib.$<PropertyName>, or LibraryName.$<PropertyName> in a multi-library application.

To view the properties of a library

- Select the library icon or name in the Studio Browser and press **F6/Cmnd-6** to open the Property Manager, or click on the **Props** button on the main Omnis toolbar

or

- Right-click on the library icon or name in the Studio Browser, and select the **Properties** option from the library context menu

The Property Manager displays the properties of the currently selected library: to view all the properties (preferences) of a library, ensure that the **Advanced** option is enabled. As with any property in the Property Manager, you can move the pointer over a property name to display its help text. The following is a summary of all the Library Properties under the General tab; the default value is provided below, where applicable.

Property	Default	Description
$disabledefaultcontextmenu	kFalse	If true, the default context menu for the object will not be generated in response to a context click ($clib$.disabledefaultcontextmenu and $cobj.$disabledefaultcontextmenu must both be false for the menu to be generated)
$disksize		The total disk size of the file (in bytes)
$extension	kFalse	If true, the library is an extension library
$freesize		The estimated free bytes within the file
$ignoreexternal	kFalse	The ignore external mode for the library
$isprivate	kFalse	If true, the instance or library is private
$name		The name of the library, minus the extension: see Library Default name below
$nodebug	kFalse	If true, the local debugger is disabled for the library
$parentfolder		The pathname of the folder containing the library (with a trailing pathname separator): see below
$pathname		The full pathname of the library, including the library file name
$prefs		The library preferences group is a property of a library, shown on its own tab in the Property Manager: see Library Preferences
$remotedebug	kFalse	If true, remote debugging of this library is allowed. Cannot be set to true in an always private library
$shared	kTrue	If true, the file is open in shared mode. On OSX 64 bit this property cannot be set to true as shared access is not supported
$userinfo		A developer property that can store data of any type. The property manager only allows assignment if its current value is empty, null, or has character or integer data type. Must be character to be used with client methods in the JavaScript client
$userlevel	0	The current user level of the library
$vcsbuilddate		The date and time when this library was built using the VCS
$vcsbuildersname		The name of the user who built this library using the VCS
$vcsbuildnotes		The notes entered by the user when this library was built using the VCS

Library Parent folder

The $parentfolder property returns the pathname of the folder containing the library file (with a trailing pathname separator). Note that this is only visible in the Property Manager when the properties of the library are accessed via the Notation Inspector. Using this property, you could, for example, find the full notation for the library folder (path) and drag it to the Code Editor.

Library Methods

The Property Manager shows any methods for a library. The $modifypasswords([bSilent=kFalse]) method opens the #PASSWORDS system table which contains the master and user passwords for desktop libraries only, not JS Client applications.

The $overridetables method lets you override individual entries within the System Class tables at runtime: see Overriding System Class Tables.

Library Preferences

The library preferences are displayed under the **Prefs** tab in the Property Manager when viewing the properties of a library, or they can be accessed in your code using the notation: $clib.$prefs.$<PropertyName>, or LibraryName.$prefs.$<PropertyName>. The following is a summary of all the Library Preferences; the default value is provided, where applicable.

Property / Preference	Default	Description
$alwayslog	kFalse	If true, the Send to trace log command and tr always write non-diagnostic messages to the the check for debuggable code)
$centuryrange	1980	The start of the default range for dates entere
$defaultname		Default internal library name: see below
asof 35439 $disableclassdatanotation	kFalse	If true, $classdata for all classes in the library v **Setting this property is an irreversible operat**
asof 35439 $disablemethodtextnotation	kFalse	If true, $methodtext and $methodlines will no **Setting this property is an irreversible operat**
$disablewebservicelogging	kFalse	If true, WSDL Web Service Server logging doe requests to services in the library
$errorprocessing	kEPreport	A kEP... constant that indicates how unhandle belonging to this library are processed kEPlog kEPlogStackAndReport kEPreport
$exportcontrolcharacters	kFalse	If true, export types which normally map cont spaces,leave the data unchanged
$exportedquotes	kTrue	If true, exported text is enclosed in quotes
$fiscalyearend	31 Dec 1900 00:00:00	Fiscal end of year date
$iconlib		Internal name of the alternative library for ret #ICONS
$iconsets		Comma separated list of icon set folders to be Folder names datafile, lib, studio and studioid cannot be used. Omnis searches the icon sets order specified by this property
$initiallayoutbreakpoints	320,768	A comma separated list of layout breakpoints $layoutbreakpoints when making a remote fo
$justifiedtextthreshold	75	A percentage value,0-100. For fields with justi the minimum percentage of the total field wi paragraph line must occupy before the line of fill the entire field width
$osdroplimit	100000000	Maximum number of bytes of dropped data t in pDragValue for evDrop when $osdropflags kOSDROPincludeData. kOSDROPwithoutDataIfOsDropLimitExceede still occurs when the limit is exceeded
$reportcalculationerrors	kTrue	If true, Omnis reports errors that occur during evaluation
$reportnotationerrors	kTrue	If true, notation warnings will be handled as e
$sensitivefieldnames	kFalse	If true, field names are case sensitive
$serverlessclientstringtable		The string table (tab-separated value .tsv file i shared by all JavaScript client remote forms in Serverless Client Application File. Only assign number enables SC development
$sharedpictures	kSharedPicModeNone	Indicates if Omnis uses shared picture format kSharedPicModeNone, kSharedPicMode256C kSharedPicModeTrueColor
$sqlstripspaces	kTrue	If true, Omnis strips trailing spaces from retrie columns; provides backwards compatibility w
$startuptaskname	Startup_Task	The name of the startup task class
$styleplatform	<the current platform>	The field styles platform group to use for this kJavaScript kmacOS kMSWindows kUnix
$userexportdelimiter	;	The character the library uses for user-delimit
$validcolumninbadrowisnull	kFalse	If true, a valid list column in a bad (non exister #NULL rather than an empty character string
$weekstart	kMonday	The beginning of the week, a day constant: k kWednesday kThursday kFriday kSaturday kS

Disabling Class Data and Method Text

The library preferences $disableclassdatanotation and $disablemethodtextnotation control whether or not other Omnis libraries can access class data and/or method code within the library. They can be set using the Property Manager or via the VCS when building a library.

When $disableclassdatanotation is kTrue for a library, you will no longer be able to read or write $classdata from any Omnis class using Omnis code. In addition, JSON export of the library is disabled as the class data is disabled. IMPORTANT: future access to this library in the Omnis VCS will no longer be possible.

When $disablemethodtextotation is kTrue for a library, you will no longer be able to read or write $methodtext or $methodlines from Omnis code or via the Property Manager. In addition, method text will not be exported during a JSON export of the library.

Library Default name

A library has an internal name stored in $name which Omnis uses to reference the library in your code and elsewhere, such as the prefix for a class name in a multi-library structure, i.e. Libraryname.classname. The default internal library name is created automatically using the name of the disk file, with the file extension removed. Any remaining characters in the set . $ ( ) [ ] (dot, dollar, open and close parenthesis, open and close square bracket) are converted to _ (underscore), although you should avoid using these characters in your library name. The default internal library name will have the case of the library name to be consistent with Omnis running on the macOS and Linux platforms. (Prior to Studio 10, library name and file paths on the Windows platform were converted to upper case when opened which resulted in the default internal name for a library being upper case on the Windows platform.)

You can assign an alternative internal name for a library by setting the $defaultname library property and once set this is used to reference the library, overriding the auto-generated name. The characters and format of the string allowed in $defaultname are limited: no leading or trailing spaces, the name cannot start with a digit, or contain the following characters: . $ ( ) [or].

Omnis prevents the internal name of a library from being set to the name of a static function group, such as FileOps, by appending a digit (or digits) to the internal name that would otherwise be used, in the same way as it does when opening a library which would result in a duplicate internal name. So in the case of FileOps, the internal name of the library would typically be fileops1. However, it is best to avoid using a function group name, or any other function or command name, as a library name to avoid any possible conflicts.

If you rename the library file on disk the $defaultname remains the same retaining all class references. If you change the $defaultname property after you start developing your library, all class references that use it will fail: therefore, in a multi-library system you should set it once before you start adding classes to your library.

Multi-library Projects

Omnis lets you structure your application into one or more libraries that you can load either together or separately. This lets you

- break up your application into smaller, less complex subsystems
- develop the subsystems in different programming groups or departments
- test the subsystems or modules separately
- reuse libraries in different applications, mixing and matching reusable code without modification

Although Omnis always ensures the integrity of objects, there is no built-in locking or concurrency checking to prevent two users from modifying the same object. If more than one user opens an object in design mode, the last one to close the object overwrites the changes made by the other users. There is no way to ensure that changes made to an object are seen by other users before the library is reopened: objects are cached in memory and it is not possible to predict when Omnis will discard an object from the cache. In a team of developers you should therefore use the Omnis VCS.

Library APIs

You can use the $getapiobject method to expose the methods in a library to be used in another library. The $root.$modes method **$getapiobject** returns a reference of an object in another library, allowing you to use its methods. You can use $root.$modes.$getapiobject(“libraryA”) from libraryB to call into $getapiobject method of libraryA startup task which must return an object reference. For example, the startup task method $getapiobject of libraryA can do:

```
Quit method $clib.$objects.libAPI.$newref()
```

Where libAPI is an object within the library which implements some methods that you can use. If libraryA does not return an object reference, the returned value to the caller is NULL. The startup task in the called library must be named ‘Startup_Task’ (the default name) in order for this to work.

Omnis VCS

The Omnis VCS provides you with a full-featured version control system for your Omnis libraries and other components. If you put your application under version control, you eliminate the inherent risks involved in group development. See later in this manual for details about the Omnis VCS.

Comparing Classes

The Studio Browser includes a tool that lets you compare classes in two different versions of the same Omnis library or different revisions of the same class in a VCS project. The Compare Classes tool lets you compare all the classes in one library or VCS project or individual classes. To use the new tool, click on the Compare Classes option in the Studio Browser.

Shared Access to Libraries on macOS

The 64-bit macOS version of Omnis Studio does not support shared access to libraries. For libraries ($root.$libs.LIB) on 64-bit macOS, the $shared property *cannot be set* to true as shared access is not supported: in effect, this property is redundant on this platform.

Starting Omnis with a file

Omnis can be started up by double-clicking on a file, such as a library file, or via the command line. The sys(250) function returns a list of files which were used to open Omnis, e.g. double-clicked from the Finder or passed on the command line. This is empty if Omnis was opened directly by double-clicking.

In addition, the task method $openfiles can be overridden and will be called when Omnis is used to open a file or set of files by the OS. This will be passed the list of files as a parameter.

On Windows, the $singleinstance root preference needs to be set to kTrue to use the same instance of Omnis to open a file, otherwise another instance of Omnis will be started.

Opening Initial File As Library error

The **reportErrorOpeningInitialFileAsLibrary** item in the “defaults” section of the config.json file (default value true) allows you to control whether or not Omnis reports an error trying to open the initial file as a library; this applies when a file is dropped on the Omnis program, or the file is double-clicked. If the option is set to false, the error message is not shown.

Library Conversion

When you try to open an existing Omnis library, created in a previous version of Omnis Studio, *Omnis will prompt you to convert the library*. You should make a secure backup of the old library before you open and convert it in the latest version of Omnis Studio since *THE LIBRARY CONVERSION PROCESS IS IRREVERSIBLE*.

The library conversion prompt applies to libraries you open manually in Omnis Studio and any libraries located in the Startup folder that are loaded automatically and require conversion.

Class Locking and Library Conversion

In order to enhance the integrity and security of deployed Omnis Studio libraries, the mechanism used to lock classes in a private library has changed in Omnis Studio Revision 35659.

Consequently, all libraries opened in Omnis Studio 11 revision 35659 or later *WILL REQUIRE CONVERSION, INCLUDING LIBRARIES CREATED WITH ALL PRIOR REVISIONS OF OMNIS STUDIO 11* (as well as Studio 10 or earlier libraries). *THE LIBRARY CONVERSION PROCESS IS IRREVERSIBLE.*

THEREFORE, AND IN ALL CASES, YOU SHOULD MAKE A SECURE BACKUP of all existing Omnis Studio 11 libraries BEFORE OPENING THEM in Omnis Studio 11 Revision 35659 or later.

Library Conversion Logs

The library converter adds an entry to the Find and Replace log that allows you to quickly navigate to each change made by the converter by double-clicking on a line in the log. In addition, the converter writes a log file to the ‘conversion’ folder in the logs folder in the data part of the Studio tree. The log file provides a more permanent record of the changes applied to the converted library.

Conversion Log Delimiter

The conversion log file uses tab-delimited format, with exported text in quotes (the default). You can change both of these options using configuration items in the config.json file, in the log section:

```
"conversionLogDelimiter": "\t",
"encloseConversionLogTextInQuotes": true
```

If conversionLogDelimiter is empty, Omnis uses the default log delimiter, a semicolon (;).

Conversion Prompts & Working Messages

You can enable the option “disableAllLibraryConversionPrompts” to suppress conversion prompts when a library that needs to be converted is opened from the Startup folder: the option is in the “defaults” section of the Omnis Configuration file (config.json) and defaults to false, so you can set this option to true to prevent library conversion prompts.

You can disable the working messages such as “Converting class…” during library conversion by setting the “showLibraryConversionWorkingMessage” option located in the “defaults” section of the config.json file. The option defaults to true, but you can set it to false to disable the conversion working messages.

Default Classes

When you create a new library in Omnis, it contains certain default classes including a task class called *Startup_Task,* a remote task called *Remote_Task* (in a web/mobile app), and various *System Classes* that control the behavior and appearance of your library. As you begin to prototype your application, you don’t need to modify the default classes, but this section gives you a brief overview of how they affect your library.

Startup Task

When you create a new library, it contains a task class called Startup_Task. When you open your library in design mode (or in a runtime environment), the Startup_Task is opened and the initialization code within it is run automatically. Therefore, if you want your library to perform some action when it starts up (such as open a remote form or a database session), you can put the code to do it in the $construct method in the Startup_Task. (Many of the example apps under the **Samples** option in the **Hub** use the Startup_Task to setup the data or open the forms for the demo, so examine those to get an idea how the Startup_Task can be used.)

The Startup_Task is not relevant for web or mobile apps, since the end user will be opening your application in their web browser or application wrapper: any initialization of a web or mobile app should be done in the initial remote form to open in the end user’s browser.

Each library has a preference called $startuptaskname which stores the name of the startup task, and is set to Startup_Task by default. To change the task that is run when your library opens, you need to change this property, but in most cases you can leave it set to Startup_Task.

The startup task has a special function when you are designing your library and adding other classes and variables to your library. When you start to create or prototype your app you don’t need to change the Startup_Task, so you can proceed to create your data schemas and UI classes in your library.

Library Startup Task

The Omnis root preference $clibstartuptask reports the startup task for the library containing the current executing method.

Open/Close Library Notifications

There is a task message $openlibschanged that is sent after a library or libraries have been opened or closed. It is sent in a development version of Omnis only.

Remote Tasks

To open or test a Remote form it needs a Remote Task, therefore you need to create a Remote Task in your library and assign it to the $designtaskname property of the remote form. If you use a wizard to create a Remote form, you can create a remote task during the wizard process which is assigned to the remote form automatically.

When you create a new web/mobile library using the **Create New Project Library** option in the Studio Browser it will contain a **Remote_Task** class.

System Classes

Every new library contains a number of *System Classes*, contained in a folder called 'System Classes'. You can hide or show them using the Class Filter option in the Studio Browser (the Class Filter option is visible when the library is selected).

System classes are special types of class that hold information about the Omnis environment, including field styles, fonts, input masks, and external components. You can edit some of the system classes to change the way Omnis behaves. *The settings for these tables are stored for each separate library.* You can copy system classes from one library to another and you can edit them, but some options available for normal classes are not available for these tables. Like other classes, you can check system classes into the Omnis VCS, so you could maintain one set of system classes for use with a number of different libraries in a multi-library project.

System class	Description
#BFORMS	Boolean formats: these specify the format of Boolean fields allowed in your library
#DFORMS	Date formats: these specify the format of short date, and date and time values
#EXTCOMPLIBS	the External components available in the current library: here you can load or remove ext comps for your library or Omnis itself
#ICONS	the icon datafile for the current library used in legacy apps (double-click to edit it); use *Icon sets* for new web and mobile apps using the JS Client
#JSMASKS	input masks for edit controls in *remote forms*
#JSWFONTS	font table for *remote form* classes
#MXRFONTS#WIRFONTS#UXRFONTS	font table for *report* classes under macOS, Windows, or Linux/Unix; cross platform apps may contain all these
#NFORMS	number formats for numeric data entry fields
#STYLES	character styles for window and report fields, and text objects: you can print a list of styles by right-clicking on the class and selecting Print Class
#TFORMS	text formats for character-based fields
#DEBUGGER	the current local debugger code breakpoint locations, which means code breakpoints (and their conditions) are restored when a library is reopened. #DEBUGGER does not appear in the Studio Browser class list, but it is included in $clib$.classes

The following system classes relate to window classes and desktop library user access only, and therefore are not used for web or mobile apps:

System class	Description
#MASKS	input masks for data entry fields in window classes only
#MXWFONTS#WIWFONTS#UXWFONTS	font table for *window* classes under macOS, Windows, or Linux/Unix; cross platform apps may contain all these
#PASSWORDS	the master and user passwords for desktop libraries only, not JS Client: this is hidden by default, but you can show it via the Class Filter option or by pressing Shift+Ctrl+A in the Studio Browser

You can edit a system table by double-clicking on it in the Studio Browser. For example, you can double-click on #DFORMS which opens the Date formats dialog showing all the date formats for the current library.

By using the FONT system tables for remote forms, report classes (or window classes), you can map fonts used on one operating system to fonts appropriate for the other operating systems.

#BFORMS: Boolean formats

Formats used for the $formatstring property in Masked Entry fields when $formatmode is set to kFormatBoolean.

	Formatting
1	T
2	t
3	Y
4	y
5	O
6	1
7	[GREEN]O;[RED]O
8	[GREEN]Y;[RED]Y

#DFORMS: Date formats

The entries in #DFORMS are used to specify the format of Date Time variables (when defined in the Method Editor), and for the $formatstring property in Masked Entry fields when $formatmode is set to kFormatDate.

	Formatting
1	D m y
2	H:N
3	h:N A
4	H:N:S
5	H:N:S.s
6	D m Y H:N
7	D m Y H:N:S
8	D m Y H:N:S.s
9	M/D/Y
10	w, n D, y
11	D/M/Y
12	w, D n, y
13	M/D/Y h:N A
14	D/M/Y H:N

The following standard date formatting characters are supported:

Character	Description
D	Day (12)
V	Day of week (Fri)
w	Day of week (Friday)
E	Day of year (1..366)
n	Month (June)
M	Month (06)
m	Month (JUN)
y	Year (1989)
Y	Year (89)
A	AM/PM
H	Hour (0..23)
h	Hour (1..12)

#JSMASKS or #MASKS – Input masks

Formats used for the $inputmask property in Masked Entry fields.

	Formatting
1	>>(###) ###-####
2	>>(###) ###-#### Ext(#####)
3	>>(####) ######
4	>>(####) ###### Ext.###
5	>>#### #### #### ####
6	>>###-###-###
7	>>aa ## ## ## a
8	>>##-aaa-##

	Formatting
9	>>##-aaa-####
10	>>##/##/##$_{\text{D/M/Y}}$
11	>>##/##/####$_{\text{D/M/y}}$
12	>>##/##/##$_{\text{M/D/Y}}$
13	>>##/##/####$_{\text{M/D/y}}$

#NFORMS – Number formats

Formats used for the $formatstring property in Masked Entry fields when $formatmode is set to kFormatNumber.

	Formatting
1	0
2	0.00
3	#,##0
4	#,##0.00
5	#,##0;[RED](#,##0)
6	#,##0.00 ‘cr’;#,##0.00 ‘dr’
7	‘£’ #,##0;[RED]‘£’-#,##0;;‘Nil’
8	‘$’ #,##0;[RED]‘$’-#,##0;;‘Nil’
9	
10	
11	0.00 E+00
12	0.00 E-00

#TFORMS – Text formats

Formats used for the $formatstring property in Masked Entry fields when $formatmode is set to kFormatCharacter.

	Formatting
1	‘(’@@@‘)’ @@@‘-’@@@@
2	‘(’@@@‘)’ @@@‘-’@@@@ ‘Ext(’@@@@@‘)’
3	‘(’@@@@‘)’ @@@@@@
4	‘(’@@@@‘)’ @@@@@@ ‘Ext.’@@@
5	@@@@ @@@@ @@@@ @@@@
6	@@@‘-’@@@‘-’@@@
7	@@ @@ @@ @@ @U

Overriding System Class Tables

You can override individual entries within the System Class tables at runtime, without modifying the system classes in the library. This may be useful in a multi-library deployment, where all the libraries need to share the same base set of system classes, but you may want to change individual settings in the formatting tables, such as the date format, according to the language or location of the end-user.

The definitions for these alternative formatting tables can be stored in a JSON file, which should be named “tables.json” and placed in the Studio folder under the main Omnis folder. You can use the $overridetables method to load an entry from the JSON file to override an entry in one of the default system tables in the current library.

The override only applies while the library is open: therefore, if you close and re-open the library, you need to call $overridetables again if you want to override the default system tables: typically, you would do this at the start of $construct in the Startup task of your library.

The tables.json file should contain a JSON object, and each member of the JSON object defines the content of one or more of the formatting tables: tables which do not have an entry for a member are not affected when that member is used. The following format is used:

```
{
  "en": {
    "#BFORMS": [ "[GREEN]Y;[BLUE]Y","y","Y" ],
    "#DFORMS": [ "D/M/Y H~N", "D m Y  H:N"],
    "#TFORMS": [ "@@ @@ @@ @@ @U", "'('@@@@')' @@@@@@ 'EXT'@@@"],
    "#MASKS": [ ">>###-###-###", ">>##/##/####~M/D/y~"],
    "#NFORMS": [ "0.00 E+00"]
  },
  "de": {
    "#BFORMS": [ "[GREEN]Y;[YELLOW]Y" ],
    "#DFORMS": [ "D/M/Y H~N" ]
  }
}
```

The $overridetables method has the following syntax:

- $clib.$overridetables(cJsonPath,cEntry[,&cErrorText])
  Uses member cEntry in JSON table file cJsonPath to override the system tables with entries stored in member cEntry. Returns Boolean true for success, false and cErrorText for failure

Therefore, you could execute the following to load tables.json from the Studio folder:

```
Do $clib.$overridetables(con(sys(115),"studio",pathsep(),"tables.json"),"en",lErrorText) Returns bStatus
```

Locale-style names have been used, such as “en”, to indentify the members for which the tables are to be loaded, but the text could be anything to identify the set of tables to be loaded so long as you use the same name in the $overridetables() method.

$overridetables replaces the contents of the system tables with the members of the array, and sets any entries after the array members to empty. The corresponding system table class becomes read-only: you can open its class editor, but you cannot change it within the class editor.

Using the $clib.$prefs notation group for the table will change the table used at runtime, but a change made using the notation will not be saved to disk.

Class Types

There are several different types of class in Omnis, each one performing a particular function in your library, or your application as a whole. In general, classes are either *Data classes* or *UI classes,* depending on whether they define the data structures or the UI and other visual elements in your application. The types of class are:

- **Schema class**
  data class that defines a server table and its columns on your server database; see Schema Classes
- **Query class**
  data class that defines one or more server tables and their columns on your server database; see Query Classes
- **Table class**
  data class that maps to a schema class and contains default methods for processing your server data; see Table Classes

See SQL Classes and Notation for more info.

- **Remote form class**
  UI class that defines the forms to be displayed on web or mobile devices using the JavaScript Client; all remote form instances require a remote task to open; see JavaScript Remote Forms
- **Remote menu class**
  UI class that that can be used by the JS Popup menu component; see Remote Menus
- **Remote task class**
  a remote task class controls JavaScript remote form instances and a remote task class controls JavaScript remote form instances and maintains the connection between a client and the Omnis App Server in web and mobile apps; see Remote Tasks
- **Report class**
  UI class that defines the reports that can be generated in your code, saved to a file and displayed in the JS client as a PDF; see PDF Printing

- **Task class**
  all libraries contain a Startup_Task class used to initialize the library when it is opened (not required for web & mobile apps, use a Remote task class); otherwise, a task class can be used to control instances and events in a Desktop application
- **Object class**
  class that contains methods and variables defining your own structured data objects that can be instantiated in your UI
- **User Constants**
  class that contains user-defined constants for use in your methods and expressions. A user constant is a named value, where the value cannot be changed during execution; see User Constants
- **Code class**
  class that contains methods you can use throughout a library, but only in server executed or local desktop code (code classes cannot be used in client methods so should not be used for web or mobile apps using the JavaScript client)

The following classes are used for Desktop applications (thick client) only, and should not be used for JavaScript web or mobile applications (these are hidden in the Community Edition):

- **Window class**
  UI class that defines the data entry windows and forms for Desktop apps only (do not use for web or mobile apps)
- **Menu class**
  UI class that defines standard pulldown, popup, and hierarchical menus for Desktop apps only (described under Window components)
- **Toolbar class**
  UI class that defines the toolbars for Desktop apps only; the toolbars can be installed on the main Omnis toolbar or floating in the application window (described under Window components)

See Window Components for information about Window, Menu, and Toolbar classes.

The following classes are used for accessing Omnis data files in legacy applications, and are only included for backwards compatibility, so they should not be used for new applications (these are hidden in the Community Edition):

- **File class**
  data class that defines the structure of a file (the Omnis equivalent of a table) in an Omnis data file
- **Search class**
  data class that filters the data stored in an Omnis data file

Creating New Classes

The Studio Browser gives you the option of creating new classes using the blank or empty class templates, using the **New Class** option, or using the class wizards which let you build fully functional classes complete with data fields and methods created for you automatically. These methods of creating classes are accessed using the list of hyperlink Options in the Studio Browser.

The **New Class** option in the Studio Browser (available when a library is selected) lets you create a blank or empty class that you can build and modify from scratch. Templates are available for all the different class types in Omnis. If you are new to Omnis or you want to prototype your application quickly, you can use the **Class Wizards.** When you are more experienced, or you want to create classes from scratch, then you can use the **New Class** templates.

Class Names

The name of a class can be anything you like, but the name would normally take account of its function within your application. You may like to use a prefix in the class name to denote the type of class, so for example, you could prefix the name of a Schema Class that is linked to a Customer server table with "s", therefore you could name it "sCustomers". Similarly, a JavaScript Remote Form used for Customer information could be named "jsCustomerForm", prefixed with "js" to denote a JavaScript remote form class.

To remove all possibilities of any conflicts or errors in Omnis code, it is recommended that you only use **alphanumeric characters** and **do not use spaces** in class names; however, you can use _ (underscore) to separate words if necessary.

You cannot use the following characters in class names: any character with a value less than space, a character in the string ” . , ; : ! ?)] } ([{ + - * /| & > < = “, or a single or double quote character. In addition, a class name cannot start with the dollar character $.

Remote Form names should not include the hash symbol (#) or other special symbols since this may cause unexpected results in a web browser, or in the case of #, the remote form may not open in test mode at all.

Extra validation

For Studio 11, stricter class name validation was introduced to prevent certain characters from being used in class names, including most of the characters described above. The **extraClassNameValidations** item in the 'defaults' section of the config.json file is set to True meaning that Omnis performs extra validations before assigning a name to a class. This extra validation is strongly recommended, as using the characters it excludes in class names can cause confusion and potentially errors. This property is present only to allow code from previous versions to continue working if the additional validations cause any code to fail.

Class Wizards

The **Class Wizards** option in the Studio Browser (available when a library is selected) lets you create fully functioning classes (e.g. remote forms) based on the selections you make during the wizard, and in the case of UI wizards, the class you are building can be linked to other classes in your library. The class wizards are good for building or prototyping your application very quickly and save you building classes from scratch. For example, you can create a JavaScript remote form based on a schema class in your library using the **SQL JavaScript Remote Form** wizard.

Data Classes and Wizards

Depending on the type of data you want to enter or retrieve in your client application you will need to define certain structures in your library to handle this data. These structures are stored in your library file as *data classes*.

This section introduces schema, query, table, file, and search classes. All these classes are covered in greater detail later in this manual.

Accessing SQL databases

If you want to handle data from a SQL-compliant DBMS, you must create Omnis *schema* and/or *query classes* that map to the table and column structures on your server database. You can create schema classes from scratch, or you can create them automatically from the tables on your database server using the SQL Browser (the Tutorial shows you how to do this).

Data type mapping

When creating schema classes, you need to choose column data types that map directly to the tables or views on your server database. To do this successfully, you need to choose the data type for each column that best represents the type of data in your database server. See SQL Programming for more information.

Omnis Datafiles

File classes and Search classes are used for accessing Omnis data files in legacy applications, and are only included for backwards compatibility, so should not be used for new applications.

If you want to store and retrieve your data using a non-client/server setup, you can store it in an *Omnis data file*. In this case you need to design the structure for your data using Omnis *file classes*. In addition, you can use an Omnis *search class* to filter the data stored in an Omnis data file.

A good alternative to using an Omnis data file is to use SQLite and if you are converting a legacy Omnis app that uses an Omnis data file, you may like to convert your data to SQLite using the Convert Data File to RDBMS option, available under the **Add Ons** option in the **Tools** menu.

Omnis Data Types

This section describes in detail the standard data types you can use to represent data in Omnis. Choosing the right type for your data ensures that Omnis will do the right thing in computations requiring conversion. It also lets Omnis validate the data as you enter or retrieve it. Some of the basic data types have subtypes, or restrictions of size or other characteristics of the data that give you finer control over the kind of data you can handle. The following data types are available.

Field or Variable Type	Description
Character	standard character set sorted by ASCII value or UTF32 encoded characters
National	Same as Character but sorted by National sort order

Field or Variable Type	Description
Number	multiple types for representing integers, fixed point and floating point numbers
Boolean	single-byte values representing true or false values, or their equivalents
Date Time	multiple types for representing simple dates and times, or composite date and times, between 1900 and 2099 to the nearest hundredth of a second
Sequence	proprietary data type for numbering Omnis data file records
Picture	stores color graphics of unlimited size and bit-depth in platform-specific format or in a proprietary shared picture format
List	structured data type that holds multiple columns and rows of data of any type
Row	structured data type that holds multiple columns of data in a single row
Object	your own structured data type based on an object class
Binary	stores any type of data in binary form, including BLOBs
Item Reference	stores the full notation of an object in your library or Omnis itself
Field Reference	passes a reference to a field (parameter variables only)
Object Reference	lets you create an object instance of local, instance, class or task variable scope

Character

Character data can contain characters from any of the various single-byte standard character sets or UTF32 encoded characters. You can define a Character column of up to 100 million (100,000,000) characters in length, so can store up to 400MB of character data. Omnis only uses the amount needed for each string, and not the maximum amount.

Character columns or fields in Omnis generally correspond to SQL VARCHAR data on database servers and have a varying length format.

In Omnis character data is sorted according to its ASCII character set representation, not the server representation. The ASCII character set sorts any upper case letter in front of any lower case letter. For example, these character values

```
adder, BABOON, aSP, AARDVARK, Antelope, ANT
```

are sorted as

```
AARDVARK, ANT, Antelope, BABOON, aSP, adder
```

National

Like Character data, National data can contain characters from any of the various single-byte standard character sets or UTF32 encoded characters. You can define a National column of up to 100 million (100,000,000) characters in length. However, when you sort National data, Omnis sorts the values according to the ordering used by a particular national character set.

The ordering for the English language follows: A, a, B, b, C, c, D, and so on. For example, if the previous values were values of a national column or field, Omnis would sort them as follows:

```
AARDVARK, ANT, Antelope, adder, aSP, BABOON.
```

If you store data in an Omnis data file, Omnis stores a copy of the ordering in the file along with the data. If you use the data file on another machine, Omnis preserves the original ordering.

Number

A *number* variable can be an integral or floating point number having various storage and value characteristics, depending on its subtype. The following table summarises the different subtypes for numbers.

Number type	Storage (bytes)	Range
Integer		
Short integer	1	0 to 255

Number type	Storage (bytes)	Range
32 bit integer	4	-2,000,000,000 to +2,000,000,000
64 bit integer	8	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
Number		
Short 0 dp **	4	Single-precision floating pointapprox -3.4E38 to +3.4E38 *Display format: n decimal places
Short 2 dp **	4	e.g. 123.50
Floating dp	8	Double-precision floating pointapprox. -1.8E308 to +1.8E308 *Displayed using 16 significant digits
Number 0 dp	8	Display format: n decimal places,e.g. 123
Number 1 dp	8	e.g. 123.4
Number 2 dp	8	e.g. 123.45
Number 3 dp	8	e.g. 123.456
Number 4 dp	8	e.g. 123.4567
Number 5 dp	8	e.g. 123.45678
Number 6 dp	8	e.g. 123.456789
Number 8 dp	8	e.g. 123.45678901
Number 10 dp	8	e.g. 123.4567890123
Number 12 dp	8	e.g. 123.456789012345
Number 14 dp	8	e.g. 12.34567890123456

*When Number fields are used to store integer values, the largest scalar value that can be stored precisely is 9007199254740992 (2^53). Values larger than this will incur rounding error proportional to their magnitude. Numeric values with precision > 16 will also incur rounding error.

** The types Short 0 dp and Short 2 dp can temporarily store numeric values outside their range, until they are converted to their CRB storage format. To prevent this behavior and ensure the correct values are stored, you can add validation to your code.

File Classes and Integers

You can use 64 bit integers in file classes, provided that a field (column) is not indexed. In Integer subtype droplist in the file class editor, the 64 bit subtype is available but only when the field is not marked as indexed, otherwise if the field is indexed 64 bit is not present: the 64 bit subtype is not allowed in compound indexes either. The notation to manipulate file classes and indexes does not allow 64 bit integers to be used for indexed fields.

Floating Point Numbers

There are many pitfalls in using floating point numbers in programming. Computers do not represent these numbers exactly, only approximately within the precision of the machine. This can lead to all kinds of anomalous problems with comparison of values, particularly with equality comparisons. Two floating point numbers may differ by an infinitesimal amount depending on the technique used to generate the values, even though logically they should be the same.

In general, you should not use equality comparisons with floating point numbers. If you are working with “fixed-point” data such as money values, use scaled integers for comparison and arithmetic.

For example, instead of comparing two floating point variables F1 and F2 containing the amounts $5.00 and $10.00, compare two integer variables I1 and I2 containing 500 and 1000. Display I1 * .01 when you need a decimal value. You can also use the *rnd*() function to round the numbers to a certain number of decimal places before comparing them.

Boolean

The *boolean* data type represents single-byte values of true (yes), false (no), empty, or null. You should take care to give each Boolean column or field an initial value, because Omnis initializes boolean data to “empty”, not NO or null.

When used in a data entry field in a window, boolean data is treated as three characters in which any data entry is interpreted as a YES or NO. A ‘Y’, ‘YE’ or 1 is seen as YES while an ‘N’ or 0 will suffice for No. If the field is a check box, you enter the boolean value by clicking on the box. If you don’t initialize the field and the user does not click on the box, the field has an “empty” value.

You can use boolean values in expressions. The numeric value is 1 for Yes values and 0 for No and empty values. NULL values are treated as undefined in numeric calculations. For example, (null+1) is null and (null>1) is null.

When converted to character strings, Boolean columns or fields can take values "YES", "NO", "NULL", or empty, "". In some cases, for example when setting up search criteria, you can enter values other than these for a Boolean field: in this case, Omnis converts them and matches them with empty. Thus, for example, the value 'FALSE' is converted to empty, as are values like SAM, HAPPY, and so on.

Date time

The date and time group of data types contains three basic subtypes: a four-byte *Short date*, a two-byte *Short time*, and an eight-byte *Long date and time*. The following table summarizes the date and time subtypes.

Date Time subtypes	Storage (bytes)	Range
Short date	4	1900..1999
Short date	4	1980..2079
Short date	4	2000..2099
Short time	2	Minute resolution
Date time(#FDT)	8	Formatted #FDT, to centiseconds
Date time(D m Y)	8	Formatted D m Y, to centiseconds

Note that the display of dates depends on the settings in the #DFORMS system table. Also the long date and time subtypes are identical in value, only displaying differently in window fields.

Short Date

The *short date* type spans the range JAN/1/0000 to DEC/31/9999. There are three specific built in ranges: 1900 to 1999, 1980 to 2079, and 2000 to 2099. By choosing the appropriate range, you can enter just two digits for the year and Omnis recognizes the century correctly. For example, if you select the range as 2000 to 2099, a date you enter as 7,12,57 will be read as 7,12,2057 rather than 1957. To enter a date outside the three specific year ranges, you need to set up your own date display format.

Omnis accepts dates in different formats automatically, with the exact format depending on whether your system is US or European. For example, you could enter the 7th of December, 1998, as any of the following strings.

US system	European system
12-7-98	07-12-98
12/7/98	7/12/98
98	98
DEC 7 98	7 DEC 98

You can use any character to delimit the day and month figures. If you don't specify the year or month and year, Omnis assumes the current year or month and year, respectively.

Omnis supports three kinds of date arithmetic in expressions.

- Addition of days:
  Date + Days = Date (forward)
- Subtraction of days:
  Date - Days = Date (back)
- Subtraction of dates to yield number of days between the dates:
  Date1 - Date2 = Number of Days between the dates

Omnis uses the string variable #FD to define the display format of dates. There are also several date functions that let you manipulate date strings.

Short Time

Short time types have two-byte values in the form HH:NN. The range of possible time values is from 00:00 to 23:59.

You can use time in expressions. Omnis automatically converts the time into numeric values using the conversion HH*60+NN, giving the total number of minutes. The #FT string variable controls conversions between time and string types.

Long Date and Time

The combined *Date Time* type can hold a complete date and time to 1/100th second. It has various subtypes depending on the display format you select (stored in #FDT) and uses 8 bytes when stored in a data file.

Date and Time Calculations

The numeric value of a date or time variable in an expression depends on the format string for that variable. So, if DATE1 has date format string H:N and DATE2 has date format string H:N:S, DATE1 has a numeric value equal to the number of minutes since midnight and DATE2 has numeric value equal to the number of seconds since midnight. It follows that DATE1+1 adds 1 minute to DATE1 and DATE2+60 adds 1 minute to DATE2.

Addition and subtraction involving two date/times cause the numeric value of each to adjust so that they are both based on a common denominator. Thus DATE1-DATE2 returns a numeric value equal to the correct difference between the two times in seconds. However, DATE1*1-DATE2*1 loses the information that DATE1 and DATE2 represent date times and returns a meaningless difference between the DATE1 value in minutes and the DATE2 value in seconds, for example, 500 minutes - 600 seconds.

Note that calculations involving combined dates and times do not work properly if the date part is before 1900. Comparisons between two datetimes with different date format strings work properly.

When you compare parts of dates, for example, the month part of a date, dtm('11 June 98'), Omnis compares the string representation of the month unless some calculation forces it to use the number representation of the month. Thus the expression dtm('11 Dec 98') is less than dtm('11 June 98') because 'D' is before 'J' in the alphabet. To force a correct numeric comparison, add 0. For example

```
If dtm('11 June 98')<(dtm('11 Dec 98')+0)
  OK message {6 is less than 12}
End If
```

You should try to use straight date comparisons if you are comparing full dates. Don't try to convert them into integers or other types of data. Let Omnis do the work for you.

Century Ranges for Dates

When entering data into a date time field or variable without specifying the century, the date normally defaults to be within the hundred year range starting with 1st January 1980. However, you can specify the start of the hundred year default range as a library preference with the option of overriding it for individual date types.

You can use the $centuryrange library preference to set the default century range ($clib.$prefs.$centuryrange), a four digit year is specified which defaults to 1980. So if, for example, $centuryrange is set to 1998, dates for which no century is entered default to between 1st January 1998 and 31st December 2097.

In addition, the 30 date formats which are stored in the #DFORMS system table can include the century range by including a four digit year at the end of the date format. For example, date formats starting at 1st January 1998 include 'D m Y H:N:S 1998' and 'YMD 1998'. This can be used to override $centuryrange for particular date types.

The same mechanism can be used to control the conversion of character values to dates using the dat() function, for example:

```
Do dat(charvar,'D m Y 1998') Returns datvar
```

Century ranges are used when dates are entered from the keyboard or when a character string is converted to a date. If you enter a date that includes the century, the century range is ignored. Century ranges do not affect how a date value is stored or displayed, Omnis always stores the full date including the century.

Sequence

Every time Omnis inserts a new record into an Omnis data file, it assigns a unique number, a *record sequencing number* or RSN to that record. There is a special data type, the *sequence*, for this type of data. Each RSN references a location in the data file. If you delete a record, Omnis does not reuse the RSN. The RSN is stored as a 32-bit integer so its maximum value is 2^32-1, which is approximately 4,295 million! The sequence type is not applicable to client/server data.

Omnis assigns record sequencing numbers (RSNs) according to the following rules:

- The first record in a file has RSN 1, the second record RSN 2, and so on
- An RSN is never used again, even though the record may no longer exist

A window field with sequence type provides a way for the user to see the RSN for any record in an Omnis data file, even though they cannot change it.

Omnis assigns the RSN just before saving the record in the data file, so it is not available for any calculations prior to the *Update files* command.

Picture

The *picture* data type holds color graphics with a size limited only by memory. The space each picture consumes depends on the size and resolution of the image. The internal storage of a picture is either in native format (Windows bitmap or DIB or metafile or Mac PICT) or in Omnis shared color format. Server databases store picture data as binary objects (BLOBs).

List

The *list* is a structured data type that can hold multiple columns and rows of data. A list can hold an unlimited number of lines and can have up to 400 columns. When you create a list variable you set the type of each column. The data type of each column in your list can be any one of the other data types including Character, Number, Date, Picture, and List: Yes, you can even have lists within lists!

Omnis makes use of the list data type in many different kinds of programming tasks. Normally you would create a variable with list data type and build your list in memory from your server data or Omnis data file. Then you could use your list data as the basis for a grid or list field on a window, or you could use it to generate a report.

You can store lists in Omnis data files directly. To store a list in a SQL table on a server, you can map it to a binary field of some kind.

Row

The *row* is a structured data type, like a list, that can hold multiple columns and types of data, but has one row only: it is essentially a list type with a single row. A row can have up to 400 columns. When you create and define a row variable, you set the type of each column. As with lists, the data type of each column in your row can be any one of the other data types including Character, Number, Date, Picture, List, and Row.

Object

Object classes let you define your own structured data objects. Their structure, behavior, and the type of data they can hold is defined in the variables and methods that you add to the object class. A variable with *object* type is a variable based on an object class: the subtype of the variable is the name of an object class. For example, you can create an instance variable of Object type that contains the class and instance variables defined in an object class.

Object instances created from an object class (via subtype) belong to the current task at the point of their creation; this provides consistency with object instances created via $new.

When you reference a variable based on an object class you create an instance of that object class. You can call its methods with the notation VarName.$MethodName(). For an object variable the initial value contains the parameters which are passed to $construct() for the class when the instance is constructed. The instance lasts for as long as the variable exists.

You can store object instances in a list. Each line of the list will have its own instance of the object class. You can store object variables, and hence their values, in an Omnis data file or server database which can store binary values. If an object variable is stored in a data file the value of all its instance variables are stored in binary form. When the data is read back into Omnis the instance is rebuilt with the same instance variable values.

Object reference

The Object reference data type provides non-persistent objects that you can allocate and free using notation. Non-persistent means that objects used in this way cannot be stored on disk, and restored for use later.

You can use the Object reference data type with local, instance, class and task variables. It has no subtype. To create a new Object instance, referenced by an Object reference variable, you use the methods $newref() and $newstatementref(). These are analogous to the $new() and $newstatement() methods, and they can be used wherever $new() and $newstatement() can be used.

Object references are deleted automatically when they are no longer required in order to free up memory. Object references are deleted when a variable or list column no longer contains the reference. Therefore calls to $deleteref are not required unless you want to release memory sooner than would otherwise occur under the automatic process. Object reference variables are no longer valid after the task that created the variable closes.

Binary

The *binary* type can store structured data of unlimited length up to your maximum available memory. Omnis does not know anything about the format and structure of the data in a binary column or field. In this type of column or field you could place, for example, desktop publishing files, MIDI system exclusive files, CAD files, and so on. You could store the definition of an Omnis class in a binary field.

Binary data corresponds to binary large objects (BLOBs) on most database servers.

Item Reference

You can use a variable of type *Item reference* to store an alias or reference to an object in Omnis or in your library. You assign the notation for the object to the item reference variable using the *Set reference* command. You can use an item reference variable in calculations or expressions which saves you having to quote the full path to the object. You can also use an item reference variable with the *Do* command to return a reference to the object or instance created by the command.

Field reference

You can pass a reference to a field using the *field reference* data type, available for parameter variables only. A parameter variable with the field reference type must have a valid field in the calling method. Once the field reference parameter variable is set up, a reference to the parameter is the same as using the field whose name is passed.

Nulls and empty values

A variable or column of any data type can be NULL. This means the value is unknown or irrelevant, and that there is therefore no way to operate on the column value. A null value is distinguishable from an empty value, which represents empty or uninitialized data.

When defining a file class, you can specify that a field **Can Be Null** or **Cannot Be Null**. This controls the handling of rows written to Omnis data files only and is irrelevant for client/server data, since it doesn't prevent fields from getting null values in Omnis calculations. Null data from a SQL database corresponds to null values in Omnis fields and variables, and null values are sent to a server database as SQL nulls.

You can use the hash variable #NULL to represent null values in calculations. For example, to set a variable to null:

```
Calculate LV_Variable as #NULL
```

The result of arithmetic, comparison, and logical operators on null data is always null. With string functions such as con() and jst(), however, Omnis translates null to empty. The *isnull*() function returns kTrue if the value is null and kFalse if not.

When you use an Omnis sort on columns or variables with nulls, Omnis sorts the nulls first and separately from the empty values (or, for a descending sort, last). In a sorted report the nulls come first and do generate a break.

When exporting records in a text format, null values export as an unquoted string NULL, unless a particular format doesn't support nulls. In this case, Omnis translates the null to empty. Occurrences of this unquoted string in an import file import as nulls.

Formatting strings and input masks

You can further structure the display of Character, Number, Date, and Boolean data using Masked Entry (window class) fields and the $formatmode and $formatstring properties.

Current Record Buffer

The Current Record Buffer, or CRB, is an area of RAM, that Omnis uses to hold your current data. For example, if you are accessing a number of file classes or a SQL view, the CRB holds the current record or data for those files or view.

Schema Classes

A *schema class* is a type of data class that represents a table or view on your server database. A schema class contains the name of the server table or view on your server, and a list of column names and Omnis data types that map directly to the columns in your server table or view. The Omnis data types defined in a schema class should map to the equivalent server types, and the column names must conform to any conventions about case used by the server. For example, if the server column names are case sensitive, the column names in your schema class must be in the correct case.

Schema classes do not contain methods, and you cannot create instances of them. You can however use a schema 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.

To create a new schema class

- Select your library in the Studio Browser
- Click on the New Class option, then click on the Schema option
- Name the new class and press Return
- To edit the class, double-click the class in the Studio Browser

The schema editor lets you enter the name of the server table or view and the column definitions You can move from column to column in the editor either using the Tab key, by clicking in the column, or with the keyboard Up and Down arrows.

Having created a schema (or query) class, you can use the SQL Form Wizard to create a SQL form based on the class to view and enter data into your server database.

Creating a schema class automatically

You can create a new schema class from scratch, as described above, or you can create one based on a server table on your SQL database. To do this:

- Open a database session in the SQL Browser and navigate to your server tables.
- Drag the server table from the SQL Browser and drop it onto your library in the Studio Browser.

This process creates a schema class that maps to your server table (or view) automatically, and ensures that the data classes in your Omnis library map to the data on your server exactly. You can then use the SQL Form Wizard (described in the UI wizard section) to create a form based on the automatically generated schema class. This is covered in the Tutorial in the Making a Schema section.

Query Classes

A *query class* is a type of data class that lets 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.

Query classes do not contain methods, and you cannot create instances of them. You can however use a 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.

To create a new query class

- Select your library in the Studio Browser
- Click on the New Class option, then click on the Query option
- Name the new class and press Return
- To edit the class, double-click the class in the Studio Browser
- Enter the names of the schema classes or schema columns

When you open the query class editor the Catalog pops up 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.

Table Classes

A *table class* provides the interface to the data modeled by a schema or query class. When you create a list based on a schema or query class, a table instance is created automatically which contains the default SQL methods. You should only need to create a table class when you want to override the default methods in a table instance, or you want to add methods to a table. A table class contains the name of the schema or query class it uses, and your own custom methods that override or add to the default table instance methods.

To create a new table class

- Select your library in the Studio Browser
- Click on the New Class option, then click on the Table option
- Name the new class and double-click on it to edit it

You add methods to a table class in the Method Editor, and change its properties in the Property Manager. To associate a table class with a schema or query class, you need to set its $sqlclassname property to the name of a schema or query class.

Like Schema and Table classes, you can use a table class as the definition for an Omnis list using the $definefromsqlclass() method, which lets you process your server data using the methods you added to the table class. See the *Omnis Programming* manual for further details about the SQL lists and their methods.

Exporting Libraries to JSON

The ability to export and import Omnis libraries in JSON format allows you to use a third-party version control system, such as GIT or SVN, in order to manage Omnis application (library) source code. This will allow efficient and secure application development in a team of Omnis developers, as well as the sharing of Omnis libraries and third-party tools among members of the Omnis community: we have a repository on GitHub containing many example libraries: https://github.com/OmnisStudio

If you are using GIT to store your JSON files, all 'end of line' (eol) characters in .omh files need to be converted to 'carriage return linefeeds' (crlf), to avoid errors when importing the files. Therefore, you should create a .gitattributes file in the root of your GIT repo, and add the following line to it, to configure the line feeds:

```
*.omh text eol=crlf
```

You can *export* an Omnis library to a directory tree containing several text files in **JSON format** representing your library. Additionally, you can *import* an Omnis library from a JSON tree previously exported from Omnis. Exporting and Importing libraries is done in the Studio Browser, but there are several methods you can use to create your own library export and import tools.

Exporting Libraries

To export a library to JSON, you need to select the library under the main **Libraries** option in the Studio Browser. After selecting the library, the **Export Lib to JSON** option will be visible in the library options, allowing you to export the library to JSON. (After you have exported a library to JSON, the Update and Rebuild options will appear in the Studio Browser.)

If you have multiple libraries open in the Studio Browser, the Export, Update and Rebuild options will apply to the *currently selected library*. In addition, different libraries will be exported to different JSON trees, under the export folder.

Export Lib to JSON

The **Export Lib to JSON** option exports the currently selected library to a new JSON tree. The location of the export folder defaults to 'exports' in the main Omnis tree, and the export process automatically creates and names a sub-directory in the export folder using the name of your library.

You can export multiple libraries to JSON by selecting the **Libraries** node in the Studio Browser tree and selecting one or more libraries in the *Library pane* (on the right). The 'Export to JSON' option will appear allowing you to export the selected library/libraries.

The old Web Client plugin or iOS client remote forms are not exported (or imported). In addition, #PASSWORDS or the old system table classes such as #MAWFONTS are not exported (or imported).

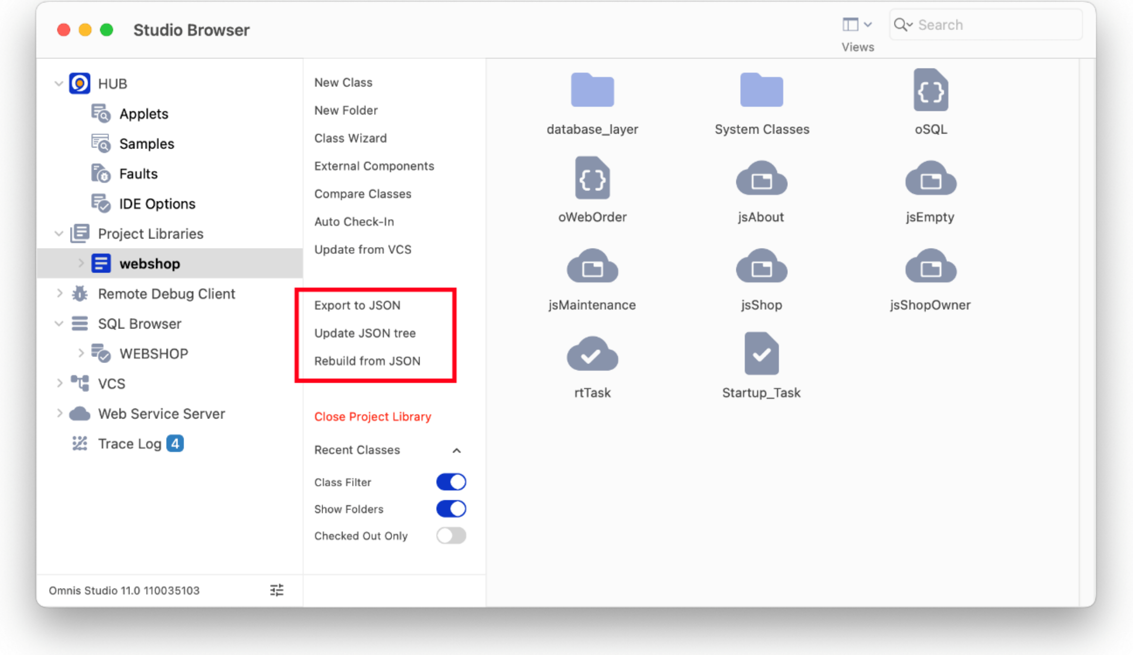
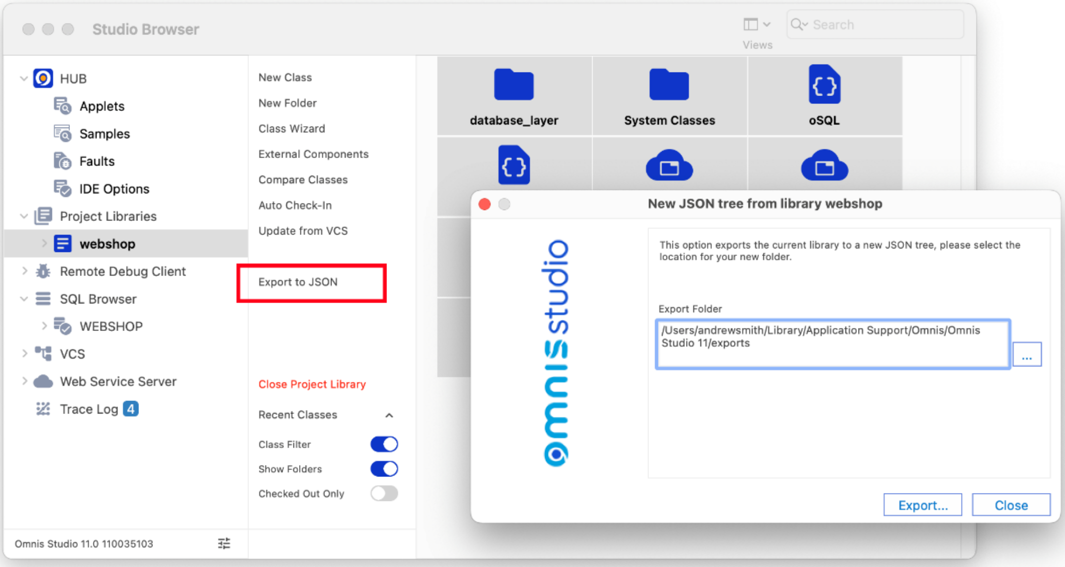


Figure 70:



New JSON tree from library webshop

This option exports the current library to a new JSON tree, please select the location for your new folder.

Export Folder

/Users/andrewsmith/Library/Application Support/Omnis/Omnis Studio 11/exports

Figure 71:

Update JSON tree

The **Update JSON tree** option exports the library to its associated JSON tree, which in effect will update any classes or methods that have changed, or add any new classes in your library. You should be aware that the update process *deletes the existing JSON tree*, and replaces it with a completely new JSON tree built from the updated library.

The update process first checks for any conflicts and reports these if any are found. For example, Omnis will report an error if a JSON file or folder is missing or has been renamed. You need to rectify these errors before you can update, or you can ignore the conflicts in the error log window and proceed with the update.

The conflict detection process uses the modify date of each file in the JSON tree for the class, so if a date has changed a conflict will still be reported even if the file contents have not changed.

Rebuild from JSON

The **Rebuild from JSON** option archives the current library open in the Studio Browser to the 'archives' folder and replaces it with a new library built from the associated JSON tree.

Each time you use the Rebuild option, Omnis places a new copy of the current library in the archives folder and appends a number to the name of the library. The last version of the library in the archive folder is then used during the restore process as the most recent archive.

Once the Rebuild option has been run, the Restore Library option appears.

Restore Library

The **Restore Library** option overwrites the current library in the Studio Browser with the previously archived version.

Library and JSON mapping

The Studio Browser maintains a log of which library maps to which JSON folder, which is essential when working with multiple libraries. A file called 'exports.json' is created in the 'studio' folder that contains the mapping for all your exported libraries, so for each library there is a record of the name and path of the Omnis library file, the name and path of its associated JSON folder, and the path of the archived library, if it exists: note the name of the most recent archive library is used.

Importing Libraries

Note the options to import or export a library from or to JSON are not available in some editions.

You can import a library into the Studio Browser from an existing JSON tree that was previously exported from Omnis Studio using the **Export Lib to JSON** option. For example, you could check out an Omnis JSON tree from a third-party VCS, such as GIT or SVN, and import it into Omnis to start a new project. For example, you can get sample Omnis libraries in JSON format from our GitHub repository at: https://github.com/OmnisStudio

Note you cannot open a library from a JSON tree using the standard Open Library option in the Studio Browser (which can only open a .LBS file). You have to import a JSON tree first to create the library before it can be opened in the Studio Browser.

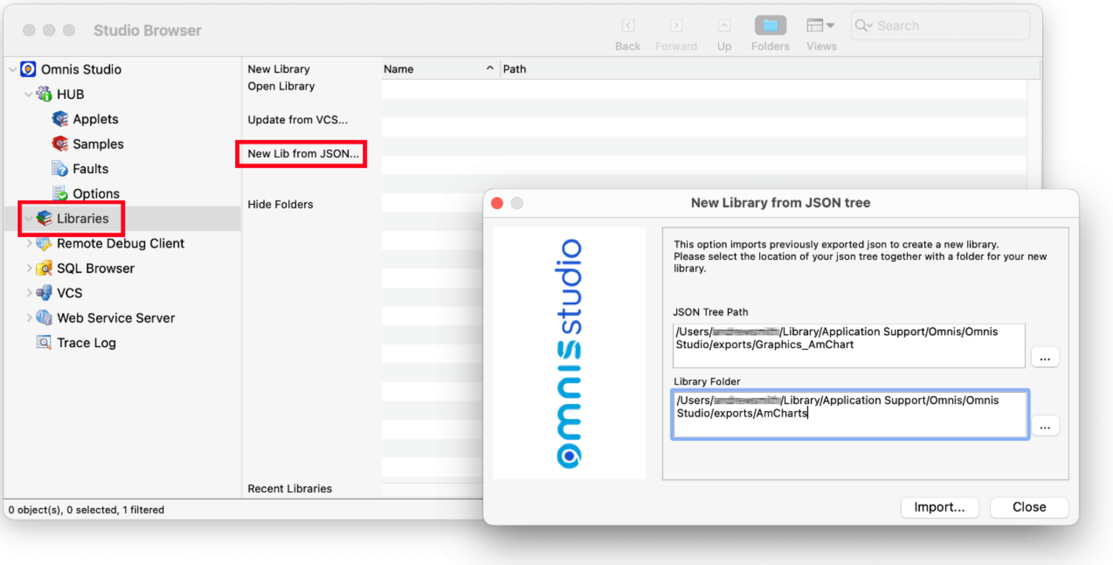
New Library from JSON

To import a library from a JSON tree, you need to select the **Libraries** node in the Studio Browser and click on the **New Lib from JSON** option.

The **New Lib from JSON** option imports a JSON tree that was previously exported from Omnis and creates a new Omnis library file (.LBS); the import folder must already exist. When you have imported and created the new library, its classes will appear in the Studio Browser.

Directory and JSON File Structure

The following sections describe the JSON file & folder structure of a library exported from Omnis Studio using the **Export Lib to JSON** option, which may help you understand how the exported JSON could be managed. Note that all text files exported from Omnis use UTF-8 encoding, including the .json and .omh files, and are formatted suitable for viewing in a text editor.



This option imports previously exported json to create a new library. Please select the location of your json tree together with a folder for your new library.

JSON Tree Path: /Users/[redacted]/Library/Application Support/Omnis/Omnis Studio/exports/Graphics_AmChart

Library Folder: /Users/[redacted]/Library/Application Support/Omnis/Omnis Studio/exports/AmCharts

Figure 72:

Libraries

An Omnis library is represented by a folder that contains the file called 'library.json': this folder has the same name as the library and is referred to as the 'library folder'. library.json contains top-level information about the library, such as the library preferences and version number.

Within the library folder, there is a tree of class directories that represents the folder structure of the Omnis library. Each class in your library has its own directory, and if the class itself is an Omnis folder class, it contains sub-directories for the Omnis classes contained in that Omnis folder.

Classes

Each class directory has the same name as the class name (see the note on directory and file naming below). Every class directory contains a JSON file named 'class.json'. This contains top-level information about the class, including:

- Class type
- Class properties
- For classes that support methods: definitions of class and instance variables, and for task and remote task classes, definitions of task variables.

File classes also have a file called 'indexes.json' within the class directory, if the file class defines any indexes.

Methods

If the class supports methods, the class directory also contains a JSON file named 'methods.json' provided that there are some class methods. methods.json contains an array of the class methods, where each entry contains various properties of the method and definitions for parameters and local variables.

There is a file in the class directory for each method defined in methods.json, named <method name>.omh (subject to the file naming rules below), that contains the method code. The '.omh' file extension is proprietary to Omnis, but the file format is text like the other files.

LF (linefeed) characters in code are exported as Unicode private-use character 0x2fffe, to reduce issues with other tools (note CR characters are also mapped to 0x2ffff).

Objects

If the class can contain objects, then there are two different structures depending on the class type:

- For file, query, schema and search classes, all objects and their properties etc. are in a single file called 'objs.json' in the class directory. objs.json contains an array of objects.

- For all other class types that can have objects, the class directory can have a number of sub-directories:

- objs

- bobjs

- inheritedobjs

The ‘objs’ directory contains a sub-directory for each object in the class, where the directory name is the object name (subject to the directory naming rules below). Each object sub-directory contains a file named ‘object.json’ that contains object properties etc, and if the object has methods, there is an identical structure to that used for the class methods: a methods.json file, and <method name>.omh files.

The ‘bobjs’ directory is only present for window classes (JavaScript forms do not have background objects). It contains a sub-directory for each background object in the class, named using the object ident (subject to the directory naming rules below as older libraries can unfortunately contain objects with duplicate idents). Each background object sub-directory contains a file named object.json that contains object properties, etc.

The ‘inheritedobjs’ directory is only present for classes that support inheritance. It contains a sub-directory for each superclass object that either defines or overrides a method in the subclass. Each sub-directory contains methods.json and <method name>.omh files just like those used for class and object methods, representing the methods defined or overridden for the object.

Binary Data

There are various properties which require a binary representation in the JSON library representation. These are handled in two ways:

- If Omnis recognises a PNG, e.g. in #ICONS or a report background picture, it outputs a PNG file to the tree, and the JSON contains the name of the PNG file.
- Otherwise, Omnis outputs the BASE 64 encoding of the binary data to the JSON file.

Directory and File Naming

Where possible, directories and files are named using the Omnis name (class name, object name, object ident, or method name). However, there are some considerations:

- Although it is not recommended for naming objects in Omnis, class and object names can contain characters that are not allowed in file system names, e.g. path separators for all platforms, ?, *. To cater for this, the JSON library representation escapes these characters as % followed by the 2 lower case hex characters that represent the escaped character. As a consequence, Omnis also escapes the % character.
- Omnis libraries can contain classes where the names only differ by their case. In addition, they can contain objects with duplicate names. In these cases, the JSON library representation prefixes the name with the string %_<n>_ where <n> is an integer index (for objects this is the order value, and for classes this is a value starting at 1 and incremented for each class with the same case-insensitive name: note that Omnis always exports classes in ascending name order, meaning that the prefix for each class in a set of classes with the same case-insensitive name will be the same each time you export the classes, unless you add or remove a class with the same case-insensitive name).

Library Dependencies

Libraries can depend on other libraries. In many cases, the presence of the external library is not required for Omnis to successfully import or export the JSON library representation. However, there are three cases that affect tokenization, and as a consequence, mean the external library or libraries must be open when exporting or importing a library:

- Design task. If the design task is in an external library, the external library must be open.
- Superclass. If the superclass is in an external library, the external library must be open.
- External file classes. If the code or tokenized properties use a variable in a file class in an external library, the external library must be open.

The export option detects the required external libraries in cases 1-3 above while it generates the JSON library representation. It adds an error to the error list when it encounters a reference to an external library that is not open, and returns kFalse. In addition, if the export succeeds, it adds an array to library.json named “includes”: this is an array of all required external libraries. The import library option will fail if any of the included libraries are not open.

External File classes & Tokenization

By default, Omnis tokenizes variables in external file classes using the file name and a field token. For development, you should use both file and field names (to avoid untokenization issues when the external library is not open), whereas for deployment it might be more desirable for performance to use both file and field tokens.

In Studio 8.0, the only control over these tokenization options is via the browser context menu Retokenize… option. For Studio 8.1, there are some new root preferences that you can use to control this:

- $tokenizeexternalfilenames: If true, Omnis uses tokens rather than text when tokenizing external file names
- $tokenizeexternalfieldnames: If true, Omnis uses tokens rather than text when tokenizing external field names

You can use these preferences when importing a library to control how the output library tokenizes variables in external file classes. The values of these preferences are stored in the "defaults" entry in config.json.

Export & Import JSON Notation

$exportjson

The $exportjson method can be used to export a library to a JSON tree.

- $root.$exportjson(rLib, cOutPath [,&lErrList, &lWarningList])
  exports a JSON tree for the library and returns kTrue for success, or kFalse for failure. Parameters:
  rLib is an item reference to the library to export.
  cOutPath is the pathname of the directory into which $exportjson will generate the JSON for the library, or the pathname of a directory which already contains a previous JSON representation of library, and which $exportjson will update to reflect the current contents of the library.
  lErrorList and **lWarningList** are lists that receive errors and warnings about the export process. $exportjson defines these lists, so there is no need to define or clear the parameters before calling $exportjson. See Error Messages

In the case of failure, lErrorList contains error reports, and subject to the Omnis preferences discussed later $exportjson may have cleaned up by removing any partially output JSON library representation before it returned.

In the case of success, lErrorList is empty. lWarningList may contain various warnings about the export process, such as duplicate object idents or object names.

In addition, certain errors or warnings contain a note that there is an entry in the Find and Replace log, which allows you to identify the problem. Errors in the Find and Replace log are drawn in the theme red color.

The error list and warning list each contain 3 columns:

- class: Item reference to the class for which the error or warning is being reported.
- errorcode: Unique integer error code for the error or warning.
- errortext: Error text corresponding to the errorcode.

The $exportjson method displays a working message if it executes for more than a second, allowing you to cancel the export if required, in which case $exportjson returns kFalse and adds error 23433 to the error list.

If $exportjson is being used to overwrite an existing JSON library tree, and an error occurs during the export of a class, Omnis will mark the class.json file for the class in error by adding an "error" entry. This allows a subsequent export to recognise classes exported with an error, and automatically attempt to replace their content.

$importjson

The $importjson method allows you to import a JSON tree previously exported with $exportjson; the method creates an Omnis library file at the specified path.

- $root.$importjson(cJsonFolder, cLibPath [,&lErrorList, &lWarningList , bReplaceExisting])
  imports a JSON library representation tree, and returns kTrue for success, and kFalse for failure. Parameters:
  cJsonFolder The pathname of the JSON library representation directory. $importjson validates this by checking for the presence of library.json in this directory.
  cLibPath The pathname of the new library file to be created from the input JSON library representation. This file must not already exist.
  lErrorList and **lWarningList** are lists that receive errors and warnings about the import process. $importjson defines these lists, so there is no need to define or clear the parameters before calling $importjson. See Error Messages
  bReplaceExisting allows you to replace an existing library, backing up the existing library; see below

In the case of failure, lErrorList contains error reports, and $importjson has deleted a partially created output library.

In the case of success, lErrorList is empty. lWarningList may contain various warnings about the import process, such as duplicate object idents or object names.

The error list and warning list each contain 4 columns:

- pathname: The pathname of the file containing the problem.
- errorcode: Unique integer error code for the error or warning.
- errortext: Error text corresponding to the errorcode. If a method line cannot be parsed, the text causing the error is included in the error message.
- lineno: For some errors, the line number (in the file with the specified pathname) where the error occurred.

The $importjson method displays a working message if it executes for more than a second, allowing you to cancel the import if required, in which case $importjson returns kFalse and adds error 23433 to the error list.

Note that $importjson ignores classes where the folder name ends in .conflict<n> (see later).

If **bReplaceExisting** is true (the default is kFalse), $importjson() closes the library if it is already open, backs it up to the import archives folder, and imports the library to replace the backed up version; if the import fails, $importjson() restores the original library from the archived copy. Omnis keeps the last 10 archived copies.

The archived copies are stored by default in the archives folder in the Omnis data folder. Each library has its own sub-folder in archives, named using the library name. You can override the archive folder by setting the **archivefolder** member of $prefs.$exportimportjsonoptions Omnis preference.

$comparejson

The $comparejson method checks the specified library against the JSON library tree stored at cOutPath, and returns wChangesRow to indicate what will occur when $exportjson runs. The method behaves identically to $exportjson, except that it builds wChangesRow.

- $root.$comparejson(rLib, cOutPath, &wChangesRow[, &lErrList, &lWarningList]) checks the specified library against the JSON library tree; returns kTrue for success.

For success (kTrue returned), wChangesRow has 4 columns, and each column is a 2 column list, where column 1 is the class name and column 2 is the path to the class representation in the JSON library tree. The columns in wChangesRow are:

- **new** Each entry in the list identifies a new class, that is, a class that will be exported because it is in the library, but not the JSON library tree.
- **delete** Each entry in the list identifies a class that will be deleted from the JSON library tree, because either it is no longer present in the library or it has been moved to a different folder in the library. NOTE: If a folder is to be deleted, there will not be delete entries for its content (which is implicitly deleted).
- **replace** Each entry in the list identifies a class which has changed in the library since the last JSON import or export of the class.
- **conflict** Each entry in the list identifies a class which has possibly been changed in the library, but more importantly, the copy of the class in the JSON library tree has *definitely changed* since the last JSON import or export of the class.

The $comparejson method provides you with some information before overwriting the contents of an existing JSON library tree.

There are two conditions which will cause Omnis to arbitrarily overwrite a class during export:

- The method text file extension has been changed (see the section on Omnis preferences).
- The copy of the class in the JSON library tree is marked to indicate that an error occurred during a previous export.

Export & Import JSON Preferences

There is a root preference $exportimportjsonoptions (a row containing parameters) that affects the behavior of $exportjson(), $importjson(), and $comparejson() (the settings are also stored in the ‘exportimportjsonoptions’ group in config.json). The parameters are:

- **hideexportworkingmessage**
  Boolean (default false). If true, the working message is hidden for $exportjson and $comparejson.
- **hideimportworkingmessage**
  Boolean (default false). If true, the working message is hidden for $importjson.
- **deleteexportoutputtreeifcancelled**
  Boolean (default true). If true, $exportjson deletes a partially exported output tree, if the export is cancelled by the user.
- **exportoverwritesconflicts**
  Boolean (default false). If true, $exportjson replaces conflicts when overwriting an existing tree (conflicts are ignored when the method text file extension has changed, or class.json contains the error marker - in other words, in these cases, the class is always replaced). If false, $exportjson does not replace the folder for a conflicting class, and before it exports the class, renames the folder for the class to be of the form name.conflict<n> where <n> is an optional integer added if the .conflict folder already exists from a previous export; this makes both the original JSON and the new JSON available in the library tree.
- **importtreatsunknownpropertyaswarning**
  specifies whether or not unknown properties in imported JSON are treated as a warning; it is true by default.
- **exportcodefoldingstate**
  controls whether or not the code-folding state in the methods in your library is exported; the option is set to false by default so the code folding state is not exported.
- **includenotationinide**
  Boolean (Default false). Specifies whether the notation methods $comparejson(), $exportjson() and $importjson() are present in the IDE property manager tables. This also controls whether these notation methods appear in the code assistant.
- **fullexportimport**
  Boolean (Default true). When true, all library information is included in the JSON export
  If set to false, Omnis does not export certain information which is not required to represent the library, including ‘internalversion’, ‘omnisbuild’ and ‘moddate’. A library exported with fullexportimport set to false can only be imported if fullexportimport is set to false.

Error Messages

The following error messages may be returned by the export & import JSON methods.

Error Code	Error Text ($ is replaced at runtime)
23428	Property in input JSON is only allowed in custom styles:$
23429	Bad fieldname in object $
23430	Import requires external library to be open: $
23431	Export requires external library to be open:
23432	JSON import and export must be executed on the main thread
23433	Operation cancelled by user
23434	Invalid class type:
23435	Image file is not a PNG
23436	Error reading PNG file: $
23437	Error opening PNG file: $
23438	Error reading method text file
23439	Error opening method text file
23440	Internal error: invalid path:
23441	Superclass is missing:
23442	Error creating class:
23443	External class must be available before the library can be imported:
23444	File does not exist
23445	Error creating output library:

Error Code	Error Text ($ is replaced at runtime)
23446	Output library path is invalid
23447	Output library already exists
23448	library.json does not exist in the input JSON folder
23449	Error creating directory:
23450	Error writing output file:
23451	Cannot export class because it is protected
23452	Cannot export class because its superclass '$' is not available
23453	Cannot export class because its design task '$' is not available
23454	Added external library dependency for library that masks the export library $defaultname:
23455	Parameter 2 of $exportjson() must be the pathname of an existing folder
23456	Parameter 1 of $exportjson() must be an item reference to a library
23457	See find and replace log:
23458	Object number $ (ident $) has a duplicate name: $
23459	Input JSON folder does not exist
23460	Object number $ has a duplicate ident: $
23461	Internal error: property definition does not contain data source for property:$
23462	Internal error: Invalid field number when accessing fields CRB:$
23463	Internal error: Unable to access CRB for object:$
23464	Internal error: no CRB for property:$
23465	Internal error: invalid property data type:$
23466	Cannot export or import custom style because the property cannot be found:$
23467	Internal error: cannot extend field list
23468	Internal error: Error extracting multi-value property when generating JSON property definition
23469	Internal error: cannot get list for multi-value properties
23470	See find and replace log: Invalid method line - tokenizing and untokenizing the method line does not result in the same text:
23471	Error setting up new imported object:
23472	Cannot export or import custom style because the object cannot be found:$
23473	Invalid property: $
23474	Internal error: Unknown property in library information property definitions:$
23475	Internal error: Bad format table id:$
23476	Internal error: Invalid attribute passed to getExporterValue() or setImporterValue():$
23477	Internal error: Cannot create temporary complex grid object
23478	Internal error: Invalid external component set property:$
23479	Cannot obtain object definition for field: $
23480	Cannot create output directory: $
23481	Invalid user tool type value encountered when exporting toolbar object:$
23483	Internal error:External component has no data block:$
23484	Internal error:Invalid row:$
23485	Error parsing responsive position information:
23486	Cannot export remote form $. Only JavaScript client remote forms can be exported to JSON
23487	Imported object with a duplicate ident: $
23488	Imported object with a duplicate name: $
23489	Discarded inline comment ($) as Omnis cannot tokenize Sta: command with inline comment appended using libConverterAddsInlineCommentToStaCommandParameter from config.json
23490	Member missing:
23491	Row has no content:
23492	Cannot obtain row:
23493	Unknown property in input JSON:$
23494	Cannot import table from JSON:$
23495	Invalid type for property $ (expected Boolean)
23496	Invalid non-primitive data for property $
23497	Object has invalid member count:
23498	Invalid type (expected Character):
23499	Invalid constant for $
23500	Invalid date for $
23501	Invalid type for $ (expected Integer)
23502	Value is too long for property $
23503	Invalid value for property $
23504	Integer value of property $ is outside valid range
23505	Invalid constant:
23506	Class type in JSON does not match destination class type

Error Code	Error Text ($ is replaced at runtime)
23507	Error creating temporary class: $
23508	Error getting imported class data: $
23509	Error setting imported class data: $
23510	Error creating record definition or list: $
23511	Internal error:cannot obtain member name for record definition for file $
23512	Column of array is not defined as expected:
23513	Invalid or missing integer value:
23514	Invalid or missing constant value: $
23515	Invalid record definition entry:
23516	Missing parameter number for parameter:
23517	Object name is empty
23518	Non-primitive value must be stored as an object:
23519	Unable to allocate method number for imported method $
23520	Error setting method name:
23521	Error setting operation id:
23522	Error setting method text:
23523	Error storing method in class:
23524	Duplicate object name:
23525	Invalid or missing Boolean value:
23526	Could not add file class object:
23527	User constants or file class entry already used when importing:
23528	Could not add file class index:
23529	Could not set file connection:
23530	Could not find index part name:
23531	Could not find object type:
23532	Could not find section type:
23533	Invalid object type:
23534	Invalid section type:
23535	Invalid screen size:
23536	Invalid layout breakpoint:
23537	Unable to resolve search object name:
23538	Error tokenizing calculation:
23539	Invalid or missing calculation value:
23540	Invalid order value for object
23541	Error saving system table: $
23542	Invalid number of hotspots:
23543	Custom property entry count does not match custom property value count:
23544	Invalid or missing character value:
23545	Cannot find field name:
23546	Error setting page setup item:
23547	Could not locate constant value for property:
23548	Value not stored as an object:
23549	Cannot find column: $
23550	Invalid events:
23551	Object does not support events:
23552	Value out of range:
23553	Invalid property value:
23554	Custom property name does not match:
23555	Error setting shortcut value:
23556	Cannot resolve parent folder name:
23557	Error importing date:

Chapter 3—Omnis Programming

Omnis Studio has a powerful programming environment that lets you create almost any type of enterprise or web application. The Omnis programming environment contains hundreds of 4GL commands and functions, as well as a low-level scripting language, called *Omnis Notation*, that allows you to manipulate objects dynamically in a web browser, on a phone, or an end user's desktop computer. To program in Omnis, you must consider the following things:

- **Variables**
  variables are the principal data container in Omnis: most objects in Omnis can contain variables, but their *scope* and the kind of *data* they can contain depends on the *type* of variable: you can use the method editor to add variables to an object: the definition and dynamic manipulation of variables is at the heart of programming in Omnis Studio.
- **Methods**
  methods are pieces of Omnis code contained within the objects in your application, each performing a particular operation specific to the object or the application as a whole: creating and modifying methods in your library is key to creating the UI and business logic in your application.
- **Events**
  almost all user actions in Omnis generate an *event*: when an event occurs a *message* is sent to the object in which the event occurred: you can write methods behind the objects in your library to handle the events.
- **Tasks**
  when your application runs in Omnis many object instances are created, such as web forms, reports, and data objects: these instances are opened and handled within a *task*. Omnis creates a default task for web & desktop apps, but you can create your own tasks that allow you to handle the objects in your application.

All the above topics are covered in this chapter. You create and modify methods in Omnis using the *Method Editor* and Omnis code is written in the *Code Editor*.

Variables

Variables can hold different types of data and are visible in different parts of your application depending on their data type and scope. For example, if you create a variable of list data type in a remote form, the list variable and hence its data is visible within the remote form and all its instances, but is not accessible elsewhere in your library. The Data Types available in Omnis are described in detail in the *Libraries and Classes* chapter.

Declaration and Scope

A variable may be global, accessible from all parts of your application, or it may have its scope restricted to certain areas so that it cannot be referred to from elsewhere. By declaring variables in the proper scope, you limit the potential for arbitrary connections across your application and thus reduce the potential for error and the complexity of your application.

The following table lists the different kinds of variables and their scope. It also shows when they are initialized and destroyed.

Variable	When Initialized	When Destroyed
Parameter	on calling the method	returning to the calling method
Local	on running the method	on terminating a method
Instance	on opening an instance	on closing the instance
Class	on opening a library	on clearing class variables or closing a library
Task	on opening an instance of the task	on closing the task instance
Hash	on starting Omnis	on quitting Omnis

Apart from hash variables which are permanently built into Omnis, you must create all variables with the appropriate type and scope in the objects in your library using the method editor. After you have declared them, variables that are in scope are listed in the Catalog, or in the Code Assistant in the Method Editor. You can remove a variable using the Delete Variable option in the variable pane context menu. Declared variables are removed from memory when they are destroyed.

Parameter Variables

You can use a parameter variable to receive a value in a method, for example, a value that has been passed to the method using the *Do method*. You would normally do something with the value in the method and possibly return a new value. Parameter variables are visible within the called or recipient method only. They are initialized when the method is called, and cleared when the method returns to its caller.

Local Variables

Local variables are local to the method. You can refer to the variable within that method only. Local variables are initialized when the method begins execution, and are cleared automatically when the method terminates.

Instance Variables

Instance variables are visible to the instance only, that is, all methods and objects in the instance. You can define an instance variable only for classes that can be opened or instantiated, that is, remote forms, remote tasks, tasks, tables, reports, and object classes, as well as External Objects (in desktop apps you can also instantiate window, menu, and toolbar classes). Note that you cannot declare instance variables in code classes. There is a set of the declared instance variables for each instance of a class: these are initialized when the instance is constructed (opened) and cleared when the instance is destructed (closed).

Class Variables

Class variables are visible within the class *and all its instances*. You can declare class variables for tasks, tables, reports, and code classes (in desktop apps you can add class variables to window, menu, and toolbar classes). Any object or method in the class can refer to a class variable, and all instances of the class also have access to the class variable.

Class variables are not automatically cleared from memory. You can remove them from memory by closing the library containing the class, or using the *Clear class variables* command.

Task Variables

Task variables are visible within the task, all its design classes and instances. In practice, you can refer to a task variable from any method within any class or instance that belongs to the task. Omnis initializes task variables when you open the task: so for the Startup_Task this is when the library opens. Note that you cannot declare a task variable for a class until you have set the **$designtaskname** property for the class.

Hash Variables

Omnis has a built-in set of global variables, called *hash variables* since they start with the “#” symbol. You can view them in the Catalog (F9/Cmnd-9). Hash variables are global, unlike any other variables, so all libraries have access to them. The advantage of having global variables is that you can use these variables to pass data between libraries in an application. The disadvantage is that any data you place in hash variables remains there when you switch between libraries or combine libraries, with potentially unpredictable results.

Adding a Variable

You can add variables to a class or object in the *Variable Pane* of the Method Editor. If the variable pane is not visible you can show it using the **View>>Show Variable Panes** menu option on the Method Editor menu bar. Alternatively, you can add a new variable simply by typing its name in a line of code in the Code Editor and declaring the variable in the Create Variable dialog; see below.

The tabs in the Variable Pane let you define Task, Class, Instance, Local and Parameter variables: note that the local and parameter tabs only appear after you have added or selected a method in the method editor. You can add up to 400 variables of each type to the current object, including 400 local and parameter variables for each method in the current object. The name, type, subtype, and initial value of each variable is listed in the variable pane. You can size the columns in the variable pane by sizing the column headers.

You cannot declare a task variable within a class until you have set the $designtaskname property for the class: see the section below on *Adding Task Variables*.

To add a new variable

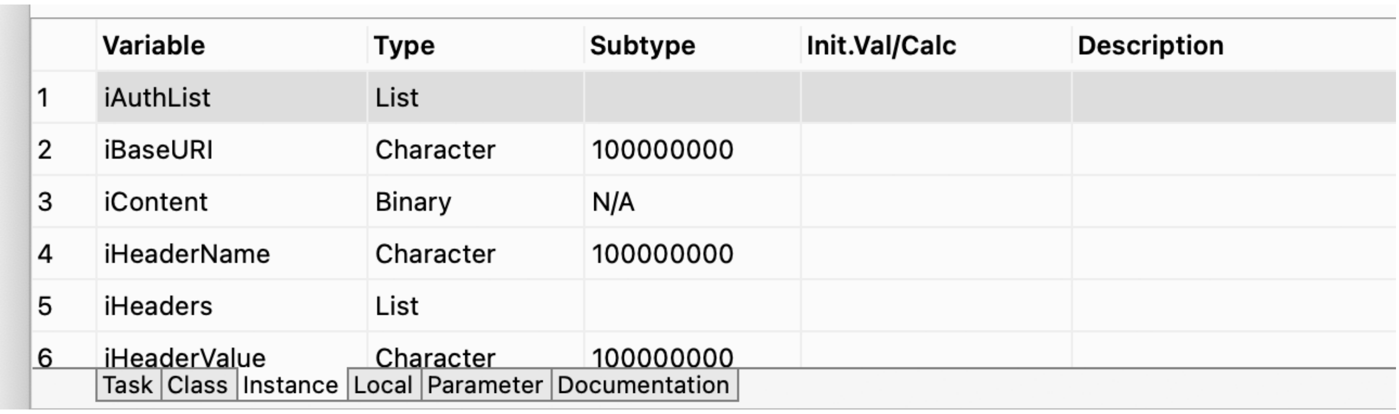
- Open the class in which you want to add the variable
- Right-click on the background of the class and select the **Methods** option to open the Method Editor

The variable pane is at the top of the Method Editor window:

- Choose the tab for the type of variable you require, for example, the Instance tab to create an instance variable
- Click in the ***blank field*** under the Variable column header
- Enter the name of the variable

or

- Right-click in the variable pane to open the variable context menu



	Variable	Type	Subtype	Init.Val/Calc	Description
1	iAuthList	List			
2	iBaseURI	Character	100000000		
3	iContent	Binary	N/A		
4	iHeaderName	Character	100000000		
5	iHeaders	List			
6	iHeaderValue	Character	100000000		

Figure 73:

- Choose **Insert New Variable** and click in the variable name to edit it, or type over the new variable name if it is selected (see Variable naming below)
- Tab to the **Type** box and choose the type from the droplist using the mouse or arrow keys

or when the focus is in the Type box

- Type the first letter(s) of a data type to select it, for example, you can type "nu" to select the Number data type, or "b" for Boolean and "bi" for Binary type

For Number and Date Time variables

- Tab to the **Subtype** box and choose a subtype: again, you can type the first letter(s) of a subtype, for example, for Numbers you can type "L" to select the Long Integer subtype

For Object and Object Reference variables

- Click on the dropdown list in the Subtype box to open the **Select Object** dialog
- Open the **External Objects** node and choose the object for your variable; you can use the Search box to find an object in the list

You can enter an initial value or calculation for all types of variables. The initial value allowed for a variable depends on its type.

Variable Type

As well as scope, the data type you choose for a variable is critical to how it functions in your code and any calculations, The types available include Character, Number, Boolean, Picture, List, Row and Object, which are described in full under the Omnis Data Types section in the *Libraries and Classes* chapter.

Variable Naming

Variable names can be up to 255 characters long, although in practice you should keep them as short but descriptive as possible. When you name a variable you should prefix its name with one or more letters to indicate its scope. For example, parameters variables can begin with the letter "p", local variables "lv" or just the letter "l", instance variable "iv" or just the letter "i", and so on. (You could use the variable prefixes described in the Creating Unrecognized Variables section.)

There are certain limits or restrictions on the characters you can use in a variable name. Each character in a variable name must either be a Unicode alpha character, a decimal digit, or a character in the range U+80 to U+ff inclusive. The first character cannot be a decimal digit. As with class names, you cannot use the following characters in variable names: . $ & ( ) [ ] and #. In addition, do not use spaces in variable names, however you can use _ (underscore) to separate words if necessary.

Duplicate names and Scope

To avoid all ambiguity between variables of different scope, you should not use duplicate names, and use a naming convention similar to the one described above. When two or more types of variable use the same variable name, a reference to that variable could be ambiguous. In a situation where more than one variable of the same name exists, Omnis automatically uses the variable with the smallest scope. Therefore, it is possible, though not good practice or recommended, to have local, class, and task variables called "MYNAME". As Omnis resolves ambiguity, a reference to MYNAME will refer to the local variable if it exists for the current method.

Adding Local and Parameter Variables

Local and parameter variables are inserted into the currently selected method. Therefore, to insert these variables for a particular method, you need to select the method before inserting local and parameter variables.

Parameter variables receive values from a calling method in the order that they appear in the variable pane. You can reorder parameter variables by dragging them into position in the variable pane. To do this, click and drag the fixed-left column or row number for a parameter variable in the list.

Normally you must declare all types of variable, including local variables, in the variable pane before you can use them in your code. However, you can declare a special type of local variable in your code without first declaring it in the method editor. To declare such a variable, prefix the variable name with %% to create a string variable, or prefix the variable name with % for a numeric variable of Floating point type. You can type such variable names directly into your code in the method editor, at which time their names are added to the Local variables pane.

Item Reference Classes

When creating an **Item Reference** variable, the variable subtype dialog (that allows you to enter the class for a variable) has two tabs at the bottom: Generic and Instance:

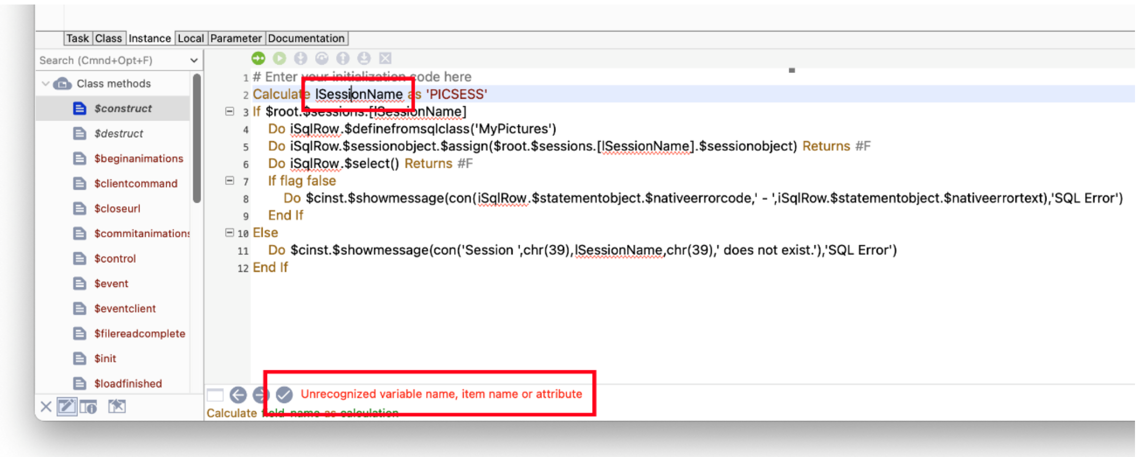
- **Generic** allows you to enter a class.
- **Instance** allows you to enter a class of the form @classname or @library.classname (these classes are identified by a leading @ character).

An item reference that uses a class in this way provides code assistance for both the built-in methods and properties of an instance of the class, as well as user methods for instances of the class.

In addition, there is an entry field that allows additional notation to be appended, e.g. $objs.objname (this field has code assistance; the additional notation must start with a $). For example, if you have a field named *test* in window class *myWindow*, you can enter $objs.test in the entry field, and select the window class from the tree. This results in an item reference class @myWindow.$objs.test. When using the variable, code assistance is for this field, so you get both the built-in methods and properties and the user methods for the field.

Creating Unrecognized Variables

You can add a new variable simply by typing its name in a code line and declaring the variable in the **Create Variable** dialog. When you type the name of the new variable in your code, initially it will not be recognized and is marked as an error (red curly underline by default).



```
# Enter your initialization code here
Calculate lSessionName as 'PICSESS'
If $root.$sessions.[lSessionName]
   Do iSqlRow.$definefromsqlclass('MyPictures')
   Do iSqlRow.$sessionobject.$assign($root.$sessions.[lSessionName].$sessionobject) Returns #F
   Do iSqlRow.$select() Returns #F
   If flag false
      Do $cinst.$showmessage(con(iSqlRow.$statementobject.$nativeerrorcode,' - ',iSqlRow.$statementobject.$nativeerrortext),'SQL Error')
   End If
Else
   Do $cinst.$showmessage(con('Session ',chr(39),lSessionName,chr(39),' does not exist.'),'SQL Error')
End If
```

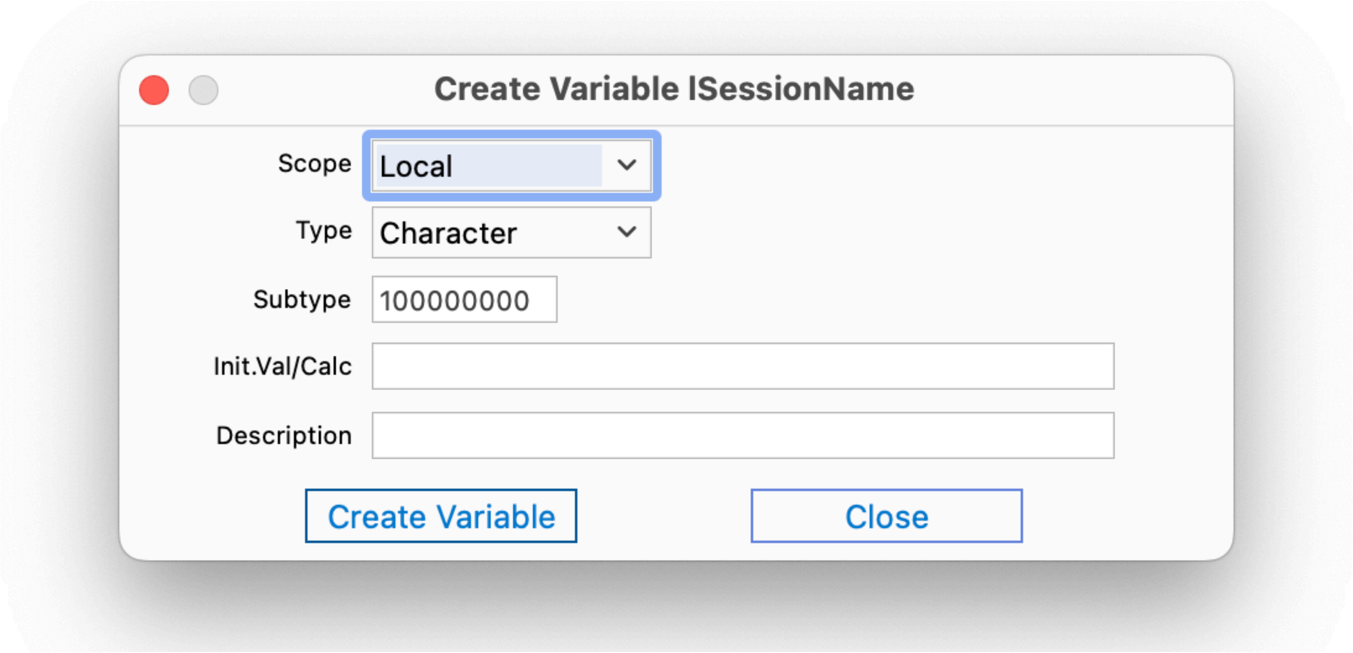
Unrecognized variable name, item name or attribute

Calculate field name as calculation

Figure 74:

You can click on the **Fix** button (at the bottom of the Code Editor window, as above) to open the **Create Variable** dialog, allowing you to declare the new variable, including its scope, data type, subtype, initial value and description (Omnis will try to guess the scope and type based on the current context and variable name; see below about naming).

The unrecognized variable dialog can open when assigning a new or unknown variable name to a property in the Property Manager. In this case, for properties such as $dataname, the initial type of the variable creation dialog is set to the most likely data type for the control, e.g. List data type for a list form control. The dialog restricts the scope of the new variable to what makes sense based on class type, and so on.


Create Variable lSessionName

Scope: Local

Type: Character

Subtype: 100000000

Init.Val/Calc:

Description:

Figure 75:

The Create Variable dialog will open when Omnis encounters an unrecognized variable name, but you can disable this behavior if you set the “canUseCreateVariableOnVarNotFound” setting in the “ide” section of config.json to false (it is true by default).

Create Variable Prefixes

When you type the name of a new variable in your code, you can specify the initial scope for the variable using a predefined *prefix;* in this case, the Create Variable dialog will select the scope automatically. For example, you can begin the variable name with “i” to create an instance variable, or “p” to create a parameter. The default variable prefixes are:

Prefix	Variable scope
i	Instance
c	Class
p	Parameter
l	Local
t	Task

The prefixes allowed in the Create Variable dialog can be configured in the Omnis configuration file (config.json) using the entry called “createVariableScopePrefixes” located in the ‘codeAssistant’ section in config.json:

```
"createVariableScopePrefixes": [
  "i:Instance",
  "c:Class:",
  "p:Parameter",
  "l:Local",
"t:Task"
],
```

The Create Variable dialog processes these entries in array order, and as soon as it finds a scope that is allowed for the method being edited (e.g. instance variables are only allowed for class types that have instances), where the first part of the entry value case insensitively matches the start of the variable name, it uses the configured scope (the second part of the entry value after the colon) to set the initial scope suggested by the dialog. If no prefix match occurs, the scope suggested is local.

Create Variable Suffixes

As well as setting the scope of a variable, using a prefix, you can specify the data type of a variable using one of a set of predefined *suffixes.* For example, you could enter the name “iData**Row**” which would create an instance variable of type Row, or typing “iData**List**” would create a list, and typing “iVarRef” would create an item reference. The default variable suffixes are:

Suffix	Variable type
Row	Row variable (kRow)
List	List variable (kList)
Ref	Item reference variable (kItemref)
Date	Date variable (kDate)
Obj	Object variable (kObject)
Bin	Binary variable (kBinary)

The suffixes allowed in the Create Variable dialog can be configured in the Omnis configuration file (config.json) using the entry called "createVariableTypeSuffixes" located in the 'codeAssistant' section in config.json:

```
"createVariableTypeSuffixes": [
  "Row:kRow",
  "List:kList",
  "Ref:kItemref",
  "Date:kDate",
  "Obj:kObject",
  "Bin:kBinary"
],
```

Omnis strips any consecutive digits from the end of the desired variable name, and then compares (case independently) the end of the resulting name string against the suffixes in the config.json array (strings before the colon in each array entry). If there is a match, and if the variable type is suitable (e.g. it is not a non-client executed type when creating a variable for a client-executed method), then the initial type is set using the type constant after the colon.

Deleting Unused Variables

The context menu of the Variable pane has the option **Delete Unused Variables...**, available by Right-clicking on the variable pane away from a variable line. When selected, it opens a dialog from which you can select variables to delete. The dialog displays the variables of the current type displayed in the variable pane, which are potentially unused. This means the variables could still be in use, for example, they could still be used in subclasses or notation.

Adding Task Variables

To add a task variable for a class you *have* to set its $**designtaskname** property. In most cases, the design task for a class is specified as the Startup_Task by default. You can change it using the Property Manager or the notation. The design task for a class is ignored at runtime.

To set up the design task for a class

- Click on the class in the Browser
- Display the Property Manager or bring it to the top (F6/cmnd-6)
- Click on the droplist in the $**designtaskname** property to view the current tasks

The list of tasks will contain a Startup_Task, and any tasks you may have created.

- Select the design task by clicking on it

You will now be able to define task variables for this class.

Changing the Scope of Variables

You can change the scope of a variable at any time by dragging the variable from one variable pane to another. For example, you can change a class variable into an instance variable by dragging and dropping it onto the instance variable tab. Note you cannot change the scope of task variables.

Variable Initial Values

When you declare a variable in the variable pane of the method editor you can assign it an *initial value*. The first time a variable is referenced in a method, Omnis assigns the specified initial value to the variable. You can set the initial value to be a number, string, calculation, some notation, an Omnis constant, or another variable name. In the latter case, when you first use the variable it gets the value in the other variable, regardless of the order of declaration.

For class variables only, the *Clear class variables* command clears the current values in all class variables and resets them to their initial values.

You can set the initial value of parameter variables, which in effect gives them a default value, but when and if a value is received in the method the initial value is overridden. For example, you may want to assign an initial value of zero to a parameter value to avoid it being null if a value is not received.

For instance variables, the initial value is assigned when the instance is created, e.g. when a form is opened.

Initial Parameter Values

Any parameters that are omitted when you call a method are initialized using their initial value. This is the default behavior for new libraries. The library preference $clib.$prefs.$useoldparameterpassing controls this behavior. If true, an empty parameter that is not the last parameter is initialized to empty or zero, rather than its initial value in the called method parameter definition (this does not apply to client executed client methods in the JavaScript client). The library preference defaults to false in new libraries, and true in converted libraries to maintain backwards compatibility.

Variable Context Menu

You can lookup and edit the value of any variable or constant in Omnis at any time using its context menu. You can Right-click on a variable name wherever it appears in Omnis to open its context menu and view its current value. The Variable context menu displays the variable name, its current value, which group of variables or class it belongs to, and its type and length. You can also perform various debugging functions from this menu as well.

If you select the first option in the Variable context menu, Omnis opens a variable window containing the current contents of the variable which you can edit. Note that you cannot edit binary variables.

Variable Tooltips

You can pass the pointer over a variable or constant and a *variable tooltip* will pop up displaying the variable's current value and description. Variable tooltips are available wherever variable names appear in Omnis including the method editor and Catalog. However, they are not available if Help tips are enabled for the tool containing the variable.

For some variable types, such as list or binary variables, the tip may say "not empty" which tells you the variable has a value, but it is too long to display.

The value of Boolean variables is shown when you hover over the variable. The "Show Empty Booleans" option in the Debugger Options menu in the Code Editor controls whether empty Booleans are shown as Empty or No/False; the default is on, meaning that unset Booleans are shown as empty.

Variable panel

The Variable panel is displayed in the lower-right section of the Method Editor and allows you to view and modify variables while debugging or stepping through your code. This is described in more detail in the Debugging Methods chapter.

Viewing Variables in the Catalog

You can view the variables in your library and the current class using the Catalog (press F9/Cmnd-9 to open it). The Variables pane shows all the Task, Class, and Instance variables for the current class, plus all Local and Parameter variables for the currently selected method. Following the Event Parameters group, the Catalog also lists any file classes in your library. You can enter the name of any variable that appears in the Catalog into your code either by double-clicking on the name in the Catalog (assuming the cursor is at a position that can accept input), or by dragging the variable name out of the Catalog into the method editor.

When you drag a variable from the Catalog, Omnis shows you what type of variable it is and its name. Note that you can also drag variables from the Catalog and drop them onto window and report classes to create a data field for the variable.

You can also drag a variable from the variable pane in the method editor to any calculation or entry field in the command palette. To drag a variable name you need to click and drag the fixed-left column or row number in the variable list.

Auto Fill Variable Option

When you want to enter a variable in the method editor command palette and you can’t remember its full name, you can type the first few characters of the variable, wait a short while for a list to appear, and choose the variable from the list that pops up. The list contains all the variables beginning with the characters you typed. The time it takes for the autofill option to work is set in the $notationhelptimer Omnis preference (the default is 1000 milliseconds).

Custom Variable Types

You can define your own custom variable types. To do this you have to create a custom method called $<customatttribute name>.$assign, and then the $type, $subtype and $sublen properties of the custom variable return their value according to the type of parameter 1 of $<customatttribute name>.$assign.

Comparing Variables

You can do comparisons in the Omnis language between *binary* variables, *object* variables and *object reference* variables, when both sides of the operator are the same type.

Binary comparisons compare the data byte by byte until there is a non-matching byte, in which case the first variable is greater than the second variable if the non-matching byte in the first variable is greater than that in the second variable. The comparison extends to the length of the shortest variable: if all bytes match, then the first variable is greater than the second if it is longer than the second, and vice versa.

Object comparisons compare the object instance – if the instance is the same, the variables are equal.

Methods

Omnis provides a complete *4GL programming language* comprising over 400 commands, each command performing a specific function or operation. In addition, Omnis provides a means to manipulate the objects in your library called the *notation*: this accesses the standard properties and methods contained in the objects in your library. Method are added or updated in the Method Editor.

In previous versions of Omnis, the number of method lines was limited 1024, but this limit has been removed. However, although the number of method lines is theoretically unlimited, the maximum number of method lines is capped at 256,000 to maintain efficiency in your code.

Each method line can contain an Omnis command, or some notation, or often a combination of these: you can also add comments to method lines. For example, to open a window from a menu line you only need one command in your method, that is the *Open window instance* command, which as the name suggests opens an instance of a window. A method that connects you to a server database requires several commands executed in a particular order. You can perform most operations using the notation and the *Do* command. For example, you can open a window using the *Do* command and the $open() method.

For further details about specific commands and notation used throughout this chapter, see the Omnis Help (press F1 to open it), or the *Omnis Reference* manuals. When you start to program methods, you will need to use the debugger which is described in the *Debugging Methods* chapter.

Notation

Omnis structures its objects in an object tree, or hierarchical arrangement of objects and groups that contain other objects. The complete tree contains all the objects in Omnis itself, together with your design libraries, classes, and other objects created at runtime. You can view the complete object tree in the Notation Inspector.

The object at the base of the tree is called $root. The $libs group contains all the current open libraries and lets you access each library and its classes at design time. The classes and objects in each library are stored in their own separate groups: for example, the $remoteforms group contains all the remote form classes in a library. Most of the other groups directly under $root contain the objects created at runtime when you run your application: for example, the $iremoteforms group contains all the remote form instances currently open, or $iremotetasks contains all the remote task instances currently open.

When you want to reference a particular object, a class or instance perhaps, you must access the right branch of the object tree. For example, you must access the $remoteforms group to reference a remote form class. Whereas, to access a remote form instance, say an instance of the same remote form class, you must reference the remote form instance via the $iremoteforms group, which is contained directly under the $root object.

To facilitate a system of naming or referring to an object in the object tree, and its properties and methods, Omnis uses a system called the *notation*. The notation for an object is really the path to the object within the object tree. The full notation for an object is shown in the status bar of the Notation Inspector. You can use the notation to execute a method or to change the properties of an object, and you can use a notation string anywhere you need to reference a variable or field name.

In the notation all property and standard method names begin with a dollar sign "$", and methods are further distinguished from properties by having parentheses after their name. Standard objects and group names also begin with a dollar sign. To write the full notation for an object you need to include each object and group in the path to the object, separating each object using "." a dot. For example, to refer to a remote form class in a library you would use the following notation

```
$root.$libs.LIBRARYNAME.$remoteforms.RemoteFormName
```

This notation includes $root as the base object, the $libs group containing all the open libraries, the name of your library, the $remoteforms group containing all the remote form classes in your library, and lastly the name of the remote form itself. If you want to refer to a particular object in your remote form you need to add the $objs group and the name of the object

```
$root.$libs.LIBRARYNAME.$remoteforms.RemoteFormName.$objs.Objectname
```

In addition, there are a number of shortcuts that let you reference objects, without always referring right back to the $root object, and certain global objects that you can use to make your code more generic. These are described below.

Item References

To save you time and effort, and to make your code more efficient, you can create an alias or reference to an object which you can use in place of the full notation for the object. To do this, you create a variable of type *item reference* and use the *Set reference* command to assign the notation to the variable. The item reference variable can be of any scope, and the notation can be any valid Omnis notation for an object, a group, or even an object property. For example

```
# Declare variable WinRef of type Item reference
Set reference WinRef to Libraryname.$windows.Windowname
# creates a reference to the window which you can use in your code
Do WinRef.$forecolor.$assign(kBlue) ## changes the window forecolor
```

You can enter the notation for an object in the initial value field for the item reference variable. You can also find the full notation for an object in the Notation Inspector and drag it to the notation field when you enter the *Set reference* command.

You can also use an item reference variable to return a reference to a new object, when using methods to create a new class, instance, or object. Furthermore Omnis contains a special property called $ref which you can use to return an item reference to an object. Both these features are used in the section describing the *Do* command below.

Note that WinRef.$parentfolder, where WinRef is an item reference to a class, will return an item reference to the parent folder of the class and not the $ident value of the folder class containing the class which is usually the case.

Max Chain Depth

The **maxChainDepth** item in the 'defaults' section of config.json allows you to configure the maximum number of field or item references that Omnis will chain through in order to reach the referenced variable.

The default or minimum is 20, and in all but exceptional cases, you should leave this item set to 20. You can change it if you have a heavily recursive method that uses field reference parameters. Since the minimum value is 20, setting this to any value less than 20 results in Omnis using the value 20. The debugger field menu still only chains through up to 20 references.

Current Objects

Under $root, Omnis contains a number of global state variables that tell you about how Omnis is currently executing, or what objects, instances, and methods are currently being used. These objects provide a shortcut to the *current object* or instance that is currently executing. Mostly their names begin with "$c", and they include

- $cclass
  the current class
- $cdata
  the current open data file
- $cinst
  the current instance: usually the instance containing the currently executing method

- $cfield
  the field where the current method is executing
- $clib
  the current library
- $cmethod
  the current executing method
- $cobj
  the current object within a class or instance
- $cparmcount
  the number of parameters that the caller has passed to the current method
- $crecipient
  the current recipient of an event: if a custom method is being processed, $crecipient is the recipient of that method
- $ctarget
  a reference to the target field, that is, the field which currently has the focus (shows the caret and is sent keyboard events)
- $ctask
  the current task: is usually the startup or default task until you open another task
- $cwind
  the current window instance
- $topwind
  the topmost open window instance

You can use the current objects in place of the full notation for a specific object to make the object and its code reusable and portable between libraries. For example, you can use $cinst in a method within a window instance to refer to itself, rather than referring to it by name

```
$cinst
# rather than
$root.$iwindows.WindowInstanceName
```

You can refer to the current library using $clib. For example, to make the current library private use

```
Do $clib.$isprivate.$assign(kTrue)
# is more generic than
Do $libs.MyLibrary.$isprivate.$assign(kTrue)
```

The Flag (#F)

Many of the Omnis commands set a Boolean Omnis variable called the *flag,* or #F, to true or false depending on the success of an operation. Other commands test the current value of the flag and branch accordingly. The *Omnis Studio* Help documents whether or not a command affects the flag. You can return the current status of the flag (#F) in client executed methods in the JavaScript Client using the *flag()* function.

Functions

There are over 350 functions available in Omnis to perform all types of operation and actions that you can include in your Omnis methods. For example, the *sys()* function returns information about the current system, such as the current printer name, the pathname of the current library, or the screen width or height in pixels. The group of *String* functions can be used to manipulate character data, such as the *replace()* function which replaces the first occurrence of a target string, within a source string, with a replacement string, and returns the resulting string. The functions that you can use in your methods are listed in the Omnis Function Reference, where they are listed in functional groups to show the full range and scope of the functions available in Omnis.

Commands

The following sections outline the more important commands or groups of commands in Omnis. The commands that you can use in your methods are listed in the Omnis Command Reference. Here the commands are listed in functional groups to show the full range and scope of the commands available in Omnis. For example, the *Calculations…* group contains the *Calculate* command that lets you do calculations and assign a value to a variable, and the *Do* command that lets you execute and modify objects using the notation. The *Constructs…* group contains programming constructs such as *If…Else If, Repeat…Until,* and *For* loops.

Custom Methods and Properties

You can add methods to the objects in your library and call them what you like; you execute these methods from within the class or instance using the *Do method* command.

You can also create your own properties and methods and execute them using the notation, as you would the standard properties and methods. These are called *Custom Methods and Custom Properties*, or collectively they are referred to as *Custom Notation*. Custom notation can only be executed at runtime, in an instance of the class (e.g. Do $cinst.$custommethodname), and applies either to the instance, or an object contained in the instance.

The name of a custom property or method is case-insensitive. It can be any name starting with the dollar “$” sign, except for the set of reserved names (reserved words) in the following table.

Reserved names or words
$add
$assign
$canassign
$chaincount
$default
$findname
$ident
$isinherited
$makelist
$name
$ref
$serialize
$wind

Any class that can be instantiated can contain custom notation, including remote form, report, table, and object classes (and window classes). In practice you can use custom notation to override the behavior of the standard notation, or to add your own properties and methods to an object.

With the exception of the names in the above table, if the name of a custom method (or property) is the same as a standard one, such as “$printrecord()”, it will override the standard one.

You create custom methods for an object in the method editor. You enter custom notation for a field in the Field Methods for the field, and for a class in the Class Methods for a class.

The code for a custom method must define the method parameters as you would for any other method, and then return the result of executing the method.

The code for a custom property typically comprises two methods. The first, called $propertyname returns the value of the property using the Quit method command. The second called $propertyname.$assign defines a single parameter, which is the new value to be assigned to the custom property; this method usually returns a Boolean value, which is true if the property was successfully assigned. Note that $canassign is always true for custom properties.

An instance of a class contains whatever custom methods and properties you define in the class, together with the properties and methods for that type of instance. The object group $attributes contains all the built-in and custom notation for an instance. You can use $first() and $next() against $attributes, but $add() and $remove() are not available.

You can reference custom notation using the notation **“Notation.$xyz”,** where Notation is some notation for an instance of a class and “$xyz” is the name of your custom property or method. If you specify parameters, such as Notation.$xyz(p1,p2,p3), they are passed as parameters to the custom method, and a value may be returned.

You can use the *Do default* command within the code for a custom method or property, to execute the default behavior for a method or property with the same name as the custom notation. You can use the *Do redirect* command to redirect execution from custom notation in one instance to another instance containing custom notation with the same name.

To create a custom method

- Open the Class or Field methods for a class
- Right-click on the method names list, and select Insert New Method from the context menu
- Enter a name for your custom method, including a dollar sign at the beginning of its name
- Enter the code for the custom method as you would any other method

Userinfo property

File, window, report, menu, toolbar, schema, and query classes have the $userinfo property which you can use to store your own value. The Property Manager only allows you to assign to $userinfo if its current value is empty, null or has character or integer data type. The data stored in $userinfo can be of any other type but it must be assigned to the class at runtime using the $assign() method.

Using Custom Methods

The following example uses a task class containing a custom method called $printprinter(). You can call this method from anywhere inside the task instance using

```
Do $ctask.$printprinter()
```

The $printprinter() method sets the print destination and calls another class method depending on whether the user is accessing SQL or Omnis data; it contains the following code

```
# $printprinter() custom method
Begin reversible block
  Send to printer
  Set report name REPORT1
End reversible block
If iIsSQL
  Do method printSQLData
Else
  Begin reversible block
    Set search name QUERY1
  End reversible block
  Do method printOmnisData
End If
```

The next example uses a window that contains a pane field and a toolbar with three buttons. When the user clicks on a button, the appropriate pane is selected and various other changes are made. Each button has a $event() method that calls a custom method called $setpage() contained in the window class. Note that you can send parameters with the custom method call, as follows

```
# $event() method for second toolbar button
Do $cwind.$setpage(2)
```

The $setpage() custom method contains a parameter variable called pPage, and has the following code

```
# $setpage() custom method
Switch pPage
  Case 1
    Do $cwind.$objs.MainPane.$currentpage.$assign(1)
    Do $cwind.$title.$assign('Queries')
  Case 2
    Do $cwind.$objs.MainPane.$currentpage.$assign(2)

    Do $cwind.$toolbars.$add('tbModify1')
    # installs another toolbar
    Do $cwind.$title.$assign('Modifying')
  Case 3
    Do $cwind.$objs.MainPane.$currentpage.$assign(3)
    Do $cwind.$menus.$add('MReports')
    # installs a menu in the window menu bar
    Do $cwind.$title.$assign('Reports')
  Default
    Quit method kFalse

End Switch
```

The final example uses a window containing a subwindow, which in turn contains a tree list. The subwindow contains a custom method called $buildtree() that builds and expands the tree list. You can call the $buildtree() method from the parent window and send it parameters, using the notation

```
Do $cwind.$objs.SubWin.$buildtree(lv_ClassList)
```

The $buildtree() method contains a parameter variable called pv_SourceList of List type that receives the list passed to it, and a reference variable called TreeRef set to the tree list field, and contains the following code

```
# $buildtree() custom method
Do TreeRef.$setnodelist(kRelationalList,0,pv_SourceList)
Do TreeRef.$expand()
```

Do Command and Executing Methods

While you can use *Calculate* to change an object property or evaluate an expression, you can use the *Do* command for all expressions that execute some notation, including custom methods. In this respect, the *Do* command is the single-most powerful command in Omnis. You can use the *Do* command to set the value of a property, or to run any standard or custom method. The *Do* command has several variants which include

- *Do*
  sends a message to an object in your library, or assigns a value to an object property. Normally you should execute the *Do* command in the context of the *current object* or instance to execute one of its methods or assign to one of its properties. There are a number of common methods that you can use with the *Do* command including $open() to open an instance of a class, $assign() to change an object property, and so on
- *Do inherited*
  executes the inherited method for the current method
- *Do default*
  runs the default processing for a custom method
- *Do redirect*
  redirects method execution to a custom method with the same name as the current method contained elsewhere in your library
- *Do method*
  calls a method in the current class and returns a value
- *Do code method*
  runs a method in a code class and returns a value

Note that you can display a list of built-in methods for an object or object group by clicking on the object in the Notation Inspector and opening the Property Manager. The methods for an object are listed under the Methods tab in the Property Manager: to view all the methods of a class or object, ensure that the 'Show All' option in the Property Manager is enabled. See *Omnis Studio* Help for a complete list of methods for all the objects in Omnis. The Show Runtime Properties option in the Property Manager context menu lets you view properties that are normally available in runtime only, that is, properties of an instance rather than a design class. When runtime properties are visible in the Property Manager the methods for the instance are also shown. You cannot set runtime properties or use methods shown in the Property Manager, they are there as a convenient reference when you are writing code.

Do command

You can use the *Do* command in Omnis to do almost anything: execute some notation, evaluate an expression, and so on. Specifically, you can use it to execute a method for an object or assign a value to one of its properties. The *Do* command returns a value to indicate whether the operation was successful or not, or for some methods a reference to the object operated upon. This section shows you how you can use the *Do* command and introduces some of the most useful methods.

Calling Private Methods

The *callprivate()* function allows you to call a private method within the current class or instance and return a value. The syntax is:

```
Do callprivate(method[,parameters...] ## calls the private method
```

The function can be called in client methods in the JavaScript Client.

$open() method

Using the *Do* command with the notation you can perform many operations that are otherwise performed with a command. For example, the class types that you can open contain an $open() method which you can execute using the *Do* command. For example, you can open a window using

```
Do $windows.style="font-variant: small-caps;">WindowName.$open('InstanceName',kWindowCenter)
# opens a window in the center of the screen
```

The $open() method returns a reference to the instance created. For example

```
# Declare variable WindRef of type Item reference
Set reference WindRef to LIB1.$windows.WindowName
Do WindRef.$open('WindowInstance') Returns WindRef
# WindRef now contains a reference to the window instance
# '$root.$iwindows.WindowInstance' which you can use elsewhere, e.g.

Do WindRef.$forecolor.$assign(kBlue) ## changes the instance
```

You can use a null value instead of an instance name: therefore CLASS.$open('') would force Omnis to use the class name as the instance name. Alternatively, you can use an asterisk in place of the instance name and Omnis assigns a unique name to the instance, using the notation CLASSNAME_number. You can return the instance name in an item reference variable and use the reference in subsequent code. For example

```
# Declare variable iMenuRef of type Item reference
Do $menus.MCUSTOMERS.$open('*') Returns iMenuRef
# iMenuRef now contains a reference to the menu instance, which

# will be something like '$root.$imenus.MCUSTOMERS_23'
```

You can close an instance using the $close() method. For example, the following method opens a window instance, lets the user do something, and closes the instance

```
# initially WindRef contains a reference to the window class
Do WindRef.$open('WindowInstance') Returns WindRef
# let the user do something

Do WindRef.$close()
```

You can close the current window from inside the instance using

```
Do $cwind.$close()
```

Classes that contain the $open() methods also have the $openonce() method. This method opens an instance if one does not already exist (excluding window menus, window toolbars, and cascaded menus). In the case of a window, $openonce() brings the window to the top if it is already open. $openonce() returns an item reference to the new or existing instance, like $open().

$assign() method

You can change the properties of an object, including the properties of a library, class, or field, using the *Do* command and the $assign() method. The syntax for the $assign() method is **Notation.Property.$assign(*Value*)** where **Notation** is the notation for the object, **Property** is the property of the object you want to change, and ***Value*** is a value depending on the context of the object being changed. Usually you can use an Omnis constant to represent a preset value, and for boolean properties, such as preferences, you can use kTrue or kFalse to set the property as appropriate. For example

```
Do $clib.$prefs.$mouseevents.$assign(kTrue)
# turns on mouse events for the current library
Do $cclass.$closebox.$assign(kTrue)
# adds a close box to the current window class
Do $cfield.$textcolor.$assign(kGreen)

# makes the text in the current field green
```

$add() method

You can create a new object in your library using the $add() method. In the notation you are really adding a new object to a particular group of objects. For example, to create a new field on a window you need to add the object to the $objs group of objects for the window, as follows

```
Do $cwind.$objs.$add(kPushbutton,iTop,iLeft,iHeight,iWidth)
# adds a pushbutton to the window with the

# specified size and position
```

When using $add(), you can return a reference to the new object in a return field of type item reference. You can use the reference to change the properties of the new object. For example

```
# Declare variable WindRef of type Item reference
Do $windows.$add('NewWindowName') Returns WindRef
# now use the reference to change the new window
Do WindRef.$style.$assign(kPalette)
Do WindRef.$title.$assign('Window title')
Do WindRef.$clickbehind.$assign(kTrue)
Do WindRef.$keepclicks.$assign(kFalse)
Do WindRef.$modelessdata.$assign(kTrue)
Do WindRef.$backcolor.$assign(kRed)
Do WindRef.$forecolor.$assign(kWhite)

Do WindRef.$backpattern.$assign(2)
```

$redraw() method

Note the $redraw() method is only relevant for fat client windows, not JavaScript Remote Forms which redraw content automatically.

When you change an object or several objects on an open window using the *Do* command, you often need to redraw the window. However if you change an object before $construct() completes execution for the window instance, you don't need to redraw the window. You can redraw an object, window, or all open windows using the $redraw() method. For example

```
Do $cfield.$redraw()
# redraws the current field
Do $cwind.$redraw()
# redraws the current window
Do $root.$redraw()

# redraws all window instances
```

The $redraw() method has three parameters that allow you to specify the extent of the redraw for window fields and/or background objects: the parameters are: $redraw(bSetcontents,bRefresh,bBackObjects) where bSetcontents defaults to true, bRefresh defaults to false, and bBackObjects defaults to false.

```
$root.$redraw(kTrue,kTrue)
# redraws the contents and refreshes all the field in all window
$root.$redraw(kFalse,kFalse,kTrue)

# redraws all background objects for all open windows
```

$sendall() method

You can send a message to all the items or objects in a group using the *Do* command and the $sendall() method. For example, you can redraw all the objects in a group, you can assign a value to all the members of an object group, or you can hide all the members of a group using the $sendall() method and the appropriate message. The full syntax for the method is:

```
Do [group].$sendall({message|message,condition [,bIgnoreUnrecognizedCustomAttribute=kFalse,bRecursive=kFals
```

where *message* is the message you want to send to all the objects and *condition* is a calculation which the objects must satisfy to receive the message. For example

```
Do $iwindows.$sendall($ref.$objs.style="font-variant: small-caps;">FieldName.$redraw())
# redraws the specified field on all window instances
Do $cwind.$objs.$sendall($ref.$textcolor.$assign(kYellow))
# makes the text yellow for all the fields on the current window
Do $cwind.$objs.$sendall($ref.$visible.$assign(kFalse),$ref.$order<=5)
# hides the first five objects on the current window useful

# for window subclasses if you want to hide inherited objects
```

The optional third argument bIgnoreUnrecognizedCustomAttribute causes $sendall() to ignore unrecognized custom attribute errors, which would otherwise cause a runtime error when the library preference $reportnotationerrors is kTrue. This argument defaults to kFalse if omitted.

If bRecursive is kTrue, $sendall() sends the message to all items recursively in window class and instance $objs/$bobjs groups,and remote form class $objs groups.

$sendallref() method

When using $sendall(), you can use $ref to refer to the group member receiving the message. However, you can use $sendallref, which is an item reference to the item currently receiving the message sent with $sendall (note that $sendallref is not supported in client methods). Consider the case where a parameter passed to the message is evaluated by calling another method, or a function implemented in an external component. In this case, if you use $ref in the parameters passed to this other method or function, it will actually refer to the item involved in making the call to evaluate the parameter. This is where $sendallref() could be used, if you wish to pass some property of the group member receiving the message to the other method or function.

For example:

```
Do $cinst.$bobjs.$sendall($ref.$text.$assign(StringTable.$gettext( $cclass.$bobjs.[$sendallref.$ident].$tex
```

The example uses the text stored in the class as the row id in the string table, and assigns the text stored in the string table to the background object. In the example, $sendallref.$ident returns the ident of the background object receiving the message. If you were to use $ref.$ident, the $ref would refer to the custom attribute representing the external component function, and the call to $sendall would not have the desired effect.

$makelist() method

Quite often you need to build a list containing the names of all the objects in a group, and you can do this using the makelist() item group method. The syntax is:

- **Itemgroup.$makelist**($ref.$att1,$ref.$att2,...) generates a list from the item group

Some examples: to build a list of all the classes in the current library and places the result in cLIST:

```
Do $clib.$classes.$makelist($ref.$name) Returns cLIST
```

To build a list of all the currently installed (desktop) menus:

```
Do $imenus.$makelist($ref.$name) Returns cLIST
```

To return only the methods overridden if “MyObject” has a superclass:

```
Do $clib.$objects.[MyObject].$methods.$makelist (...)
```

To build a list of external components currently available in your system:

```
Do $components.$makelist($ref.$name) Returns lXcompList
```

To build a list of window instances currently open in the order that they appear on the screen:

```
Do $iwindows.$makelist($ref.nam) Returns lWinList.
```

If the first argument is the constant kRecursive, the $makelist method ignores containers and adds all objects to the returned list (this also applies to $appendlist, $insertlist, and $count methods for window class and instance $objs and $bobjs groups, and remote form class $objs groups).

Do inherited

The *Do inherited* command runs an inherited method from a method in a subclass. For example, if you have overridden an inherited $construct() method, you can use the *Do inherited* command in the $construct() method of the subclass to execute the $construct() method in its superclass.

Do default

You can use the *Do default* command in a custom method with the same name as a standard built-in method to run the default processing for method. For example, you can use the *Do default* command at the end of a custom $print() method behind a report object to execute the default processing for the method after your code has executed.

Do redirect

You can use the *Do redirect* command in a custom method to redirect method execution to another custom method with the same name that is contained in another object in your library. You specify the notation for the instance or object you want execution to jump to.

Inheritance and custom methods are further discussed in the *Object Oriented Programming* chapter.

NULL values in Calculations

The item "nullValuesWhenORtestedBecomeZero" in the "default" section of the Omnis Configuration file (config.json) controls how null values are treated in calculations.

If "nullValuesWhenORtestedBecomeZero" set to is true, when Omnis finds a NULL value as part of an OR '|' in an *If calculation* it will treat the NULL as zero. If false (the default), a NULL in a calculation results in the entire calculation becoming NULL. For example:

```
If kTrue|#NULL
  # this should fail by default (when nullValuesWhenORtestedBecomeZero is false)
End if
```

By setting "nullValuesWhenORtestedBecomeZero" to true, Omnis will process this if statement as true.

Calculate Command and Evaluating Expressions

This section describes how you use the *Calculate* command with an expression. It also discusses using square bracket notation for strings.

The *Calculate* command lets you assign a value to a variable calculated from an Omnis expression. Expressions can consist of variables, field names, functions, notation strings, operators, and constants. For example

```
Calculate var1 as var2+var3
```

in this case, "var2+var3" is the expression.

```
Calculate var1 as con('Jon', 'McBride')
```

Here the expression uses the *con()* function which joins together, or concatenates, the two strings 'Jon' and 'McBride'. You must enclose literal strings in quotes.

See the *Omnis Studio* Help for a complete list of functions. In expressions, functions appear as the function name followed by parentheses enclosing the arguments to the function. The function returns its result, substituting the result into the expression in place of the function reference. Calling a function does not affect the flag.

The Omnis operators are shown below, in precedence order, that is, the order in which they get evaluated by Omnis. Operators in the same section of the table are of equal precedence, and are evaluated from left to right in an expression.

Operator	Description
()	Parentheses
-	Unary minus
* /	Multiplication, Division

Operator	Description
+ -	Addition, Subtraction
< > = <= >= <>	Less than, Greater than, Equal to, Less than or equal to, Greater than or equal to, Not equal to
& |	Logical AND, Logical OR

When you combine expressions with operators, the order of expressions will often make a difference in the interpretation of the expression: this is a consequence of the mathematical properties of the operators such as subtraction and division. You can group expressions using parentheses to ensure the intended result. For example

```
Calculate lv_Num as 100 * (2 + 7)
```

evaluates the expression in parentheses first, giving a value of 900. If you leave off the parentheses, such as

```
Calculate lv_Num as 100 * 2 + 7
```

Omnis evaluates the * operator first, so it multiplies 100*2, then adds 7 for a value of 207.

Square Bracket Notation

You can use a special notation in strings to force Omnis to expand an expression into the string. You do this by enclosing the expression in square brackets: Omnis evaluates the expression when the string value is required. You can use this in all sorts of ways, including the technique of adding a variable value to the text in the SQL or text buffer.

You can use square bracket notation wherever you can specify a single variable or field name, including

- Command parameters, for example, *OK message*

```
OK message {Your current balance is [lv_curbalance]}
```

- Window or report fields: you can include values in text objects, such as

```
Your current balance is [lv_curbalance]
```

- Variable or field names within a *Calculate* command or text object
- Function parameters

Square bracket notation lets you refer to a value indirectly letting you code general expressions that evaluate to different results based on the values of variables in the expression: this is called *indirection*. For example, you can include a variable name enclosed in square brackets in a text object to add the value to the text at runtime. However in general, there is a significant performance penalty in using indirection.

If you need to use [or] in a string but do not want the contents evaluated, then use [[and] to enclose the contents—double up the first or opening square bracket. This is useful when you use square bracket notation with external languages that also use square brackets, such as the VMS file system or DDE.

Type Conversion in Expressions

Omnis tries its best to figure out what to do with values of differing data types in expressions. For example, adding a number and a string generally isn’t possible, but if Omnis can convert the string into a number, it will do so and perform the addition. Some other examples are

```
# Declare local variable lDate of type Date D m Y
Calculate lDate as 1200
# 1200 is no. of days since 31st Dec 1900
Calculate lDate as 'Jun 5 93'
# conv string to date in format D m Y
OK message {Answer is [jst(lDate,'D:D M CY')]} ## reformat date

Calculate lNum as lDate ## sets lNum to 1200, the no. of days
```

Boolean values have a special range of possibilities.

- YES, Y, or 1 indicate a true status
- NO, N, or 0 indicate a false status

FALSE and TRUE are not valid values: Omnis converts them to empty.

```
# Declare local variable LBOOL of type Boolean
Calculate LBOOL as 1 ## is the same as...
Calculate LBOOL as 'Y' ## or 'YES'
# the opposite is
Calculate LBOOL as 0 ## or 'NO' or 'N'
OK message { The answer is [LBOOL] }
Calculate LBOOL as 'fui' ## is the same as...

Calculate LBOOL as ''
```

You can convert any number to a string and any string that is a number in string form to a number.

```
# Declare local variable lChar of type Character
# Declare local variable lNum of type Number floating dp
Calculate lChar as 100
OK Message { [lChar], [2 * lChar], and [con(lChar,'XYZ')] }
# Gives message output 100 200 and 100XYZ
Calculate lNum as lChar
Calculate lChar as lNum
OK Message { [lChar], [lNum * lChar], and [con(lChar,'ABC')] }

# Gives message output 100 10000 and 100ABC
```

Constants

You will often find situations in Omnis where you must assign a value that represents some discrete object or preset choice. Omnis has a set of predefined constants you should use for this kind of data. For example, a class type can be one of the following: code, file, menu, report, schema, and so on. Each of these is represented by a constant: kCode, kFile, kMenu, kReport, kSchema, respectively. You can get a list of constants from the Catalog: press F9/Cmnd-9 to open the Catalog. You can use constants in your code, like this

```
Calculate obj1.$align as kRightJst ## or use Do
Do obj1.$align.$assign(kRightJst)

# aligns the object obj1 to the right
```

Although you can use the numeric value of a constant, you should use the predefined string value of a constant in your methods. In addition to ensuring you're using the right constant, your code will be much more readable. Moreover, there is no guarantee that the numeric value of a particular constant will not change in a future release of Omnis.

Calling Methods

You can execute another method in the current class using *Do method* (or call a method in a code class using *Do code method*). These commands let you pass parameters to the called method and return a value in a return field – note that the value of the return field is cleared before the method is called. For example, the following method named Setup calls another method named Date and returns a value.

```
# Setup method
Do method Date (lNum,lDate+1) Returns lDate

OK message {Date from return is [lDate]}
# Date method, the called method
# Declare Parameter var lpNum of type Number 0 dp
# Declare Parameter var lpDate of type Short Date 1980..2079
OK message {Date from calling method is [lpDate], number is [lpNum]}

Quit method {lpDate + 12}
```

Note that when you call a code class method from within an instance the value of $cinst, the current instance, does not change. Therefore you can execute code in the code class method that refers to the current instance and it will work.

WARNING Omnis does not stop a method calling itself. You must be careful how the method terminates: if it becomes an infinite loop, Omnis will exhaust its method stack.

Quitting Methods

You can use the *Quit* command, and its variants, to quit methods at various levels.

- *Quit method*
  quits the current method and returns a value to the calling method, if any
- *Quit event handler*
  quits an event handling method
- *Quit all methods*
  quits all the currently executing methods, but leaves Omnis running
- *Quit all if canceled*
  quits all methods if you press Cancel
- *Quit Omnis*
  exits your application and Omnis

You can also clear the method stack with the *Clear method stack* command, which does the same thing as the debugger menu **Stack>>Clear Method Stack**: it removes all the methods except for the current one. If you follow *Clear method stack* with *Quit method*, it has the same effect as *Quit all methods*.

Note: By enabling the **Use Minimum Lengths** option on the Modify>>Filter Commands submenu in the Method Editor, the *Quit method* command is selected by default when you type just the letter 'q' (rather than the Queue commands); by typing 'qu' all the Quit methods will be shown in the Code Assistant help list.

The *Quit method* command allows you to exit a method and return a value to the calling method (it is the same as Return in other languages). For example:

```
# Quit the method myMethod and return the flag from the Yes/No message
# to the calling method the calling method

Do method myMethod Returns lReturnFlag
# method myMethod
Yes/No message {Continue ?}

Quit method #F
```

It is possible to call another method in the return value of a *Quit method* command, but this can lead to unpredictable results, especially if the called method contains an *Enter Data* command, e.g.

```
Quit method Returns iOtherObject.$doSomeThingThatContainsEnterData
```

Flow Control Commands

The *Constructs...* group contains many commands that let you control the execution and program flow of your methods. *If* statements let you test a condition and branch accordingly: loop commands iterate based on tests or sequences: the *Comment* command lets you comment your code: and reversible blocks let you manipulate objects and values and restore their initial values when the block terminates.

Several commands in this command group have starting and terminating commands (*If* and *End if*, for example). You must use the correct terminating command, or you will get unexpected results. If chromacoding is enabled, the beginning and terminating commands for most branching and looping constructs are highlighted. You can enable chromacoding using the View>>Show ChromaCoding menu option in the method editor.

Highlighting Blocks

The start and end of any block commands are highlighted when one of the statements that makes up the construct formed by the commands is selected in the method editor. For example, if a **For** statement is the current line, then the “End for” and “For” will both be highlighted. Or if a **Case** statement is the selected line, then all cases in the same switch, “Default”, “Switch” and “End switch” will all be highlighted. The style or color of the highlighting uses a pair of chroma coding options, $currentblocktextcolor and $currentblockstyle.

Branching Commands

The *If* command lets you test the flag, a calculation, or a Cancel event. The Flag is an Omnis variable with a True or False value which is altered by some commands to show an operation succeeded, or by user input. The *Else* command lets you take an alternative action when the *If* evaluates to false, *Else if* gives you a series of tests. You must use the *End If* command to terminate all *If* statements.

A simple test of the flag looks like this:

```
If flag true
  Do method Setup
End If
```

You can do a sequential checking of values using a calculation expression:

```
If CollCourse ='French'
  Do method Languages
Else If CollCourse = 'Science'
  If CollSubCourse = 'Biology'
    Do method ScienceC1
  Else
    Do method ScienceC2
  End If
Else
  OK message {Course is not available.}
End If
```

While Loops

The *While* loop tests an expression at the beginning of a loop. The *While* command will not run the code block at all if the expression is false immediately. You would use a *While* command when you want to loop while an expression is true.

```
# Declare Count with initial value 1
While Count <= 10
  OK message {Count is [Count]}
  Calculate Count as Count + 1
End While
```

This loop will output 10 messages. If the condition was ‘Count <= 1’, it would run only once.

Repeat Loops

A *Repeat* loop lets you iterate until an expression becomes true. Repeat loops always execute at least once, that is, the test specified in the *Until* command is carried out at the *end* of the loop, after the commands in the loop are executed, whereas While loops carry out the test at the beginning of the loop.

```
# Declare Count of Integer type with initial value 1
Repeat
  OK message {Count is [Count]}
  Calculate Count as Count + 1
Until Count >= 10
```

This loop will output 9 messages.

For Loops

The *For field value* command lets you loop for some specific number of iterations, using a specified variable as the counter. The following example builds a string of ASCII characters from their codes using the functions *con()* and *chr()*.

```
# Declare Count
Calculate cvar1 as '' ## clear the string
For Count from 48 to 122 step 1 ## set the counter range
  Calculate cvar1 as con(cvar1,chr(Count)) ## add char to string
  Do $cwind.$redraw()
End For
```

The *For each line in list* command loops through all the lines in the current list.

```
Set current list LIST1
For each line in list from 1 to LIST1.$linecount step 1
  # process each line

End For
```

Switch/Case Statements

The *Switch* statement lets you check an expression against a series of values, taking a different action in each case. You would use a *Switch* command when you have a series of possible values and a different action to take for each value.

The following method uses a local variable lChar and tests for three possible values, “A”, “B”, and “C”.

```
# Parameter pString(character 10) ## receives the string
Calculate lChar as mid(pString, 1, 1) ## takes the first char
Switch lChar
  Case 'A'
    # Process for A
  Case 'B'
    # Process for B
  Case 'C'
    # Process for C
  Default
    # do default for all cases other than A, B, or C

End Switch
```

It is a good idea to use the *Switch* command only for expressions in which you know all the possible values. You should always have one *Case* statement for each possible value and a *Default* statement that handles any other value(s).

Escaping from Loops

While a loop is executing you can break into it at any time using the *break key* combination for your operating system: under Windows it is Ctrl-Break, under macOS it is Cmnd-period, and under Unix it is Ctrl-C. Effectively, this keypress ‘quits all methods’. When Omnis performs any repetitive task such as building a list, printing a report, or executing a Repeat/While loop, it tests for this keypress periodically. For Repeat/While loops, Omnis carries out the test at the end of each pass through the loop.

To create a more controlled exit for the finished library, you can turn off the *end of loop* test and provide the user with a working message with a **Cancel** button. When the **Cancel** button is visible on the screen, pressing the Escape key under Windows or Cmnd-period under macOS is the equivalent to clicking **Cancel**. For example

```
Disable cancel test at loops ## disables default test for loops
Calculate Count as 1
Repeat
  Working message (Cancel box) {Repeat loop...}
  If canceled
    Yes/No message {Do you want to escape?}
    If flag true
      Quit all methods
    End If
  End If
  Calculate Count as Count+1

Until Count > 200
```

The *If canceled* command detects the Cancel event and quits the method. To turn on testing for a break, you can use the *Enable cancel test at loops* command.

The *Break to end of loop* command lets you jump out of a loop without having to quit the method, and the *Until break* provides an exit condition which you can fully control. For example

```
Repeat
  Working message (Cancel box) {Repeat loop...}
  If canceled
    Yes/No message {Are you sure you want to break out?}
    If flag true
      Break to end of loop
    End If
  End If
Until break

OK message {Loop has ended}
```

If you have not disabled the cancel test at loops, a Ctrl-Break/Cmnd-period/Ctrl-C terminates all methods and does not execute the *OK message*. Having turned off the automatic cancel test at loops, you can still cause a *Quit all methods* when canceled. For example

```
Disable cancel test at loops
Calculate Count1 as 1
Calculate Count2 as 1
Repeat
  Repeat
    Working message (Cancel box) {Inner repeat loop}
    Calculate Count2 as Count2 + 1
  Until Count2 > 12
  Calculate Count2 as 1
  Working message (Cancel box) {Outer repeat loop...}
  Quit all if canceled
  Calculate Count1 as Count1 + 1

Until Count1 > 20
```

If the user selects Cancel in the outer loop, the method quits, but from the inner loop there is no escape.

Optimizing Program Flow

Loops magnify a small problem into a large one dependent on the number of iterations at runtime, and other program flow commands can use a lot of unnecessary time to get the same result as a simpler command.

Here are some tips to help optimize your methods.

Use the *For* command instead of the equivalent *While* or *Repeat* commands. *For* has a fixed iteration, while the other commands test conditions. By eliminating the expression evaluation, you can save time in a long loop.

Use the *Switch* command instead of equivalent *If*/*Else* commands where possible. Arrange both the *Case* commands within a *Switch* and the several *If* and *Else* if commands so that the conditions that occur most frequently come first.

Use the *Quit method* command to break out of a method as early as possible after making a decision to do so. This can be a tradeoff with readability for long methods because you have multiple exits from the method: if falling through to the bottom of the method involves several more checks, or even just scanning through a large block of code, you can substantially improve performance by adding the *Quit method* higher up in the code.

Avoid using commands that don't actually execute *within* a loop. For example, don't put comment lines inside the loop. You can also use *Jump to start of loop* to bypass the rest of that iteration of the loop.

You can speed up a frequently called method by putting *Optimize method* at the start: refer to *Omnis Studio* Help for details of this command.

Reversible Blocks

A reversible block is a set of commands enclosed by *Begin reversible block* and *End reversible block* commands: a reversible block can appear anywhere in a method. Omnis reverses the commands in a reversible block automatically, when the method containing the reversible block ends, thus restoring the state of any variables and settings changed by the commands in the reversible block.

```
# commands...
Begin reversible block
 # commands...
End reversible block

# more commands...
```

Reversible blocks can be very useful for calculating a value for a variable to be used in the method and then restoring the former value when the method has finished. Also you may want to change a report name, search name, or whatever, knowing that the settings will return automatically to their former values when the method ends.

The Omnis Help (press F1) indicates which commands are reversible.

Consider the following reversible block.

```
Begin reversible block
  Disable menu line 5 {Menu1}
  Set current list cList1
  Define list {cvar5}
  Build window list
  Calculate lNum as 0
  Open window instance Window2
End reversible block

# more commands...
```

When this method terminates:

- Omnis closes window Window2
- Omnis restores lNum to its original value
- The definition of cList1 returns to its former definition
- Omnis restores the former current list
- Omnis enables line 5 of Menu1

At the end of the method, Omnis steps back through the block, reversing each command starting with the last. If there is more than one reversible block in a method, Omnis reverses the commands in each block, starting from the last reversible block. If you nest reversible blocks, the commands in all the reversible blocks are treated as one block when they are reversed, that is, Omnis steps backward through each nested reversible block reversing each command line in turn. You cannot reverse any changes that the reversible block makes to Omnis data files or server-based data unless you carefully structure the server transaction to roll back as well.

Losing property values

Certain notation properties affect other properties when they are assigned, for example, assigning $calculated to kFalse clears $text for the field. Therefore, if the $calculated property is set within a reversible block, and the state of $calculated is reversed, the value in the $text property is not reinstated. Such relationships between properties are not supported by the reversible block mechanism. If you wish to maintain the value of a property that may get cleared during notation execution, you should store the value in your own variable and assign the value to the property at runtime.

Error Handling

When you enter a command, Omnis automatically checks its syntax. When a command is executed in a method, you can get a *runtime error*, a processing error rather than a syntax error. *Fatal errors* either display a message and stop method execution or open the debugger at the offending command.

You can cause a fatal error to occur with the *Signal error* command, which takes an error number and text as its argument. This lets you define your own errors, but still use the standard Omnis error handler mechanism.

In addition, Omnis maintains two global system variables #ERRCODE and #ERRTEXT that report error conditions and warnings to your methods. Fatal errors set #ERRCODE to a positive number greater than 100,000, whereas warnings set it to a positive number less than 100,000.

You can trap the errors and warnings by adding a method to test for the various values of #ERRCODE and control the way Omnis deals with them: this is called an *error handler*. The command *Load error handler* takes the name of the method and an optional error code range as its parameters:

```
Load error handler Code1/1 {Errors}

# warnings and errors will be passed to handler in code class
```

Once you install it, Omnis calls the error handler when an error occurs in the specified range. Please refer to the *Omnis Studio* Help for a detailed description of the *Load error handler* command and examples of its use.

There are several commands prefixed with SEA, which stands for Set error action. Using these commands, you can tell Omnis what to do after an error:

- *SEA continue execution*
  continues method execution at the command following the command that signaled the error: if the error handling routine has not altered them, #ERRCODE and #ERRTEXT are available to the command
- *SEA report fatal error*
  if the debugger is available, it displays the offending command in the method window and the error message in the debugger status line
- *SEA repeat command*
  repeats the command that caused the error.

Repeating a command should be done with care since it is easy to put Omnis into an endless loop. If the error has a side effect, it may not be possible to repeat the command. If an ‘Out of memory’ condition occurs, it may be possible to clear some lists to free up enough memory to repeat the command successfully.

Errors in the JavaScript Client

You can return the values of #ERRCODE and #ERRTEXT in client executed methods in the JavaScript using the functions errcode() and errtext().

Error Reporting for External Components

The item “allExternalComponentErrorsAreFatal” in the “defaults” section of the Omnis configuration file (config.json) allows you to manage whether or not #ERRCODE and #ERRTEXT are reported by external components. When allExternalComponentErrorsAreFatal is true (the default), and an external component sets #ERRCODE and #ERRTEXT, the error always generates a runtime error, entering the debugger in the development version of Omnis.

Calculation Errors

The library preference $reportcalculationerrors (default is true) specifies whether or not calculation errors are reported. When true, Omnis will report errors that occur when evaluating calculations, such as divide by zero errors. The report message is sent to the trace log, containing the error and code that caused the problem.

In addition, when executing calculations using *Do* and *Calculate*, Omnis enters the debugger when the error occurs (provided debugging is allowed). (This will not occur when these commands execute in the Omnis Web Client plug-in.)

Notation Errors

The item “stricterNotationErrorChecks” in the “defaults” section of the Omnis configuration file (config.json) allows you to control the sensitivity for detecting certain errors in notation. When set to true (the default), certain unresolved name errors, e.g. such as notation in the form $cinst.name or $ctask.name, result in a debugger (or runtime) error if $clib.$prefs.$reportnotationerrors is kTrue.

Redrawing Objects

There are a number of commands that let you redraw a particular object or group of objects. The Redraw command has the following variants.

- *Redraw* field or window
  redraws the specified field or window, or list of fields or windows: note this command with refresh all instances of the window
- *Redraw lists*
  redraws all list fields on the current window or redraws all lists in your library
- *Redraw menus*
  redraws all the currently installed menus
- *Redraw toolbar*
  redraws the specified custom toolbar

You can use the $redraw() method to redraw a field or fields, a window or all windows, as described earlier in this chapter.

Refreshing window instances

You can use the $norefresh window instance property to control the refreshing of windows. When set to kTrue screen updates are disabled and the window is not refreshed. You can use this property to improve performance, for example when setting a large number of exceptions for a complex grid. Setting the $norefresh property to kFalse will enable screen refreshing.

Message Boxes

There are a number of message boxes you can use in your library to alert the user. The commands for these messages are in the *Message boxes*... group. They include

- *OK message*
  displays a message in a box and waits for the user to click an OK button. For emphasis you can add an info icon and sound the system bell. You can use square bracket notation in the message text to display the current value of variables or fields. For example, *OK message {[sys(5)]}* will display the user’s serial number. You can use the kCr constant enclosed in square brackets to force a line break in the message, e.g. ‘First line[kCr]Second line’.
- *Yes/No message*, and *No/Yes message*
  displays a message in a box and waits for Yes or a No answer from the user. Either the Yes or the No button is the default. You can use the kCr constant enclosed in square brackets to force a line break in the message, e.g. ‘First line[kCr]Second line’.
- *Prompt for input*
  displays a dialog prompting the user for input
- *Working message*
  displays a message while the computer is processing data or executing a method: with a Cancel button the user can break into the processing with Ctrl-Break/Cmnd-period/Ctrl-C

Events

Events are reported in Omnis as ***Event Messages***. These messages are sent to the event handling methods as one or more ***Event Parameters***. The first parameter of an event message, pEventCode, contains an *event code* representing the event. Event messages may contain a second, a third, or subsequent parameters that tell you more about the event. For example, a click on a list box will generate an evClick event plus a second parameter pRow telling you the row number clicked on. All event codes are prefixed with the letters “ev”, and all event parameters are prefixed with the letter “p”. You can use the event codes in your event handling methods to detect specific events, and the event parameters to test the contents of event messages.

When an event occurs the *default action* normally takes place. For example, when the user presses the tab key to move to the next field on a data entry field, the default action is for the cursor to leave the current field and enter the next field on the window, and normally this is exactly what happens. However you could put a method behind the field that performs any one of a number of alternative actions in response to the tab. That is, the event handling method could use the tab to trigger a particular piece of code and then allow the default action to occur, it could pass the event to somewhere else in your library, or it could discard the event altogether and stop the default action from happening.

Event Handling Methods

You can write an event handling method for each field and object contained in remote form, remote menu, window, menu, toolbar, and report classes. The other class types do not generate events. Events for remote forms and the JS components are described in the *Creating Web & Mobile Apps* manual.

You can add the event methods for window and report fields in the *Field Methods* for the class. For menu classes you can add an event method to the *Line Methods* for a menu line, and for toolbar classes you can enter an event method in the *Tool Methods* for each toolbar control.

Window fields, toolbar controls, and menu lines contain a default event handling method called $event, and report fields contain a default event handling method called $print. If you open the field methods for a window field, toolbar control, or menu line you will see an $event method, and for each report field you will see a $print method for the object. These are the default event handling methods for those objects.

To view the event handling method for a field or object

- Show the design screen for the class
- Right-click on the field, menu line or toolbar control
- Choose Field Methods, Line Methods, or Tool Methods, as appropriate

The method editor opens showing the first method in the list for the field or object. If this is not the $event method, select it from the list to view it. Some event handlers will contain code to handle a range of possible events in the object.

The event handling method for some types of field may be empty, because there is only one possible event for the object. For example, the event handling method for a menu line is empty since you can only select a menu line. Therefore any code you put in the $event method for a menu line runs automatically when you select the line.

To enter the code for an event handling method

- Assuming you have opened a default $event method for a field, click on the next command line after the *On* event command
- Then on the next line enter the code you want to run for that event
- For example, you can open the event method for a pushbutton, that contains a single *On evClick* command which will detect a click on the button.

Some $event methods are empty, so for these:

- Select the first line of the method
- Type **on** to select the *On* command
- Type **ev** plus the first letter of the event you want to enter, or select the event from the Helper window that pops up in the Code Editor

Omnis validates the event codes you have entered, that is, it checks to see if the event code is valid for the current object, and if not, it will flag it as an error.

- When the line is complete, press Return and enter the code for your event method

The code for an event method could literally do anything, but in practice it would generally perform an action of some kind related to the object triggering the event (e.g. a button might trigger a Save, so you need to write code to do the save operation). You could use the *Do* command and some notation in your event handling method, or you can use the *Do method* command to run another method in the current class or instance, or the *Do code method* command to run a method in a code class: in all cases, you can put literally any code in an event handling method and it will run given the right event.

The On Command

You can use the *On* command to detect events in your event handling methods. Fields from the Component Store may contain a default event handling method with one or more *On* commands to detect different events. For example, an entry field contains the method

```
On evBefore     ## Event Parameters - pRow ( Itemreference )

On evAfter      ## Event Parameters - pClickedField, pClickedWindow, pMenuLine, pCommandNumber, pRow
```

These lines detect the events **evBefore** and **evAfter**, which are the event codes contained in the message sent when the user enters or leaves the field, respectively. The in-line comments indicate which event parameters Omnis supplies for that event. In most cases, the event parameters are references containing values to do with the context of the event: the field clicked on, the list row number, the menu line number, and so on.

When you select the *On* command in the method editor, the list of possible events changes to reflect the events supported by the object to which $event belongs. You can also view events divided into categories, by using the Events tab of the Catalog.

You can use the default event handling method for a field or add your own. The following event handler for a data entry field detects an evBefore as the user enters the field and performs a calculation changing the value of the field.

```
On evBefore ## user tabs into date field
  Calculate cDate as #D ## cDate is the dataname of the field
  Redraw {DateField} ## the current field

  Quit event handler
```

Code which is common to all events should be placed at the start of the event handling method, *before* any *On* commands. You can use the *On default* command to handle any events not covered by an earlier *On* command line. The general format is

```
# code which will run for all events
On evBefore
  # code for evBefore events
On evAfter
  # code for evAfter events
On default

  # code for any other events
```

When you enter the *On* command in an event handling method, it displays a list of all the available event codes in the command palette. You can click on the one you want, or you can enter more than one event code for a single *On* command, for example *On evClick, evDoubleClick*. *On* commands cannot be nested or contained in an *If* or loop construct.

When you have entered the *On* command line for a particular event and selected the next command line, you can open the Catalog to view the event parameters for that event code.

- Click on the line ***after*** an *On evClick* command line
- Open the Catalog (F9/Cmnd-9)
- Click on Event Parameters under the Variables tab

For example, an *On evClick* command displays the parameters pEventCode and pRow in the Catalog. You can use these event parameters in your event handling methods to test the event message. A click on a list box generates an evClick event message containing a reference to the row clicked on, held in the pRow event parameter. You can test the value of pRow in your code

```
On evClick ## method behind a list box
  If pRow=1 ## if row 1 was clicked on
    # Do this...
  End If
  If pRow=2 ## if row 2 was clicked on
    # Do that...

  End If
```

All events return the parameter pEventCode, which you can also use in your event handling methods.

```
On evAfter,evBefore ## method behind field
  # Do this code for both events
  If pEventCode=evAfter
    # Do this for evAfter events only
  End If
  If pEventCode=evBefore
    # Do this for evBefore events only

  End If
```

The parameters for the current event are returned by the *sys(86)* function, which you can use while debugging or monitoring which events are handled by which methods. For example, you could use the *Send to trace log* command and the functions *sys(85*) and *sys(86),* to report the current method and events, in the $event method for a field

```
# $event method for field 10 on the window
Send to trace log {[sys(85)] - [sys(86)]}
# sends the following to the trace log when you tab out of the field
WindowName/10/$event - evAfter,evTab

WindowName/10/$event - evTab
```

You can use any of the parameters reported for an event in your event handling methods. However, if you enter an event parameter not associated with the current event, the parameter will be null and lead to a runtime error.

Mouse Events

Mouse events allow you to detect user mouse clicks inside fields and the background of a window. Mouse and right- mouse button events are generated only if the $mouseevents and $rmouseevents library preferences are enabled. Under macOS, right-mouse events are generated when you hold down the Ctrl key and click the mouse.

- **evMouseDouble** and **evRMouseDouble**
  the mouse, or right-mouse button is double-clicked in a field or window
- **evMouseDown** and **evRMouseDown**
  evMouseUp and **evRMouseUp**
  the mouse, or right-mouse button is held down in a field or window, or the mouse button is released: for the mouse-down events you can detect the position of the mouse, se below
- **evMouseEnter** and **evMouseLeave**
  the mouse pointer enters, or leaves a field
- **evDrag**
  the mouse is held down in a field and a drag operation is about to start: the parameters report the type and value of the data
- **evCanDrop**
  whether the field or window containing the mouse can accept a drop: the parameters reference the object being dropped, the type and value of the data
- **evWillDrop**
  the mouse is released at the end of a drag operation. The parameters reference the object being dropped, the type and value of the data
- **evDrop**
  the mouse is released over the destination field or window at the end of a drag operation. The parameters reference the object being dropped, the type and value of the data

For the evMouseDown, evMouseUp, evRMouseDown and evRMouseUp events you can return the position of the mouse as the X-Y coordinates relative to the window background or field.

- **pMouseX**
  Mouse x coordinate

- **pMouseY**
  Mouse y coordinate

- **pMouseHwnd**
  window identifier of the hwnd receiving the mouse event: the mouse coordinate parameters are relative to this hwnd

The coordinate origin is the top-left of the hwnd.

Move Behind Events

From Studio 10, the $movebehind property allows you to control whether or not to allow mouse move events on fields in window classes, other than the top window. By default $movebehind is set to kTrue and will allow mouse move events to be processed in other windows when the window is the top window, e.g. evMouseEnter and evMouseLeave. Set the property to kFalse to turn off this behavior for the window.

In previous versions, only evMouseEnter and evMouseLeave in a complex grid in a window that was not top were allowed. To revert to the legacy behavior, add an entry called "oldMouseMoveBehindBehaviour" to the "defaults" group in the config.json file and set it to true.

Discarding Events

In certain circumstances you might want to detect particular events and discard them in order to stop the default action from occurring. You can discard or throw away events using the *Quit event handler* command with the Discard event option enabled. Note however, you *cannot* discard some events or stop the default action from taking place since the event has already occurred by the time it is detected by an event handling method. In this case, a *Quit event handler (Discard event)* has no effect for some events.

Being able to discard an event is useful when you want to validate what the user has entered in a field and stop the cursor leaving the field if the data is invalid. The following method displays an appropriate message and stays in the field if the user does not enter the data in the correct format.

```
On evAfter ## as user leaves the field
  If len(CustCode <> 6) ## check a value has been entered
    If len(CustCode = 0) ## field left blank
      OK message {You must enter a customer code}
    Else
     ## wrong length code entered
      OK message {The customer code must have 6 digits}

    End If
    Quit event handler (Discard event) ## stay in the field

  End If
```

You can also handle or discard events using the *Quit method* command with a return value of kHandleEvent or kDiscardEvent, as appropriate.

The Quit event handler Command

If you want to discard or pass an event you can use the *Quit event handler* command to terminate an *On* construct. A field event handling method might have the following structure.

```
# general code for all events
On evBefore
  # code for evBefore events
On evAfter
  # code for evAfter events
On evClick,evDoubleClick
  # code for click events
  Quit event handler (pass event)
On default

  # code for any other events
```

The *Quit event handler* command has two options

- **Discard event**
  for some events you can discard the event and stop the default action taking place
- **Pass to other handlers**
  passes the event to the next handler in the chain

Window Events

Note the following section refers to events in window classes only, so to do not apply to remote forms or JavaScript components.

So far the discussion has focused on field events, which you would normally handle in the field using an event handling method. However you can enter methods to handle events that occur in your window as well. Like fields, the event handling method for a window class is called $event, and you enter this method in the *Class Methods* for the window class.

Window classes do not contain an $event method by default, but you can insert a method with this name. You enter the code for a window $event method in exactly the same as for fields using the *On* command to detect events in your window.

Window events affect the window only and not individual fields. They include clicks on the window background, bringing the window to the front or sending it to the back, moving it, sizing it, minimizing or maximizing the window, or closing it. For example, when you click on a window's close box, the evCloseBox and evClose events are generated in the window indicating that the close box has been clicked and the window has been closed. You could enter an $event method for the window to detect these events and act accordingly.

The following window $event method detects a click on a window behind the current window, and discards the click if the user is inserting or editing data.

```
On evWindowClick ## user has clicked on a window behind
  If cInserting | cEditing ## vars to detect current mode
    OK message {You cannot switch windows while entering data}
    Quit event handler (Discard event) ## keep window on top
  End If

  Quit event handler
```

The following window $event method checks for events occurring in the window and runs the appropriate methods elsewhere in the class.

```
On evToTop
  Do method Activate

  Quit event handler
On evWindowClick
  Do method Deactivate

  Quit event handler
On evClose
  Do method Close

  Quit event handler
On evResized
  Do $cwind.$width.$assign($cclass.$width)
  Do $cwind.$height.$assign($cclass.$height)

  Quit event handler (Discard event)
```

When a window with the $edgefloat property set to floating edges is resized the evResized event is reported; this can occur either when the window itself is resized, or due to the main Omnis application window being resized (the latter only applies on the Windows platform, since on macOS the Omnis application occupies the whole screen).

Note you cannot trap an evResized and discard it since the resizing has already occurred, but you can reverse the resizing by setting the size of the open window back to the size stored in the class.

Window Event Handling (macOS)

Under macOS, whenever the end-user clicks on a window title bar (or a button on the window title bar) the evWindowClick event is generated. The event parameter pStayingBehind is true if the window receiving the click will not come to the front as a result of the click (this event can only ever be true on macOS). For example, when the user clicks on the zoom box of a window that is not on top, the window will zoom or restore, but will not come to the top.

Key events

You can detect which key or key combination is pressed by trapping the evKey event. The $keyevents library preference must be set to kTrue to enable the evKey event to be called for all window class foreground objects, including entry fields. The evKey event is sent to the target field when a key is pressed, and has three parameters as follows:

- **pEventCode**
  The event code
- **pKey**
  The key pressed
- **pSystemKey**
  The system key pressed represented by a code, as follows:

10% Code	40% Key	10% Code	40% Key
0	Letter/Number	27	Tab
1...12	F1...F12	28	Return
17	Up Arrow	29	Enter
18	Down Arrow	30	Backspace
19	Left Arrow	32	Esc
20	Right Arrow	34	Delete
21	Page Up	35	Insert
22	Page Down	53	Context Menu
25	Home	100	Pause *
26	End		

*The pSystemKey event parameter has a value of 100 to signal the Pause button (Windows **VK_PAUSE** virtual key) has been pressed.

Control Methods and Passing Events

As already described, you handle events for fields using an event handling method contained in the field, but you can add a further level of control over field events by adding a method called $control to your window. This method is called a *window control method*. To allow this method to handle events you must pass events to it from the field event handling methods. You do this by including in your field event handler the *Quit event handler* command with the **Pass to next handler** option enabled.

As a further level of control, you can add a method called $control to your tasks. This method is called a *task control method*. Events are passed to the task control method from the window control method contained in the window belonging to the task. Therefore, an event may be generated in the field, passed to the window control method, and then passed to the task control method.

Window events that are handled in the $event method for a window can be passed to the task $control method as well.

At each level an event handling method can discard the event or pass it on to the next event handler. At the task level, the highest level of control, the event can be processed and the default action takes place, or the event can be discarded and no further action occurs.

The Omnis event processing mechanism gives you absolute control over what is going on in your application, but it also means you need to design your event handling methods with care. It is important not to pass on an event to higher levels unnecessarily and to keep control methods short, to limit the time spent processing each event.

In the following example, the $control method is contained in an Omnis data entry window. It sets the main file for the window when it is opened or comes to the top, and does not let the user close the window if Omnis is in data entry mode.

```
On evToTop
  # window comes to the top or is opened
  Set main file {FCUSTOMERS}
  Quit event handler
On evClose
  If cInserting | cEditing ## vars to detect current mode
    # User closes window when in enter data mode
    OK message {You can't close in enter data mode}
    Quit event handler (Discard event)

  End If
```

Event Processing and Enter Data Mode

Normally, the default processing for an event takes place when all the event handler methods dealing with the event have finished executing. It is not possible to have active unprocessed events when waiting for user input so the default processing is carried out for any active events after an *Enter data* command has been executed or at a debugger break. Therefore if required, you can use the *Process event and continue* command to override the default behavior and force events to be processed allowing an event handling method to continue.

The *Process event and continue (Discard event)* option lets you discard the active event. For example, in an event handler for evOK the following code would cause the OK event to be thrown away before the subsequent enter data starts.

```
On evOK
  Process event and continue (Discard event)
  Open window instance {window2}

  Enter data
```

Container Fields and Events

Container fields are fields that contain other fields: examples of container fields include subwindows, tab panes, page panes, scroll boxes, and complex grid fields. The logic for handling and passing events within a container field is the same as for simple fields, it just has more levels of control.

For the purposes of event handling, you can regard the container field as both a field on the parent window, and a window since it contains other fields. In this respect, a container field can have an $event method that handles events for the container field itself, and a $control method that handles events passed to it from the individual fields inside the container field. Each field in the container field has a $event method to handle its own events. If the control method for your container field allows it, events are passed to the parent window control method, which in turn can be passed onto the task control method or discarded as appropriate.

You can nest container fields such as subwindows and tab panes, but nested container fields do not pass events.

Queuing Events

Some user actions generate a single event which is handled as it occurs by your event handling methods. The event may be dealt with completely in the field or it may be passed up the event chain as required. However some user actions generate a whole series of events, one after another. These events are placed in an *event queue*. Each event is handled by your event handling methods strictly in turn on a first-in, first-out basis. For example, when the user tabs from one field to another the current field is sent an evAfter and then an evTab event, then the new field is sent an evBefore event: all these events are placed in the event queue in response to a single user action, the tab. Similarly when you close a window, the current field is sent an evAfter, the window is sent an evCloseBox event, then it is sent an evClose event. Each one of these events is sent to the appropriate object and is handled by your event handling methods *before* the next event in the queue is handled.

In addition to events generated by user actions, you can append an event to the event queue using the *Queue* commands in the *Events...* group.

```
Queue bring to top
Queue close
Queue cancel
Queue set current field
Queue click
Queue double-click
Queue keyboard event
Queue OK
Queue scroll (Left|Right|Up|Down)
Queue tab

Queue quit
```

These commands let you simulate user actions such as key presses and clicks on buttons or windows. For example, the *Queue bring to top {Windowname}* command brings the specified window instance to the top and simulates a user clicking behind the current window. Events generated by these commands are handled after those that are currently queued. You can queue several events in succession.

User Constants

You can define your own *User constants* in a **User Constants** class for use in your methods and expressions. A user constant is a named value, where the value cannot be changed during execution. Generally speaking, user constants can be used anywhere in Omnis code and expressions, although there are exceptions, because they cannot be used anywhere that would attempt to modify them, for example:

- As the result of a Calculate command
- As the Returns component of commands such as Do
- As the dataname of a variable

To define user constants, you add their names and values to a **User Constants** class. The types and therefore values are restricted to Character, Integer, Number and Boolean. You can have multiple user constants classes, each of which defines a number of user constants and their values.

Internally, user constants are handled as a special type of file class, meaning that the same naming rules as those for file classes apply, i.e. $clib.$prefs.$uniquefieldnames, $clib. $prefs.$sensitivefieldnames and $clib.$prefs.$sensitivefilenames all apply. For naming and tokenisation purposes, user constant names are essentially file class variable names.

Also, user constant classes are always treated as memory only, and file class commands such as Clear all files, and Set memory-only files have no affect on user constant class CRBs.

When exporting a library using the JSON export, user constants classes are included, using a similar syntax to that used for file classes.

Creating User Constants

You can create a **User Constants** class this using the **New Class** hyperlink in the Studio Browser, or by Right-clicking in the Studio Browser and selecting the **New Class>>User Constants** context menu option. There is also a class filter that controls whether user constant classes are visible.

The User Constants class editor allows you to define the name, type, subtype, value and description of user constants. User constants can be named however you like, including the prefixes “k” and “ev” which are used for built-in constants and events.

If you try to delete a user constant, the editor will check the current library to see if the constant is in use, and warn you about this.

The **Catalog** lists the user constants under the **User Constants** tab. This has similar behaviour to the Variables tab.

Method Editor and Code Assistant

User constants have a syntax colour and style defined in the “IDEmethodSyntax” group of appearance.json: userconstantcolor and userconstantstyle.

The code assistant default sort order includes user constants at the start of the list, sorted with instance variables, etc.

Also, the option click menu, opened when you right click on a user constant, is a subset of that which applies when you right click on a file class variable.

The Method Editor and other editors in the IDE have validations to prevent user constants from being used where their value could be changed. Similarly, debugger variable windows do not allow user constants to be modified. However, it is impossible for the IDE to detect every such situation, e.g. due to expressions generated at runtime using square bracket notation, so in addition, as a fallback, the low-level code managing the CRB also checks for attempts to modify a user constant, and generates a runtime error if something attempts to do this.

Notation

There is a notation group in $clib, named $userconstants, supporting similar notation to $files. However, user constants classes do not have $conns, $datahead or $indexes members, and user constant objects only have the properties $desc, $ident, $name, $objinitval, $objtype, $objsubtype, $objsublen and $userinfo. $objinitval contains the value of the constant.

Using the class notation for a user constants class is the only way you can programmatically modify the value of a user constant.

The $classtype value for a user constants class is kUserConstants.

Using Tasks

Omnis contains two environments, a *design mode* and a *runtime mode*. In design mode, you can create and store classes in your library. In runtime mode, various objects or instances are created as you run your application. You can group and control the runtime objects in your application by opening or instantiating them in a *task*. You can manipulate whole groups of instances by manipulating their task, rather than having to deal with each separate instance. You define a task in your library as a *task class*, or for web and mobile applications as a *remote task*.

Task classes can contain variables and methods, and you can define custom properties and methods for a task class as well. When you open a task you create an instance of that task class. The task instance is unique in Omnis in that it can contain other instances including *remote form instances* or *report instances* (plus window, toolbar, and menu instances for desktop apps). Task instances cannot contain other tasks. When you open an instance from within a task it *belongs to* or is *owned by* that task.

By opening and closing different tasks, or by switching from one task instance to another, you can control whole groups of objects. Omnis provides certain default actions which happen as the task context switches. You define exactly what happens in the task by creating methods in the task class. For example, in the task in a desktop app you can specify which windows are opened and which menus are installed using commands or the notation.

Each library contains groups of task classes called $tasks or $remotetasks (inside $root.$libs.LIBRARYNAME), and Omnis has a group containing all open task instances and remote task instances, called $root.$itasks or $root.$iremotetasks, listed in the order that they were opened.

Default and Startup Tasks

When you create a new library, it contains a task class called *Startup_Task* by default. For web or mobile apps using Remote forms, you need to create a *Remote_Task* to handle client connections and remote form instances in web and mobile apps: see Remote Tasks. When you start to create your library, and especially if you are prototyping your application quickly, you will not need to create any new tasks, since a lot of the behavior provided by tasks is handled automatically.

When Omnis opens, it creates a task instance for the IDE to run in. This task is called the *default task*, and is represented in the notation as $root.$defaulttask. This task instance contains all the IDE objects such as the Browser, Catalog, Property Manager, and so on.

When you open a library, an instance of the startup task is created automatically. From there on all instances opened in the library are owned by the startup task. You can delete the startup task, or you can create other tasks for your application components to run in.

It is not essential to add tasks to your library, your library will safely execute in the startup task, or the default task along with the IDE objects.

The startup task instance has the same name as your library. For a simple application, the startup task will probably be all you need, with all the other class instances belonging to it. The startup task remains open for as long as the library is open, but you can close it at any time using a command or the notation. You can change the name of the task to be opened on startup by setting the library preference $startuptaskname: for all new libraries this is set to Startup_Task by default.

If you have an application that spans multiple libraries, often only the library used to start the application will have a startup task. If a library is opened using the *Open library* command with the option *Do not open startup task*, the startup task is not instantiated. In design mode, you can stop a library's startup task from running if you hold down the Alt/Option key as you open your library.

Handling Application Focus Events

You can control application focus events generated by the Operating System using the task methods $omnistofront and $omnistoback. These methods can be added to the Startup task in your library.

- **$omnistofront**
  called when Omnis is brought to the front by the Operating System as a result of a user action such as a mouse click or alt-tab key combination. On macOS, an extra boolean parameter is passed to $omnistofront. If the value of this parameter

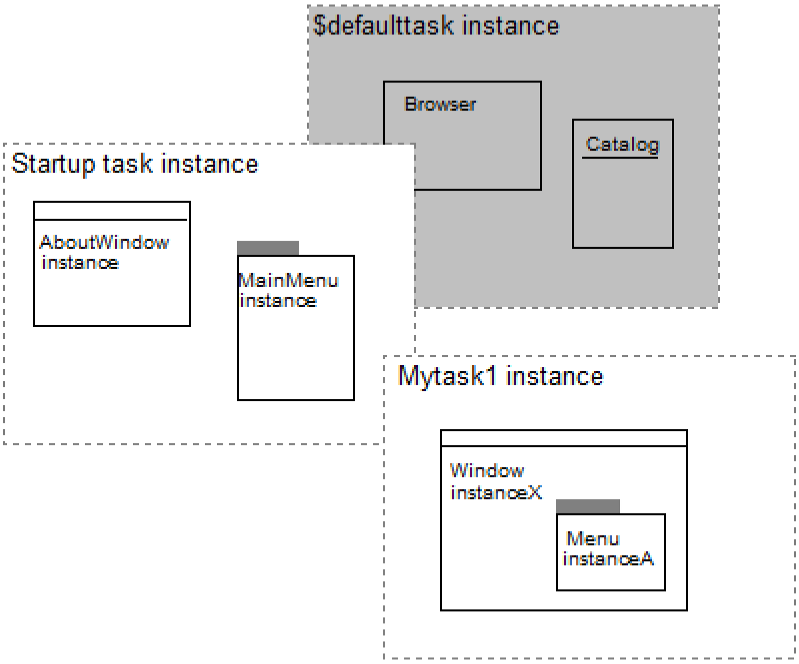


Figure 76:

is kTrue, the user has clicked on the Omnis Icon in the Dock. If a user clicks on the Omnis Dock Icon, $omnistofront will be called twice. The first call will be generated as a result of an OS AppActivated event and the parameter value will be kFalse. The second call will be generated as a result of the Dock click and the parameter value will be kTrue.

- **$omnistoback**
  called when Omnis is sent to the back by the Operating System as a result of a user action such as a mouse click or alt-tab key combination. For $omnistoback you should only use Omnis code which produces a non-visual result. Attempting to open windows, and so on, while Omnis is losing the application focus may have undesirable affects and is not supported.

Main Window Resize Message (Windows only)

The task message **$mainresized** is called when the main Omnis application window has been resized on the Windows platform (it does not apply on macOS). $mainresized has two parameters, pWidth and pHeight, which are the dimensions of the available area of the main window (excluding any docking areas if present). When the main window is minimized the parameters are both zero. You could use the Width and Height dimensions to readjust the layout of end user windows based on the available area of the application window.

In addition, the sys functions sys(251) and sys(252) return the width and height of the available area of the main window, respectively.

Creating Task Classes

This section describes how you create a task class from the Browser.

To create a task class

- Open your library and select it in the Browser
- Click on New Class and then the Task option
- Name the new task
- Double-click on the task class to modify it

You modify a task class in the method editor. You can place in the $construct method any code that you want to run when the task is opened. For the Startup_Task, the $construct method is executed when you open your library. You can add any other custom properties and methods to the task, as well as any type of variable.

Opening Tasks

Apart from the startup task instance, which is opened automatically when you open your library, you can open a task using the *Open task instance* command or the $open() method. Any parameters you supply with the command are sent to the task's $construct method.

```
Open task instance MyTask/TaskInstance2 (p1,p2,...)

# opens the task, assigns an instance name, and sends parameters
```

Alternatively you can open a task instance using the $open() method.

```
Do MyTask.$open('TaskInstance2',p1,p2,...) Returns iTaskRef

# does the same as above & returns a reference to the task instance
```

Current and Active Tasks

Omnis keeps references to two different tasks, the *active task* and the *current task,* to keep track of the tasks that own the topmost instance or GUI object and the currently executing method. The active task is the task that owns the topmost open window, installed menu, or toolbar currently in use. The current task is the task that owns the currently executing method.

A *task context switch* occurs when Omnis changes the current or active tasks. As Omnis runs your library, the current and active tasks may point to different task instances depending on the user's actions.

The Active Task

The active task is affected by the user, and is typically the task containing the topmost open window. When an instance belonging to another task is selected, Omnis performs a task context switch. As part of the context switch, messages are sent to both tasks. The active task gets sent a $deactivate() message, and the new active task is sent an $activate() message.

When the active task changes, you can use the $activate() and $deactivate() messages to perform other relevant actions such as hiding other windows, installing menus, and any other requirements your application has.

In order for Omnis to perform an automatic task context switch when the user selects an instance belonging to another task, the task's $autoactivate property must be set to kTrue.

Omnis can install and remove menus and toolbars automatically during task context switches. Menu and toolbar instances each have a $local property that you can set. When set to true, the menu or toolbar instance is made local to the task that owns it. When a task context switch occurs, local menus for the previously active task will be removed from the menu bar, and any local menus instances owned by the new active task will be installed. Toolbars behave similarly. If the tasks use different docking areas, Omnis will not hide the docking areas, only the toolbars.

You can change the active task using the notation, rather than waiting for the user to initiate a task context switch. To do this, you can set the property $root.$activetask to a different task instance name to switch tasks.

The Current Task

The current task is under the control of Omnis itself, and is the task instance which contains the currently executing method. When a custom attribute or event is sent to an instance, the current task is switched to the task which owns the instance, and when control returns from that attribute or event, the previous task is restored.

When the current task changes, messages are sent to both tasks. The current task is sent a $suspend() message, and the new current task gets a $resume() message. If the new current task is being instantiated for the first time, it gets a $construct() message rather than the $resume().

In order to avoid endless recursion a task does not get suspend or resume messages during the execution of a suspend or resume method.

Since $suspend() and $resume() are likely to be called frequently, it is important that the code for them should be kept as short and efficient as possible and should not:

- alter the user interface
- open or close an instance
- switch tasks

You can find out the name of the current task using the notation $ctask().$name, and the task that owns the instance by using InstanceName.$task().$name.

Closing Tasks

You can close a task instance using the *Close task* command or the $close() method. When you close a library all its task instances are closed, and when you quit Omnis the default task is closed and all instances belonging to the default task are closed.

When you close a task, all instances belonging to that task are closed or destructed providing they can be closed. When instances are closed, a message is sent to the instance asking it to confirm whether or not it can be closed. If the instance returns a false message, Omnis will not close that instance. For tasks, each instance belonging to the task is sent the message, and then the task itself is sent the message. If any of the instances belonging to the task cannot be closed, none of the instances nor the task instance are closed.

Task Variables

Task classes can contain both class and instance variables of any standard Omnis data type. Tasks can also contain *task variables*, which are accessible to any instance owned by the task. As with other variables, you create task variables in the variable pane of the method editor.

When two or more types of variable use the same variable name (this is not recommeded), a reference to that variable may be ambiguous. In this situation, Omnis uses the variable with the smallest scope automatically. All other variable scopes have precedence over task variables.

When a method in a code class is called from another class or instance using the *Do code method* command, the current task continues to point to the calling method. This allows methods in a code class to have access to the task variables from the calling method.

The Design Task

In order for task variables to be available to you for use in design mode, you must establish a connection between a class and the task whose variables you want to access. You do this by setting the design task ($designtaskname property) for the class. The design task determines which task variables are available to the class: if no design task has been set, the method editor does not let you declare or see any task variables.

Setting the design task for a class doesn't guarantee that the task will be available in runtime when you open your class, nor will Omnis automatically create an instance of the task. The design task is simply a way to give you access to a set of task variables while you create the classes in your library.

You can also access task variables without setting a design task by referring to the variable as $ctask.variablename. This assumes that the variable will always belong to a task and can therefore default to the current task.

If you attempt to access a task variable in an instance, and that variable is not available in the task, a runtime error of 'Unrecognized task variable' will be generated, and the variable will have a NULL value.

If you rename a task variable, any references to it are not renamed. Also if one with that name ceases to exist, references to it which were entered as VariableName are shown in design mode as $ctask.VariableName. Similarly, if some code containing a task variable is pasted into a different class, any task variables used by that code are not copied into the destination class.

Private Instances

Normally an instance is visible to other tasks and you can reference it using the notation from anywhere in your library. However you can override this default behavior by making an instance *private* to the task that owns it. You can do this by setting the instance's $isprivate property to kTrue.

When you make an instance private, you cannot destruct it, make references to it, or even see it unless you are within the task that owns it. A task can even be private to itself, so it can be closed only when it is the current task. If access to a private instance is required from outside of the task, an item reference can be set to the instance, and the item reference can be passed outside of the task. Once this has occurred, the item reference can be used to manipulate the instance.

The $root object has a series of object groups, one for each instance type, that are represented in the notation as $iwindows, $imenus, $itoolbars, $ireports, $itasks. Each of these object groups displays all public instances, as well as instances which are private to the current task. As the current task changes, the contents of these groups may change to reflect the private instances present in your library.

Private Libraries

Libraries can be private to a task, and both the library and its classes are visible only to that task.

The group of open libraries, $libs, contains a private library only when the task which owns that library is the active task. The Browser does not display classes from a private library. Standard entry points to the debugger such as shift-click on a menu line do not enter the debugger if the menu belongs to a private library.

As with private instances, if an item reference to any object within a private library is passed to an object outside the library, it is able to access the library using notation.

You can make a library private by setting its $isprivate property to true. This is normally done immediately after opening the library, but can be done at any time as long as the task which owns the library is the active task. Libraries also have the $alwaysprivate property, which, if set, means they are always and immediately private to their startup task.

Private libraries have an additional property, $nodebug, which keeps the debugger from being entered for any reason when code from that library is executing, including errors, breakpoints, and the stop key. Code from a private library with $nodebug set does not appear in the stack menu or the trace log.

When a task is closed, it closes all its private libraries unless they cannot be closed. This can occur if, for example, the library has instances belonging to other tasks. If a private library cannot be closed, it will become non-private.

Multiple Tasks

When designing an application, you might want to partition your library by creating modules containing all of the windows, reports and methods of like functionality. Each module can have its own menus and toolbars. An example containing such modules might be an accounting package, with General Ledger, Accounts Payable and Accounts Receivable modules.

In a totally modal application, where the user switches between modules, it is easy to ensure that the user sees the correct menus and tools for the current module. In a modeless, multi-window environment, controlling this can sometimes be difficult. Tasks automate the process of creating modular applications by providing all the management of menus and tools for you.

Consider the following example in which a single library is running three tasks: the *Startup_Task* and two user tasks *Task1* and *Task2*. The startup task, which opens automatically when the library opens, contains an About window. The other two tasks each contain a window, a menu, and a toolbar. When the user selects a window from either Task1 or Task2, you may want Omnis to display the correct tools and menus for that window automatically.

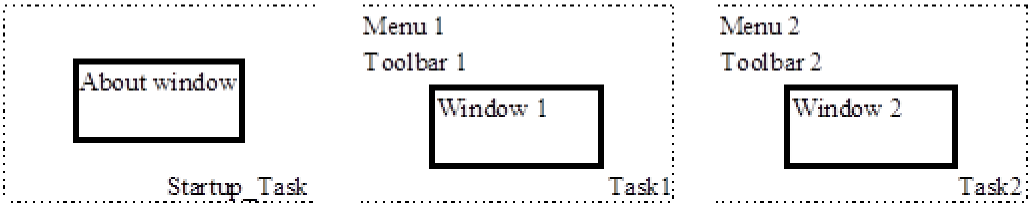


Figure 77:

When the library opens, the startup task opens and displays the About window and then opens the other tasks, each of which opens its window and installs its menu and toolbar. The startup task can close itself once the About window is closed if it's no longer needed.

To open the two tasks, you should execute the following in the $construct method of the startup task

```
Open window instance AboutWindow
Open task instance MyTaskClass1/Task1
Open task instance MyTaskClass2/Task2

Close task instance LibraryName ## close Startup_Task instance
```

Every task has a property $autoactivate, that allows the task to take control whenever the user tries to bring a window it owns to the front. If the property is set to false, the window won't come to the front. To activate each task automatically, you need to execute the following in the $construct of each task

```
Do $ctask.$autoactivate.$assign(kTrue)
```

To ensure that your menus and toolbars show and hide appropriately as the tasks change, you need to set the $local property for each class. By making each menu and toolbar local to the task that owns it, Omnis hides and shows them automatically as the task context changes.

In the $construct for a task, you can install your menu and toolbar, and set their $local property. For example

```
# $construct for task1...
Do $menus.MyMenuClass1.$open('Menu1') Returns iMenuRef
Do iMenuRef.$local.$assign(kTrue)
Do $toolbars.MyToolbarClass1.$open('Toolbar1') Returns iToolRef
Do iToolRef.$local.$assign(kTrue)

Do $windows.MyWindowClass1.$open('Window1') Returns iWinRef
```

You can do the same for the other task.

```
# $construct for task2...
Do $menus.MyMenuClass2.$open('Menu1') Returns iMenuRef
Do iMenuRef.$local.$assign(kTrue)
Do $toolbars.MyToolbarClass2.$open('Toolbar1') Returns iToolRef
Do iToolRef.$local.$assign(kTrue)

Do $windows.MyWindowClass2.$open('Window1') Returns iWinRef
```

This functionality will change menus and toolbars as you switch from one window to the other.

Preferences on macOS

The macOS application menu has a Preferences item. You can arrange for Omnis to call a method in a task, when the user selects this menu item. To do this, define a method called $prefs in your task. When the user selects Preferences from the application menu, Omnis calls the $prefs method.

If more than one task defines $prefs, Omnis installs a hierarchical menu on the Preferences menu item. Each task has an item in the hierarchical menu. In this case, each task must also contain a method called $prefsmenutext. This method must return the text to display for the task's item in the hierarchical menu, for example

```
Quit Method "My Library"
```

External Component Notation

The $components group under $root contains all the installed external components available in your XCOMP folder. You can view the contents of the $components group using the Notation Inspector.

Note that you manipulate an external component via its custom field properties, as shown below, not via the $root.$components...$comppro or $compmethods groups for the control. The groups under $root.$components is simply a convenient way of viewing the contents and functions of any external library or control.

The $components group has the standard group properties and methods, including $add() and $remove(), and you can list the components using the $makelist() method.

```
# declare variable cCompList of type List
Do $root.$components.$makelist($ref.$name) Returns cCompList
```

You can drag a reference to any of the components from the Notation Inspector to your code, in the same way as other built-in objects. You can click on a component library in the Notation Inspector and view its properties in the Property Manager. Each component library has the following properties

- $name
  the name of the component which must be unique
- $pathname
  the name and path of the external library file: this will vary across different platforms
- $functionname
  the name of the external function
- $controlhandler
  Boolean that indicates whether the external is a control handler, for example, an ActiveX is a control handler

- $constprefix
  String used as a prefix for all constants within the external
- $flags
  indicates the external flags, for example, whether it is loaded
- $usage
  Current number of controls that are using this external
- $version
  the version information

You can view the contents of an external library in the Notation Inspector. Each component library has a group called $controls containing all the controls in the library. Some libraries may contain only one control, for example, the Slider Component Library contains the Slider Control only. A control contains its own events, functions (or methods), and properties in their own respective groups, as follows

- $compevents
  group of events for the control
- $comprops
  group of properties for the control
- $compmethods
  group of methods for the control

In the notation you treat an external component property or function as you would a standard built-in property or method, that is, you can use property and method names in the notation to manipulate and send messages to an external component field. Note that property and method names should include a dollar sign when you use them in the notation.

```
Do $cwind.$objs.ClockField.$facecolor.$assign(kBlue)
# assigns a color to the face of a clock component
# using the $facecolor property
Do $cwind.$objs.QTfield.$Play()
# executes the $Play() function for a QuickTime component
```

In general, the properties of an external component are unique to the object and their names will not clash with standard Omnis field properties. However when an external component property has the same name as an Omnis property, you must access the external property using a double colon (::) before its name. For example, the Icon Field control has the property $backcolor which you must refer to using

```
Do $cinst.$objs.iconfield.$::backcolor.$assign(kRed)
# would not work without the ::
```

At runtime you can add an external component to an open window using the $add() method. You need to specify the kComponent object type, external library name, external control name, and the position of the field. For example, the following method adds the Marquee Control to the current window instance, positions the new object, and sets some of its properties

```
# declare local variable Objref of type item reference
Do $cinst.$objs.$add(kComponent,'Marquee Library','Marquee Control',0,0,15,100) Returns Objref
Do Objref.$edgefloat.$assign(kEFposnStatusBar)
# repositions the object at the bottom of your window
Do Objref.$message.$assign('I hope you can read quickly!')
Do Objref.$steps.$assign(20) ## number of pixels to step
Do Objref.$speed.$assign(20) ## lower number is faster
Do Objref.$::textcolor.$assign(kBlue) ## note :: notation
Do Objref.$::backcolor.$assign(kRed)
```

Version Notation

All external components have the $version property. To get the version of an external component you must access it via the $root.$components group, not the external component field on a window or report. For example

```
Do $root.$components.Marquee Library.$version Returns lvXversion
# returns "1.2" for example
```

If you have created any external components of your own to run under Omnis Studio version 1.x, you must recompile them for Omnis Studio 2.0.

Java Beans

Note the Java Bean ext comp is not longer installed in Studio 10 or above, but it can be obtained by contacting support.

The Java Bean external component has commands that let you control it in a Runtime Omnis. Note the Java Beans external is available under Windows only.

You request a command using the $cmd() method as follows:

```
$root.$components.JavaBean.$cmd(parameter list)
```

The parameters can be:

Parameter list	Command
"GetPaths", List	Populates the specified single column list with the Java Bean search paths: no return value
"AddPath", NewPath	Adds the specified path to the Java Bean search paths: returns true for success, or if the path is already present in the search paths
"DeletePath", DelPath	Deletes the specified path from the Java Bean search paths: returns true for success
"EnumBeans"	Enumerates Java Beans: returns the number of Beans found
"StartVM"	Starts the Java virtual machine (to test if Java is installed): returns a string containing an error, or an empty string to indicate success
"SetupDialog".	Opens the Java Bean component setup dialog
"RequestPath"	Opens the "Prompt for Java Bean Path" dialog: returns a string containing the new path: empty if none selected

Chapter 4—Debugging Methods

You add variables and methods to the classes and objects in your library using the **Method Editor** which includes a free-type **Code Editor** to allow you to write Omnis code faster and more easily.

You can debug the methods and step through the code in your library using the Omnis **Debugger,** which is an integral part of the method editor, plus you can debug your Omnis code remotely over a network using the Remote Debugger.

The Omnis debugger provides several tools to help you monitor the execution of a method, including the ability to create watch variables, interrogate and edit the contents of variables during execution, and place a variety of breakpoint conditions, which when met will interrupt execution.

The debugger operations are controlled from the Debug and Options menus on the method editor menubar. The debug options are also on the toolbar, which you can show using the View>>Toolbar menu option. The hierarchy of methods calling other methods is saved in the *method stack* and shown on the Stack menu.

You can also check your code using the Method Checker, available under the Tools menu and described in this chapter.

Note that most of the example code in this chapter is generic and can be applied to all programming tasks; however, some of the example code may relate to window classes only, but the code may be easily adapted to work with remote forms.

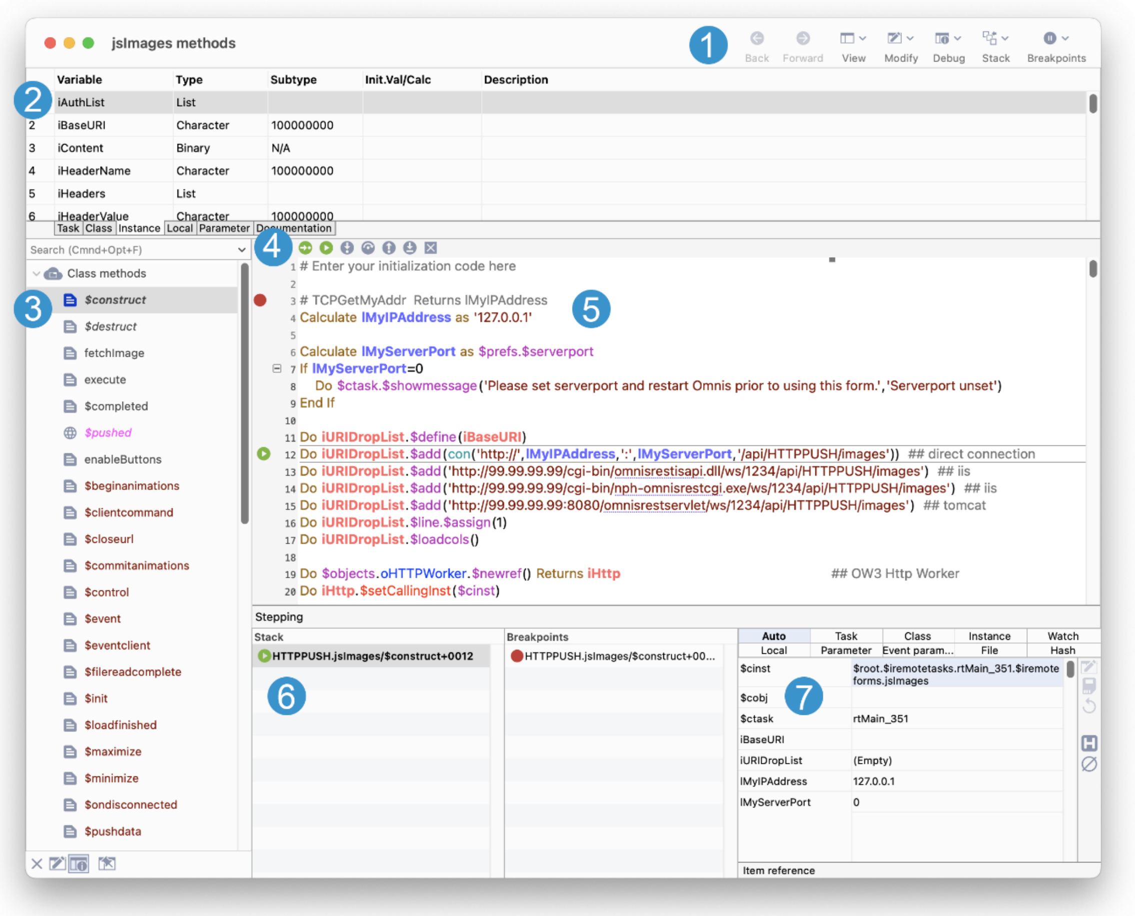
Method Editor

The **Method Editor** is the main tool you use for programming or coding in Omnis Studio. Using the Method Editor and Debugger, you can:

- Insert and Edit methods using the Code Editor and Code Assistant
- Run and step through methods using the Debugger
- Set Go points and Breakpoints
- Trace the execution of method lines and field values
- View and alter fields and variables
- Inspect the method stack

- Debug your live code, sending commands to the Trace log
- Debug your code remotely on your live app server: see Remote Debugger

The Method Editor has several different areas, each doing a different job, as described below.



	Variable	Type	Subtype	Init.Val/Calc	Description
	iAuthList	List			
2	iBaseURI	Character	100000000		
3	iContent	Binary	N/A		
4	iHeaderName	Character	100000000		
5	iHeaders	List			
6	iHeaderValue	Character	100000000		

```
# Enter your initialization code here

# TCPGetMyAddr  Returns lMyIPAddress
Calculate lMyIPAddress as '127.0.0.1'

Calculate lMyServerPort as $prefs.$serverport
If lMyServerPort=0
  Do $ctask.$showmessage('Please set serverport and restart Omnis prior to using this form.','Serverport unset')
End If

Do iURIDropList.$define(iBaseURI)
Do iURIDropList.$add(con('http://',lMyIPAddress,':',lMyServerPort,'/api/HTTPPUSH/images'))  ## direct connection
Do iURIDropList.$add('http://99.99.99.99/cgi-bin/omnisrestisapi.dll/ws/1234/api/HTTPPUSH/images')  ## iis
Do iURIDropList.$add('http://99.99.99.99/cgi-bin/nph-omnisrestcgi.exe/ws/1234/api/HTTPPUSH/images')  ## iis
Do iURIDropList.$add('http://99.99.99.99:8080/omnisrestservlet/ws/1234/api/HTTPPUSH/images')  ## tomcat
Do iURIDropList.$line.$assign(1)
Do iURIDropList.$loadcols()

Do $objects.oHTTPWorker.$newref() Returns iHttp                              ## OW3 Http Worker
Do iHttp.$setCallingInst($cinst)
```

Stack

HTTPPUSH.jsImages/$construct+0012

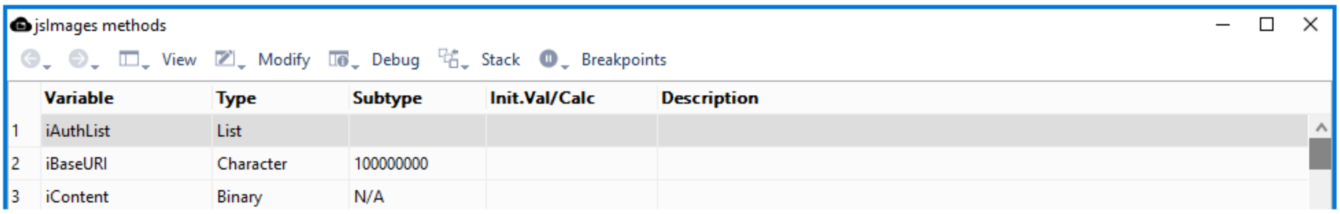
Breakpoints

HTTPPUSH.jsImages/$construct+00...

$cinst	$root.$iremotetasks.rtMain_351.$iremoteforms.jsImages
$cobj	
$ctask	rtMain_351
iBaseURI	
iURIDropList	(Empty)
lMyIPAddress	127.0.0.1
lMyServerPort	0

Figure 78:

1. Toolbar
the main toolbar gives you access to the **View, Modify, Debug, Stack** and **Breakpoints** menus; many of the options in these menus have keyboard equivalents to give you hands-free coding. The **Back** and **Forward** options let you jump back to or from called methods as you step through code.
On macOS the toolbar appears in the window title bar on the right (as above), but on Windows it appears under the window title on the left (see below); this is the only visual difference, but in all other cases the appearance and functionality is the same on macOS and Windows

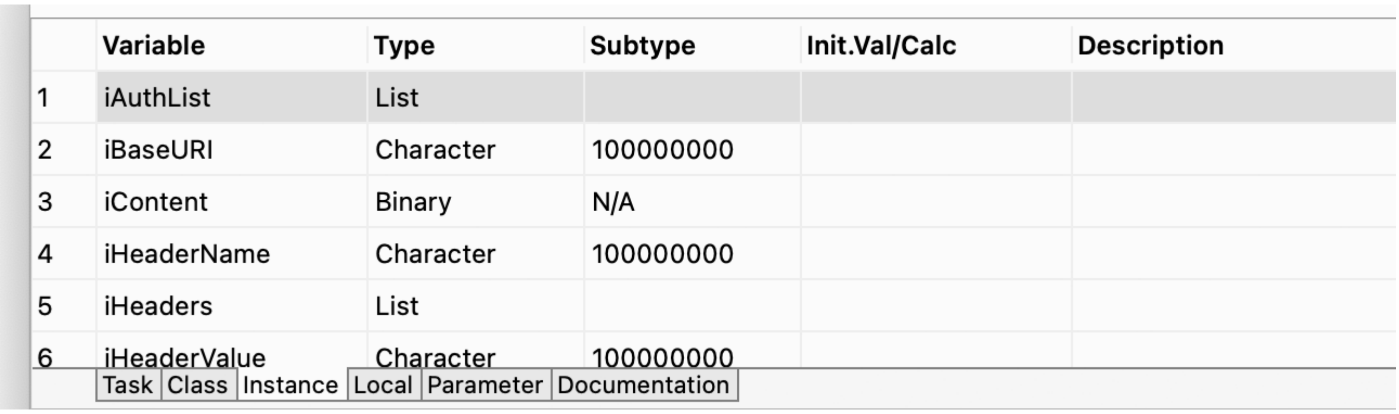


	Variable	Type	Subtype	Init.Val/Calc	Description
1	iAuthList	List			
2	iBaseURI	Character	100000000		
3	iContent	Binary	N/A		

Figure 79:

2. Variables pane
lets you add variables to the class or method, including task, class, instance, local, or parameter variables; you can hover over a variable to see its value (if available), or you can Right-click on a variable to set other option. You have to Right-click on blank space in the list or in the left margin to insert a New Variable

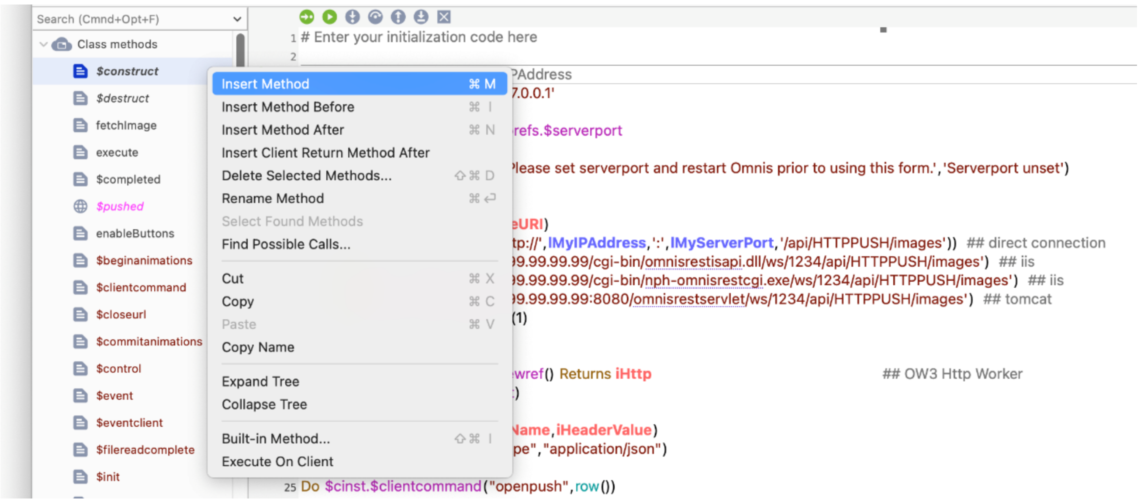
3. Method list or Method name list
lists existing methods for the class or object, or lets you add methods to the object,: the method tree list shows the methods for the class under **Class Methods,** or for UI classes it also shows the methods for the individual controls or fields in the class, listed after the class methods. You can Right-click on the method tree list to insert a new method, delete or rename a method, Cut



	Variable	Type	Subtype	Init.Val/Calc	Description
1	iAuthList	List			
2	iBaseURI	Character	100000000		
3	iContent	Binary	N/A		
4	iHeaderName	Character	100000000		
5	iHeaders	List			
6	iHeaderValue	Character	100000000		

Figure 80:

or Copy a method, or Expand/Collapse the entire method tree. When the focus is on a method name in the method editor tree, pressing Return or Enter lets you edit the method name

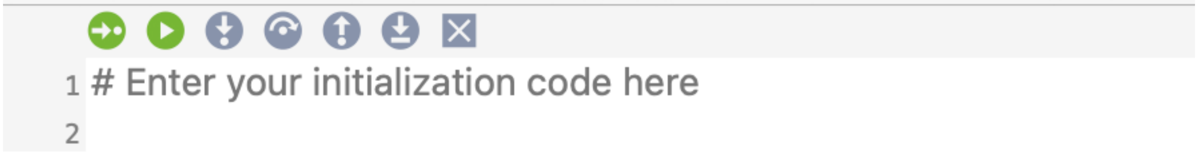


```
# Enter your initialization code here

PAddress
7.0.0.1'
refs.$serverport
lease set serverport and restart Omnis prior to using this form.','Serverport unset')
eURI)
tp://',lMyIPAddress,':',lMyServerPort,'/api/HTTPPUSH/images'))  ## direct connection
99.99.99.99/cgi-bin/omnisrestisapi.dll/ws/1234/api/HTTPPUSH/images')  ## iis
99.99.99.99/cgi-bin/nph-omnisrestcgi.exe/ws/1234/api/HTTPPUSH/images')  ## iis
99.99.99.99:8080/omnisrestservlet/ws/1234/api/HTTPPUSH/images')  ## tomcat
(1)
wref() Returns iHttp                                      ## OW3 Http Worker
)
Name,iHeaderValue)
pe","application/json")
Do $cinst.$clientcommand("openpush",row())
```

Figure 81:

4. Debug toolbar

appears at the top of the Code Editor panel allowing you to Set the Go point, Go (execute the method), Step in, Step over, Step out, Step To Line, and Clear the method stack



```
# Enter your initialization code here
```

Figure 82:

5. Code Editor

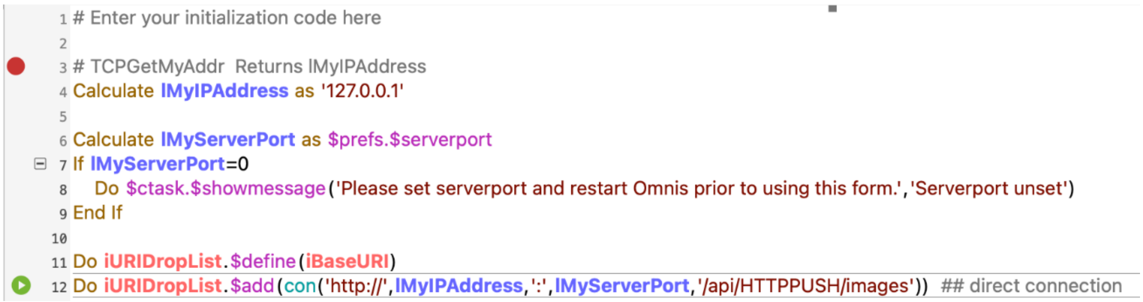
lets you add the code for a method, or view and edit the code for an existing method: there is a Help panel, at the bottom of the code editor pane, showing the syntax for the current selected Omnis command: see below for a complete description of the Code Editor and Code Assistant

6. Debugging and Breakpoint panels

this area displays the Method stack and list of Breakpoints currently set, which are redrawn dynamically as you debug and step through your code and jump from one method to another

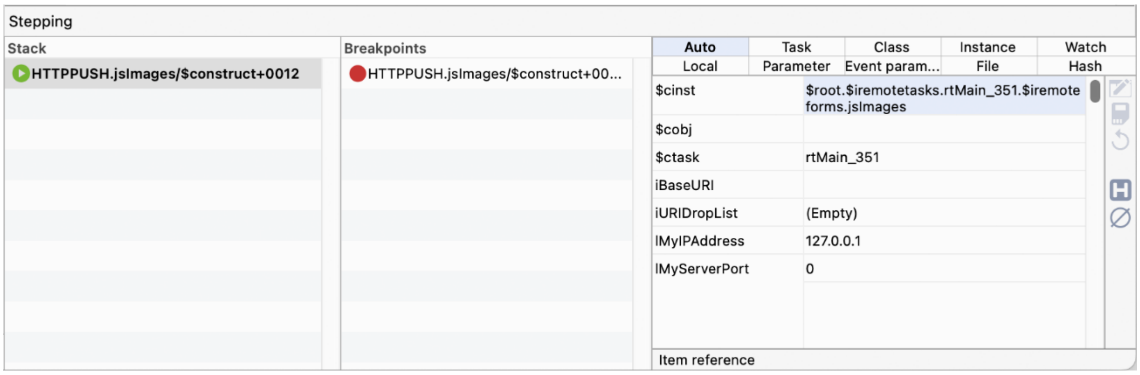
7. Variable panel

The Variable panel (on the right, see above) allows you to view and modify variables *while you debug and step through your code*; as execution pauses, it displays the current values of all the variables, as well as the Watched variables, and you can drill down into the hierarchy of objects and variables



```
# Enter your initialization code here

# TCPGetMyAddr  Returns lMyIPAddress
Calculate lMyIPAddress as '127.0.0.1'

Calculate lMyServerPort as $prefs.$serverport
If lMyServerPort=0
   Do $ctask.$showmessage('Please set serverport and restart Omnis prior to using this form.','Serverport unset')
End If

Do iURIDropList.$define(iBaseURI)
Do iURIDropList.$add(con('http://',lMyIPAddress,':',lMyServerPort,'/api/HTTPPUSH/images'))  ## direct connection
```

Figure 83:



HTTPPUSH.jsImages/$construct+0012

HTTPPUSH.jsImages/$construct+00...

$cinst	$root.$iremotetasks.rtMain_351.$iremoteforms.jsImages
$cobj	
$ctask	rtMain_351
iBaseURI	
iURIDropList	(Empty)
lMyIPAddress	127.0.0.1
lMyServerPort	0

Figure 84:

Opening the Method Editor

You can open the method editor in a number of ways, depending on the type of object you're working on and where you are in Omnis.

To open the method editor for a class

- Open your library and view its classes in the Studio Browser
- Right-click on the class in the Studio Browser list
- Select the **Class Methods** option from the context menu

To open the method editor for a Remote Form

- Open the Remote Form from the Studio Browser
- Click on the **Methods** button in the Design bar at the top of the remote form editor

Or you can

- Right-click on the background of the Remote Form
- Select the **Class Methods** option from the context menu

Or you can

- Double-click on the background of the Remote Form to open the Method Editor for the class

To open the method editor for a Report

For a Report class (or window, menu, and toolbar classes) you can:

- Open the report editor for the class (or window, menu, or toolbar editor) from the Studio Browser
- Right-click on the background of the class design screen
- Select the **Class Methods** option from the context menu

- To open the method editor for a field or object
- Open the remote form class (or window) from the Studio Browser
- ***Double-click*** on the field or object to open the Method Editor

Or

- Right-click on the field or object, e.g. a button
- Select the **Field Methods** option from the context menu

Task, remote task, table, remote object, object and code classes can only contain methods, so when you modify (double-click) these classes you go straight into the method editor.

Inserting and Editing Methods

You can insert, edit and debug the methods in your library using the ***Method Editor.***

Inserting a Method

To insert a method

- Right-click on the method list and select **Insert Method, Insert Method Before,** or **Insert Method After**

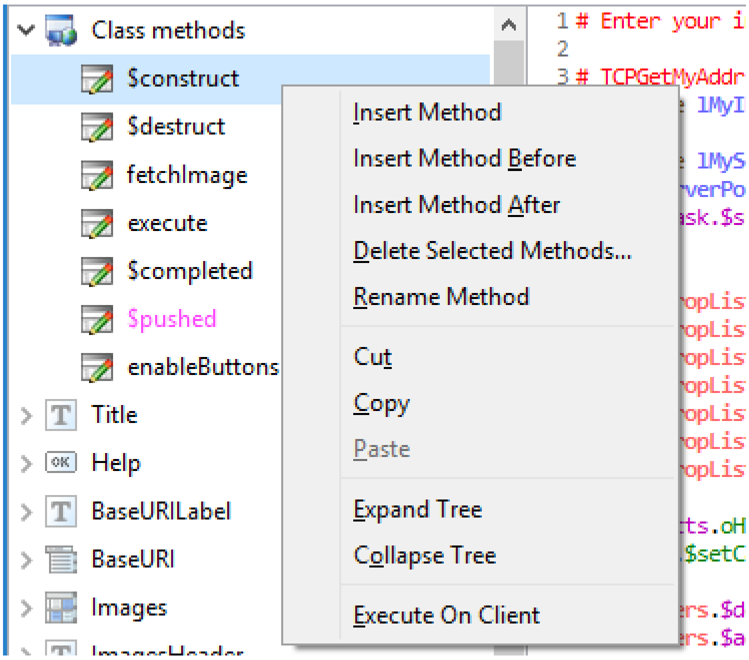


Figure 85:

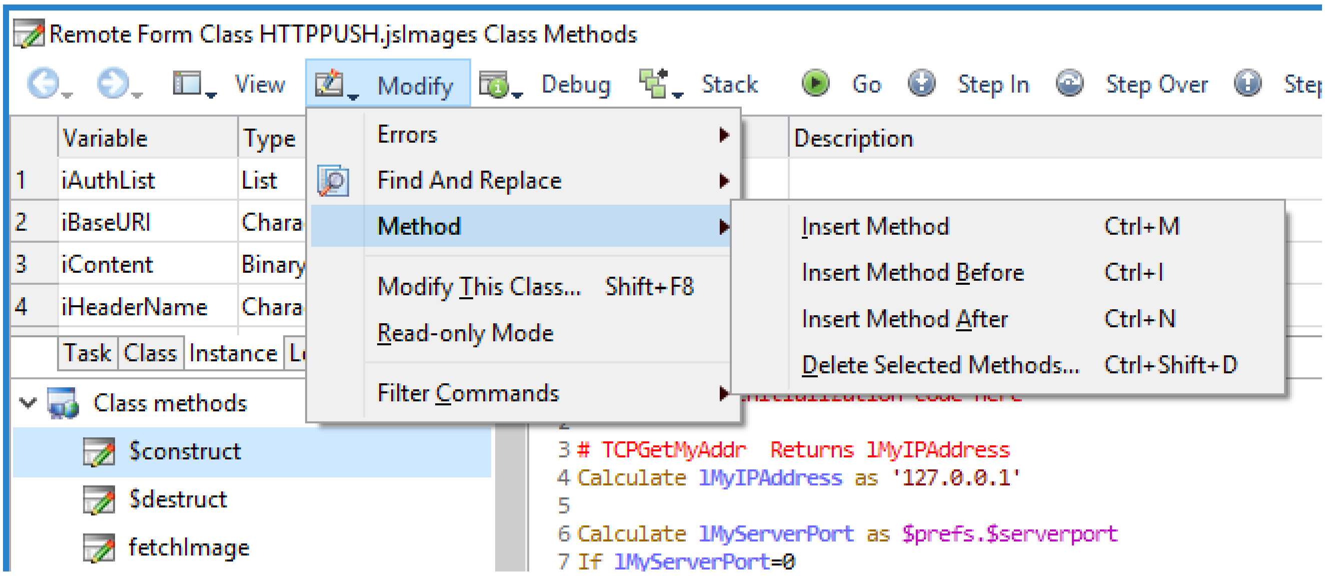
or

- With the focus on a method in the Method list, click on the **Modify** menu in the method editor toolbar and select **Insert New Method**

or

- Press Ctrl/Cmnd-I while the focus is on the method list: this option inserts a method above the current one

When you have entered the new method name, you can begin to add the code for the method in the **Code Editor** on the right-hand side of the method editor.



	Variable	Type	Description
1	iAuthList	List	
2	iBaseURI	Chara	
3	iContent	Binary	
4	iHeaderName	Chara	

```
# TCPGetMyAddr  Returns lMyIPAddress
Calculate lMyIPAddress as '127.0.0.1'

Calculate lMyServerPort as $prefs.$serverport
If lMyServerPort=0
```

Figure 86:

Maximum Number of Methods

The maximum number of methods allowed per class is 4096 (the limit was 501 in versions prior to Studio 10.2).

Line Numbers

You can display a line number for each method line in the Code Editor. You can enable line numbers using the View menu.

Showing Inherited Methods First

The **Show Inherited Methods First** option in the View menu allows you to display inherited methods at the top of the methods list in the Code Editor; the option defaults to off which means inherited methods will be shown after all other methods at the bottom of the list.

In addition, the remote debug server configuration has a new option (Show inherited methods first in method lists) which controls the information returned by the server to the client, and therefore the display in the remote debug window. The remote debug server dialog has been updated to allow this option to be edited.

The F8 shortcut works for inherited methods. So if you press F8 on the code line Do $inherited.$test() it will load the $test method in the inherited class.

Showing Built-in Methods

The **Show Built-in Methods** option allows you to hide or show the built-in class methods. When enabled (the default), the class methods node in the method name list includes the *Built-in methods* for instances of the class type being edited, including $control, $construct, and $destruct, as well as any *Control methods* that can be overridden, including $event. This applies to Remote forms and Remote tasks (plus other classes that can be instantiated including window classes). In addition, $canclose will be shown for the relevant instance types, while $select and $fetch are shown for table classes. Many other methods could be shown depending on the class type, including $filereadcomplete, $init, $term, $sfsorder, $sfscanclose, $pushed, $sqldone, $suspended, $resumed, $loadfinished, $previewurlclicked, $pdfcomplete.

The Built-in and Control methods behave in a similar way to inherited methods, that is, you can override them, or set them back to using the default, by using "Built-in Method..." option from the menus (this is analogous to using Inherit Method... for an overridden inherited method). When you override a Built-in or Control method, Omnis pre-defines the parameters of the new method to match those required by the method.

The names of the Built-in or Control methods are shown in the tree using the no set property color (this is consistent with how built-in method names are drawn in the Interface Manager).

The "overriddenbuiltinmethodstyle" color theme member can be used to give the name of an overridden method a different text style when it is shown in the tree. This theme member is in the IDEmethodEditor group of the appearance.json file, and can have the same possible values as "overriddenmethodstyle"; it defaults to 2 (italic).

Overriding or Inheriting multiple methods

You can multi-select the methods from the Methods list or a single object and then override or inherit the methods, as appropriate, from the context menu or the **Modify** menu. If all selected methods are built-in or inherited, they can be overridden with a single override method command on the context menu, or the Modify menu.

If all selected methods override inherited methods, they can be deleted and re-inherited using a single inherit method command on the context menu, or the Modify menu.

If all selected methods override built-in methods, they can be deleted and set back to built-in using a single built-in method command on the context menu, or the Modify menu.

In addition, the options **Inherit variables…** and **Override variables…** have been added to the Variable panel context menu to allow multiple variables to be inherited or overridden in a single operation. These commands are only present for subclasses, for task, instance and class variables.

Find Possible Calls

The **Find Possible Calls…** option in the Context menu on the Method list attempts to locate all possible calls to the method, from methods in the current library, or a selected set of open libraries. If there is only one open library, it performs the search immediately, displaying a progress bar. If there is more than one open library, the option opens a popup dialog that allows you to select the libraries to be searched, and then performs the search, displaying a progress bar.

The option writes results to the Find and Replace log, and then opens the Find and Replace window when it has completed the search. Note that the option does not search calculations stored with objects, it only searches method lines.

Calls located may not be actual calls to the method, since for example calls like item.$method cannot be resolved, so if the call occurs in the correct class (or in the inheritance hierarchy), the call will be treated as found.

Method Search

There is a **Search** or filter option above the method name list that allows you to find specific named methods, or methods that start with or contain specific characters. As you type in the search box, the method list updates automatically to highlight the method names that match or contain the search (in currently expanded nodes only). These lines draw in the color treelinesmatchingsearchcolor in the IDEMethodEditor section of appearance.json. The first matching method for the search is selected and its contents is shown in the Code Editor. The "Show Method Tree Search Box" option on the View menu allows you to toggle the method search box (the default is enabled). The state is saved with the window setup.

The search also includes *Object names* (as well as method names) to allow you to locate controls and other objects in the method tree list, such as containers or text labels, in order to display and edit their methods. The setting for this search behavior is stored in a new item 'includeObjectNodesInTreeSearch' in the 'methodEditorAndRemoteDebugger' group in the config.json file (true by default).

While the search box has the focus, you can use the find and replace menu of the method editor (or its find next and find previous shortcuts) to select the next or previous matching method. There is also a context menu item for the method list called "Select Found Methods", which selects all matching methods. There is a menu option 'Search Method Tree' on the Find and Replace menu that puts the cursor in the method search box, which also has a keyboard shortcut named "searchMethodTree" that appears in keys.json - note that the default disable all breakpoints shortcut has changed as a result of this change.

The saveSearchDelay item (previously savePropertySearchDelay) in the ide section of the config.json file allows you to set the delay between typing and the search being executed.

Adding Blank Method Lines

When the focus is on a method line, the **Append blank lines** menu option adds blank lines to the end of the current method and sets the current line to the first added blank line: you can also use the shortcut Ctrl/Cmnd-B when the focus is on the code editing area in the current method. This option behaves in the same way as clicking on the dead space at the end of a method in the method editor (shown in gray), but gives you the option to do this from the Modify menu, or from the keyboard using the shortcut key. Note that when you click away from the method, any blank lines at the end of the method are omitted automatically.

Configuring Blank Method Lines

Omnis adds space for 64 method lines, but you can change this to any value from 1 to 128 inclusive by editing the "methodEditor" section in the Omnis configuration file (config.json): the following entry is at the same level as the "server" entry.

```
"methodEditor": {
  "stripTrailingEmptyCommands": true,
  "blankLinesToAdd": 64
}
```

When the method editor saves a method back to the class, that is, as it is being navigated away from, Omnis strips empty method commands from the end of the method. You can disable this behavior by editing config.json using the stripTrailingEmptyCommands option.

Adding Method Lines

You can use the $addbefore() and $addafter() methods with the $methodlines property of a method to add lines to an existing method.

- **$addbefore**(rItem,cText)
  adds a new line with content cText before the line specified by rItem (rItem can be either a 1-based integer line number, or an item reference to a line in the method)
- **$addafter**(rItem,cText)
  adds a new line with content cText after the line specified by rItem (rItem can be either a 1-based integer line number, or an item reference to a line in the method)

For example:

```
Do $cclass.$methods.$remove($cclass.$methods.Test)
Do $cclass.$methods.$add("Test")
Do $cclass.$methods.Test.$methodlines.$add("# aaa")
Do $cclass.$methods.Test.$methodlines.$add("# ccc")
Do $cclass.$methods.Test.$methodlines.$add("# eee")
Do $cclass.$methods.Test.$methodlines.$addbefore(2,"# bbb")
Do $cclass.$methods.Test.$methodlines.$addafter(3,"# ddd")
Do $cclass.$methods.Test.$methodlines.$addbefore($cclass.$methods.Test.$methodlines.1,"# New line 1")
Do $cclass.$methods.Test.$methodlines.$addafter($cclass.$methods.Test.$methodlines.2,"# New line 3")
```

Method Notes

You can add notes to a method to allow you to document each method in a class. The notes are stored in the $notes property for the method. The notes can be edited on the **Documentation** tab, the last tab in the Variable definition pane. (Existing users should note that the $notes property is the $httpnotes property renamed to $notes, which is available for all methods in a class.)

You can change the width of the fields on the documentation tab in Code Editor by dragging their borders. The positions are not saved, and will revert to equal distribution when resizing the Code Editor or changing method.

Method Notes for Subclasses

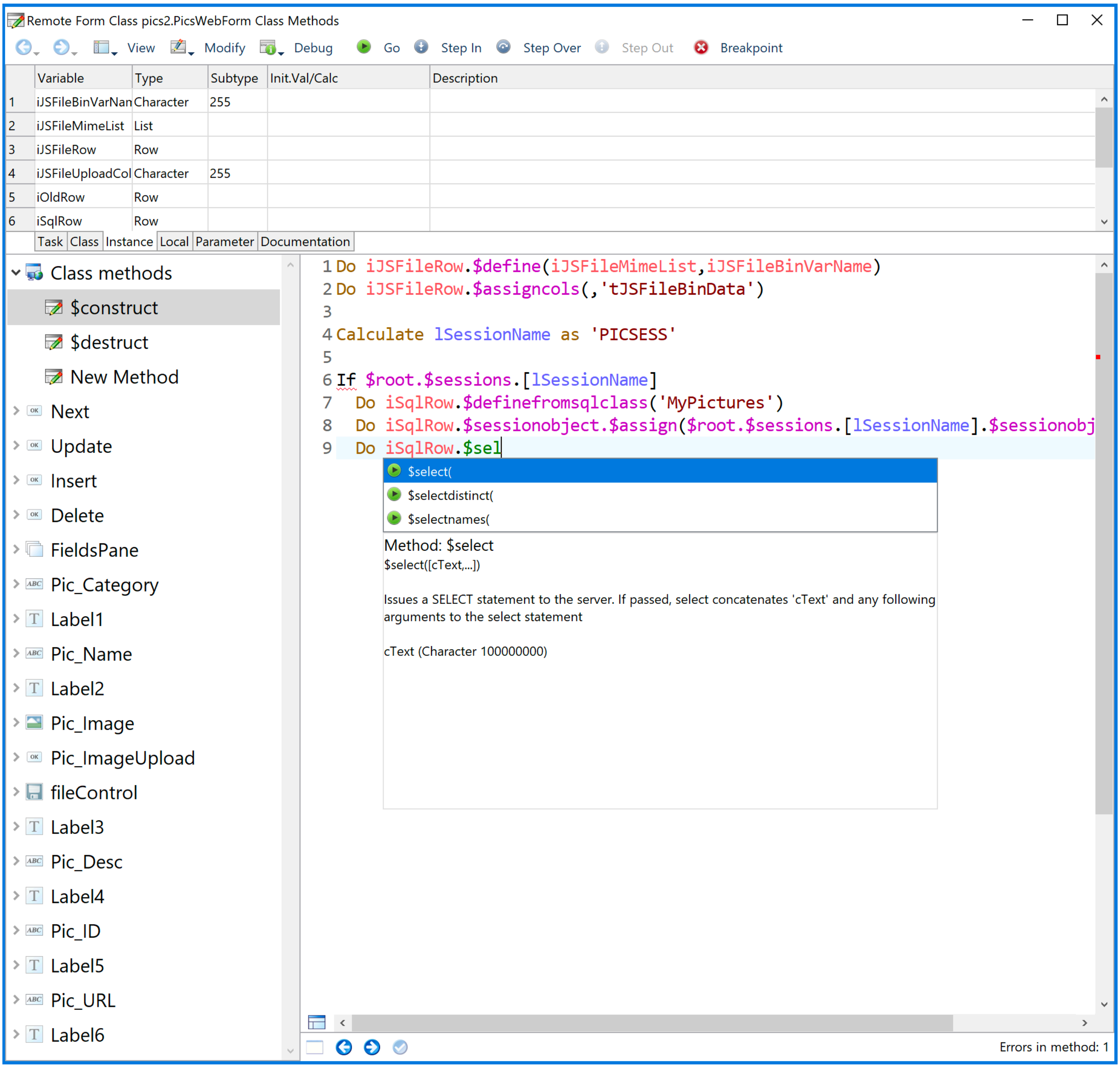
The Method Editor shows the inherited or built-in method notes for a method that is either inherited, built-in, or overridden, when the description/notes in the current class is empty. To indicate that these are inherited, they are drawn in the inherited color or built-in color, as appropriate. If you want to change the value for an overridden method, you can right-click on description/notes, and select “Make Editable”, and edit the value. You can revert to the inherited value by editing and deleting the description/notes; when the focus leaves the edit field, it shows the inherited value.

Code Editor

The **Code Editor** allows you to enter Omnis code directly into each command line in a method, and when combined with the Code Assistant, and the keyboard shortcuts, allow you to write Omnis code quicker and more easily. (The “free-type Code” Editor replaces the “point-and-click” code entry panel, available in versions prior to Studio 10, which has been removed from this version, that is, you cannot revert back to the old code entry panel.)

To enter a command, you click into or tab to *an empty command line* and type the *first letter* or the *first few letters of a command name* and select it from the Code Assistant that pops up: in some cases, you will only need to type the first letter of a command to select it, such as “d” to find the *Do* command.

As you complete or select a command name or a parameter, the insertion point will move to the appropriate point in the command line, and the Code Assistant will provide more help as you type, including help with command syntax, variable names, parameters, command options, as well as property and method names. For example, in the following screen, after typing “$sel” all possible methods are shown in the Code Assistant popup list, in this case $select() and so on (see Code Assistant for specific information about the Code Assistant).



	Variable	Type	Subtype	Init.Val/Calc	Description
1	iJSFileBinVarNan	Character	255		
2	iJSFileMimeList	List			
3	iJSFileRow	Row			
4	iJSFileUploadCol	Character	255		
5	iOldRow	Row			
6	iSqlRow	Row			

```
Do iJSFileRow.$define(iJSFileMimeList,iJSFileBinVarName)
Do iJSFileRow.$assigncols(,'tJSFileBinData')

Calculate lSessionName as 'PICSESS'

If $root.$sessions.[lSessionName]
  Do iSqlRow.$definefromsqlclass('MyPictures')
  Do iSqlRow.$sessionobject.$assign($root.$sessions.[lSessionName].$sessionobj
  Do iSqlRow.$sel
```

$select(
$selectdistinct(
$selectnames(

Method: $select
$select([cText,...])

Issues a SELECT statement to the server. If passed, select concatenates 'cText' and any following arguments to the select statement

cText (Character 100000000)

Figure 87:

To enter a line of code in the Method Editor:

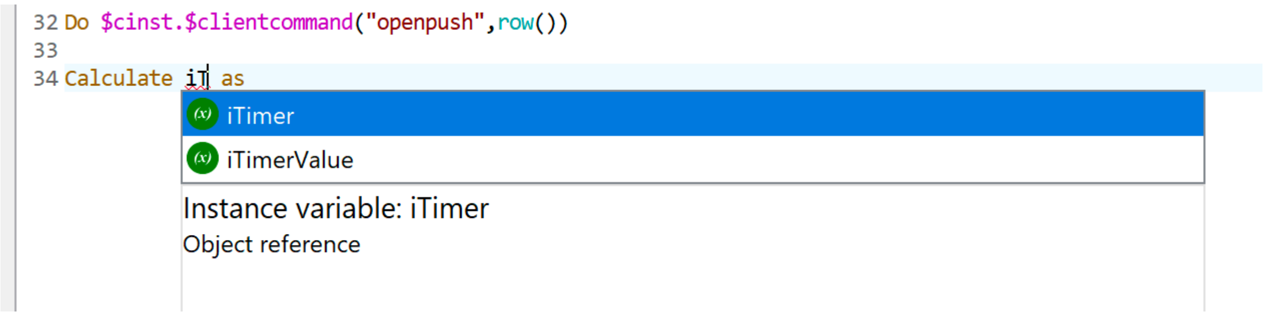
- Click or tab into an *empty method line*: the insertion point should be at the start of the empty method line. You can press Ctrl-N to create a new line under the current line

- Type the first few letters of the command you want to enter: for most commands you will only need to type 2 or 3 characters (you can ignore case and leave out any spaces in the command name)

- As soon as you start to type, the Code Assistant will drop down automatically showing a list of commands that match the characters you have typed: you can press **Tab** to select the first/selected command in the list, or use the **Arrow** keys to navigate up or down the list, and press **Return/Enter** to select a command

- Having selected the command, you can start to fill out its parameters: again, you can type the first few characters of a variable name or parameter and select it from the Code Assistant help list

For example, to enter a calculation using the *Calculate* command, you can type “ca” (note lower case) and press the **Tab** key to select the *Calculate* command from the help list, which should be the first command in the list. The insertion point should now be between ‘Calculate’ and ‘as’. Type the first few characters of the variable name or notation you want to enter, select the variable or notation from the help list (you can press Tab to select the first item in the list):



```
32 Do $cinst.$clientcommand("openpush",row())
33
34 Calculate iT as
```

Instance variable: iTimer
Object reference

Figure 88:

Once you have selected the variable name for your calculation, you can press Tab to go to the end of the command line, in this case, after the ‘as’, and then enter the calculation, including any functions or notation.

In all other respects the Method Editor behaves the same as in previous versions, including the Chroma coding which has been greatly enhanced with an updated theme. The following sections provide more detail about entering commands in the enhanced Method Editor.

Tokenization

Omnis is a tokenized language, which means that all method text has a single canonical representation generated from the tokenized representation of the code. As you enter text into the Code Editor, Omnis tokenizes the code and then updates the editor with the canonical representation. For example, this means that extra whitespace will be deleted, and attempts to indent the code using a non-default indent will have no effect. In addition, each command must occupy a single line, and command lines do not wrap. Each level of indent corresponds to two spaces.

To make the editor more efficient, it requires a fixed width font. The Method Editor Fonts… dialog restricts the list of fonts it offers via the Code combo box to fixed width fonts.

Bad name detection

Bad notation names are detected while entering code rather than handling this through automatic retokenization using double slashes. The ‘badNotationNameIsSyntaxError’ item in the ‘ide’ section of config.json controls this behavior; the default is True. Set this to false to restore the previous behavior.

Fonts

The Code Editor supports variable-width fonts. The various elements of the Code Editor, including the code area and method list, use the default fonts provided by your current operating system: e.g. on Windows Segoe UI and Consolas are used as the default fonts. You can change the fonts used under the View>>Fonts option, while the Reset option lets you return to the default fonts for your OS.

Ctrl-space

The Code Assistant drops down automatically when you type a command name, function or some notation, but you can force the **Code Assistant** to open at other times. To open the Code Assistant manually, position the caret in the code text, press **Ctrl-Space,** and the text immediately before the caret is used to determine the contents of the Code Assistant help list.

One situation in which this is useful is if you cannot remember the syntax of a command or function: position the caret immediately after the command or function name, press Ctrl-Space and then down arrow, and you will see the syntax for the command or function in the Code Assistant help list. See Code Assistant.

Multi Undo and Redo

The Method Editor supports multiple levels of Undo, and Redo. (The multi-level Undo/Redo also applies to all Edit fields in the fat client and IDE.)

Read-only Mode

When the Modify menu is enabled in the Code Editor, you can toggle the editor between read-only and write mode using the keyboard shortcut Alt+M / Cmnd+Opt+M (stored in $keys). The method editor stores the state of "Read-only mode" with the Window Setup.

Copying Methods or Code Lines

You can copy a selected method in the Method name list (on the left) using the standard Edit menu **Copy** option (or Ctrl/Cmnd-C), or the Method Editor context menu Copy option.

While working in the Code Editor (on the right), you can copy a line or selected lines of code (text) using the standard Edit menu **Copy** option, but only the selected characters will be copied. Alternatively, you can use the Copy Lines option in the Code Editor context menu to copy the *complete code* in the current line (the line containing the caret), or *all complete lines* in the current selection.

You can select all the lines in a method using the **Select All** option in the Edit menu or the Ctrl/Cmnd-A keyboard shortcut. For long methods that extend down beyond the visible area in the Code Editor, all lines are selected but the editor window does not scroll. However, in this case, if you want the editor window to scroll, you can set the "selectAllCanScrollCodeEntryField" setting in the "methodEditor" section of the config.json file to true (false by default); this allows you to select the whole method and see the end of the method.

When you copy a method or lines of code, Omnis copies the syntax coloring and other formatting, therefore, this would allow you to paste the code into a word processor or an email and retain the colors and formatting (the code is copied in HTML format on macOS and Windows).

Printing Methods

You can print a method using the Print option in the File menu; the print output will be sent to the current destination. Omnis uses the syntax colors from the *default* design theme (which is designed for a white background), i.e. if you are using a dark theme in the Code Editor, this will be ignored. You can turn off this behavior (and print everything using black text) by setting the entry "printMethodsWithSyntaxColors" in the "methodEditor" section of config.json to false.

Entering Commands

To enter an Omnis command the cursor must be on an empty line, and you can start to type the name of the command you need. As soon as you type the first letter, the **Code Assistant** will open automatically, displaying a list of commands starting with that letter: note that the command filters may limit which commands are shown, see below about the filters. As you type further letters of the command name, the Code Assistant refines the list of available commands. In most cases you will only need to type the first 2 or 3 letters to locate a command. The text immediately before (to the left of) the caret is used to determine the content of the Code Assistant help list.

To select a command from the Code Assistant help list, you can press the **Tab** key to select *the first command displayed* in the list, or you can use the **arrow keys** to navigate up and down the help list and use **Return** to choose the selected command.

Assuming the cursor is at the end of the selected command name, you can start to enter its parameters, and the Code Assistant should pop up automatically at the insertion point whenever a variable name or parameter is needed.

Command Filters

The commands in Omnis perform many different functions, including many legacy features that are no longer required for creating web and mobile apps using the JavaScript Client. There is a filter mechanism in the Code Editor to filter the list of commands that are displayed in the **Code Assistant** help list, primarily to remove any old commands, including those that allow you to manage Omnis datafiles.

Note you can still use the excluded commands in your code, and methods in converted libraries using these commands will continue to work – the filters just hide the commands from the Code Assistant help list.

The command filter is set under the **Filter Commands** submenu in the Modify menu: note this is only visible when the cursor is in the code entry area in the Code Editor. The **Exclude Old Commands** filter is enabled by default, which excludes over 100 old commands, plus there are other filters available that exclude smaller subsets of commands. You can disable the current filter using the **No Filter** option, in which case all the commands available in Omnis will be shown in the Code Assistant help list.

The current filter option is saved with the Window Setup for the Method Editor: if the saved value is no longer present, the editor reverts to no filter and all commands will be shown in the Code Assistant.

The **Reload Command Filters** option reloads the filters from the commandfilters folder, without having to quit Omnis, which is useful if you have changed or added any filters.

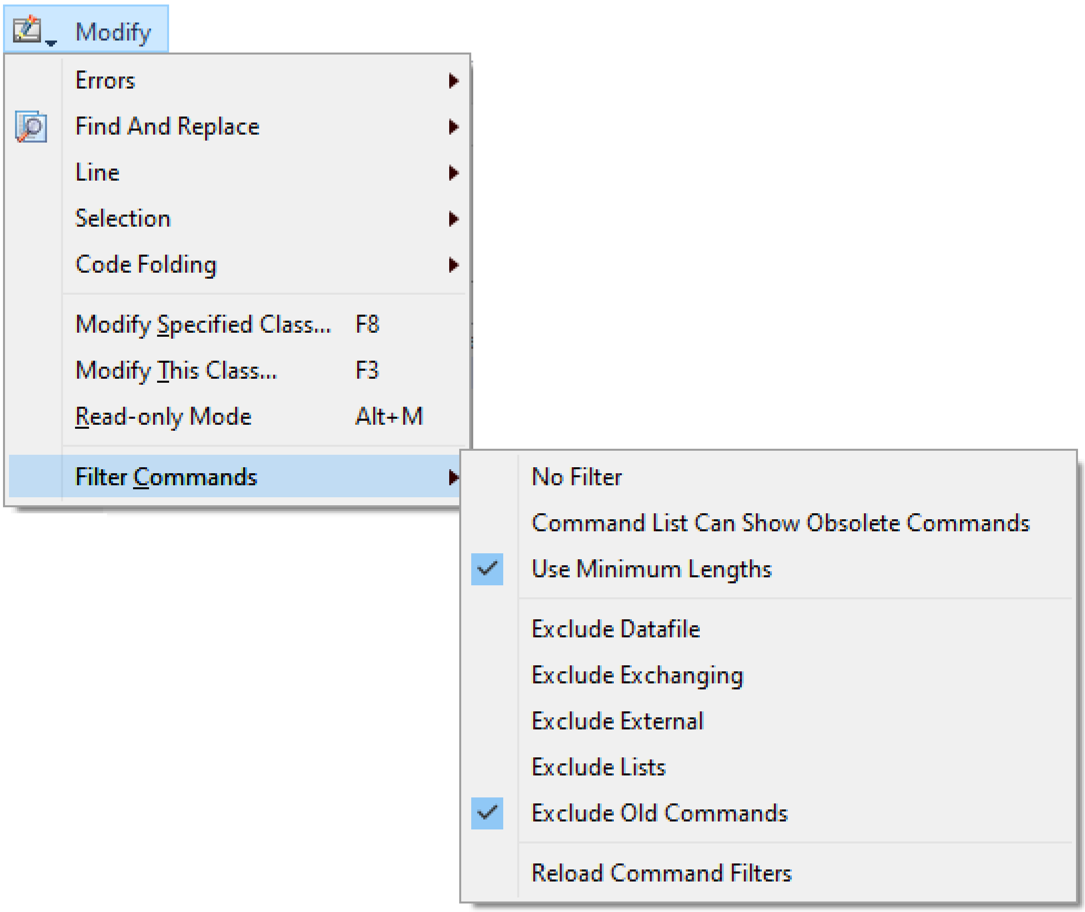


Figure 89:

Further Command Filtering

Normally, all commands matching the *first typed character* appear in the Code Assistant list, but you can limit or change which commands are shown depending on the number of characters typed – this may be useful if you want specific commands to always appear, instead of the default ones that appear first in the alphabetical list of commands.

You can control this type of filtering using the **Use Minimum Lengths** option on the **Modify>>Filter Commands** submenu, and this option is enabled by default. For example, with this enabled, the *Do* command will be selected by default when you type 'd' (rather than the *Default* command), and *Quit method* will be selected by default when you type 'q' (rather than the *Queue* commands); in the latter case, you can type 'qu' to show all the *Quit* commands in the Code Assistant list (note the *Queue* commands only apply to wndow classes in desktop apps).

The filtering enabled by the **Use Minimum Lengths** option is controlled in the file min_command_characters.json (located in the 'studio' folder) which specifies the minimum number of characters to be typed for a specified command.

The JSON file contains an object, where each member name is either a command name, or a regular expression matching a set of command names. The value of each member is the minimum number of characters to type (default 1 if there is no match for a command). In the following example, Quit method appears as soon as you type 'q', whereas the other Quit commands require you to type 'qu', and the Queue commands require you to type 'que':

```
"^Queue.*": 3,
"Quick check": 4,
"Quit method": 1,
"^Quit.*": 2
```

Regular expressions must start with ^, otherwise the entry is treated as a full command name.

If the file is not present in the studio folder, or if it cannot be loaded for some reason (e.g. invalid JSON syntax), the Use Minimum Lengths menu item is hidden.

Omnis loads the file min_command_characters.json at startup, and when you execute the **Reload Command Filters** command on the Filter Commands menu.

Editing the Command Filters

You can create your own filters, or change the ones provided, to change the commands that are shown in the Code Assistant help list. If you wish to adapt the default filter, you are advised to make a copy of it, rename the copy, then edit and save the new file.

The command filters are located in a folder called 'commandfilters' in the Studio folder: the default filter is called 'Exclude_Old_Commands.json'. Each file in this folder is loaded in the Filter Commands submenu, and the name of the JSON file is used as the menu option name. (You can examine the contents of each filter file to see which commands they exclude from the Code Assistant help list.)

The content of each JSON file is an object with a single member named "exclude", listing any commands that are to be excluded from the Code Assistant help list. The exclude member is an array, and each array entry is the exact command name (case insensitive).

You can exclude groups of commands using a regular expression to match command names: in this case, you need to anchor the regular expression to the start, using ^. For example, to exclude all old MSM... commands, you can create a filter file with the following contents (name the file 'Exclude_MSM_Commands.json'):

```
{
  "exclude": [
    "^MSM.*"
  ]
}
```

As well as creating an exclude filter, you can create a filter to only *include* certain commands, although in practice this might only be useful if you want to use a very small subset of commands in the Method Editor (since all commands that *are not included* are excluded). To create an include file, create a new filter file containing an "include" object, and add any command names to be included, e.g. to only include Do and Calculate (and exclude all other commands!), the filter should contain:

```
{
  "include": [
    "do",
    "calculate"
  ]
}
```

The default or initial filter is set in the 'currentCommandFilter' option in the 'codeAssistant' section of the config.json file: if this is empty, or the command filter files or folder are removed, then "no filter" is selected.

You need to select the **Reload Command Filters** option in the Modify menu to load any new or edited filters into the Filter Commands submenu.

Case and Omitting Spaces

You can ignore the case of all command names, so you can always start to *type a command name in lower case.* Furthermore, if the command name includes spaces, *you can omit the space(s),* which will speed up command selection in the Method Editor.

Whether or not you include the space can, however, *determine which command is selected* by the Code Assistant: this is important for the *Do...* commands, for example. Typing do<space> will immediately enter a *Do* command (and the insertion marker will be ready to accept the calculation) and *close* the Code Assistant. Whereas, to select the *Do method* command, you can type dom<tab> (note you can omit the space), or to select the *Do async method* command, you can type doa<tab>. This is quicker than typing just 'do' and then selecting the command you want from the droplist in the Code Assistant.

Another example would be in the case of the If... commands. Typing if<tab> will immediately close the Code Assistant and enter an *If* calculation command, whereas, to select the *If canceled* command, you can type ifc<tab> (note no space). Similarly, typing on<space> will select the *On* event command, while typing ond<tab> will enter the *On default* command.

Tab key

You can use the Tab key to tab between the parameters of all of the commands in the method. This is an easy way to navigate through the commands, skipping command names and keywords and moving the insertion point to the next available position. You can also use the Tab key to select the first or selected line in the Code Assistant: in this case, if you select a method, such as Do List.$define, the opening and closing parenthesis () will be added automatically and the cursor is placed between the parenthesis.

Construct Commands

If you enter a construct command using the Code Assistant, such as *If,* it will add the end construct command automatically, in this case, *End If.* You can use Undo to remove the end construct command added automatically if it is not required.

The Method Editor checks for missing associated commands as you edit, e.g. *If* with no *End If*, or *For* with no *End For*.

Command Options or Keywords

For commands that have options or keywords, usually enclosed in parenthesis, you can enter the options from the keyboard automatically. To show the options or keywords for a command, press **Tab** directly after typing the command name. For example, after typing *Do, For* and *Enter data,* press Tab and the Code Assistant appends the optional keyword(s) to the command, ready for you to enter its parameter(s). If you do not want the keyword added by tab, the Undo command will remove the option(s).

This occurs when the cursor is somewhere in the command, the command does not have the missing keyword, and no characters are selected. For example, pressing tab after entering Do $cinst.$test() will add the Returns keyword. In the case of the *For* and *For each line in list* commands, tab will cause the keywords from, to and step to be added in turn.

There is an option on the **Line** menu, *Tab Adds Missing Optional Keyword,* which controls this behaviour (it is enabled by default); the state of this option is saved with the Code Editor window setup.

Construct Parameters

Where possible, the Code Assistant help window expands "params…" for $add, $open, etc to show the constructor parameters of the class referenced. Omnis identifies the class name that precedes the method name in your code (e.g. classname.$open), and will show the construct parameters for the class.

Class Names

To enter a quoted class name, you can press Ctrl-Space when the caret is positioned after a double quote (or some text following a double quote) and select the name from a quoted list of non-system class names in the current library.

File Class Field & Library Names

When unique field names is true, the Code Editor does not enter a file class name prefix when you enter a file class field/variable name into the Calculate command, for example, for file classes in the same library as the class being edited. There is a configuration item called 'checkFileClassPrefixBasedOnUniqueFieldNames' to control this behaviour (true by default).

When unique field names is false, checkFileClassPrefixBasedOnUniqueFieldNames requires that you enter a file class name prefix, for file classes in the same library as the class being edited.

In addition, the Code Assistant only shows file class field names at the top level when unique field names is on; so if unique field names is off, the list just includes the file class names.

The Code Assistant includes library names at the top level, to allow references like lib.file.field to be entered, or lib.<library notation> to be entered.

Method Name Matching

You can search for a method name where the name of a method is required in a line of code. To do this press **Shift-space** after entering a string in the Code Assistant and any possible matching *method names* are added to the help list. For example, you could enter: *Do code method **test*** and then press **Shift-space,** and the Code Assistant displays all strings containing "test" that can be used as a method name parameter of Do code method.

For notation, if you enter $test and then Shift-space, the Code Assistant only shows matching strings that are notation (start with $) and contain "test".

Side by Side Editors

You can open two instances of the method editor to show *two methods from the same class,* for example. You can open a second copy of the method editor as follows:

- Press the **Shift** key while performing an action that opens the method editor, such as double-clicking on a remote task.
- Use the **Two Editors Side By Side** option on the method editor **View** menu.

Omnis opens a second editor, next to the current editor window, so that each editor uses half of the available screen space. On Windows, this means the available space in the main Omnis application window, and on macOS, it means the available space on the current monitor less the menu bar or toolbars.

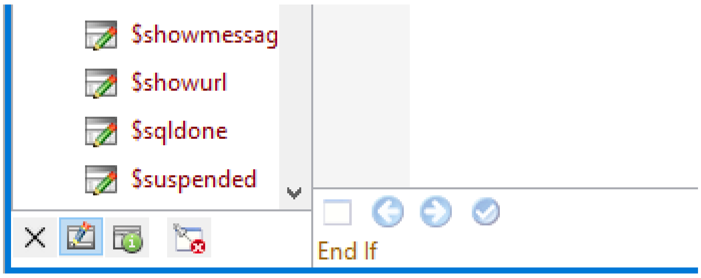
When two editors are open, the same method in each class can be selected in both editors, but the editor in the background does not display the method: it displays the text “This method is being edited in another method editor”.

The editor in the background keeps up to date with changes in the foreground editor, e.g. when you add or delete a method, the method list in the other editor updates.

There is a keyboard shortcut for the Two Editors Side By Side command, which defaults to Alt+S on Windows and Cmd+Opt+S on macOS.

Debug Panel

The Method Editor has two panels below the code entry window: the *Debug panel* and the *Editor panel*. You can show the Debug panel (or hide it) using the buttons under the method name list in the lower left corner of the Method Editor window (the X option will hide both panels).



```
$showmessag
$showurl
$sqldone
$suspended
```

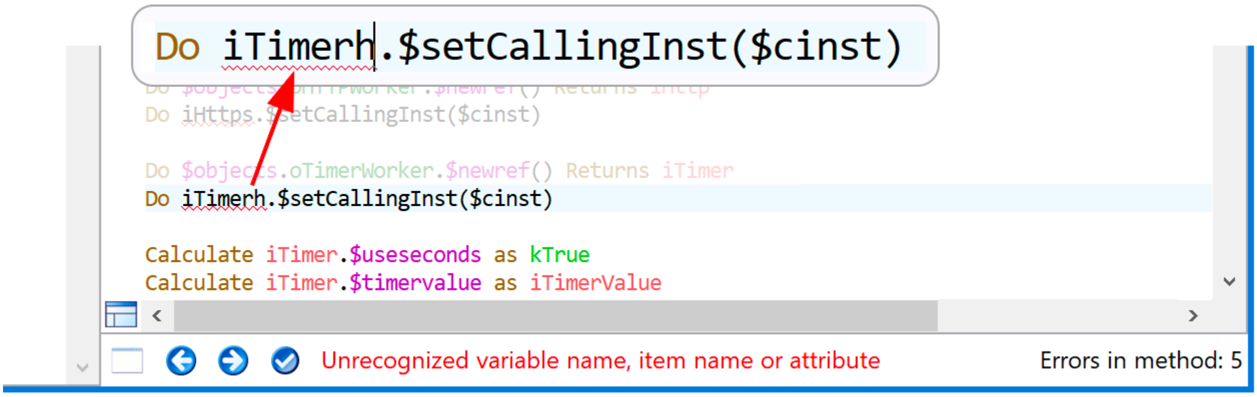
```
End If
```

Figure 90:

The equivalent options are on the Bottom Panel hierarchical menu on the **View** menu in the Code Editor.

Editor Panel and Errors

As you enter code, Omnis tokenizes the entered code and provides real-time feedback that indicates if the method code is valid. Valid method code is syntax-colored, whereas invalid method code is partially syntax-colored, and the invalid component(s) in the method line underlined using a colored wavy line (the color is taken from the current theme or set in the “badsyntaxcolor” $appearance preference).



```
Do iTimerh.$setCallingInst($cinst)
```

```
Do iHttps.$setCallingInst($cinst)

Do $objects.oTimerWorker.$newref() Returns iTimer
Do iTimerh.$setCallingInst($cinst)

Calculate iTimer.$useseconds as kTrue
Calculate iTimer.$timervalue as iTimerValue
```

Unrecognized variable name, item name or attribute

Errors in method: 5

Figure 91:

The editor panel at the foot of the Method Editor window displays the number of method errors, and when the caret is positioned within text causing an error, it displays the error text.

The editor panel has three buttons that allow you to handle errors. The Next and Previous error buttons (forward and back arrows) navigate through the errors in the method. The Fix error button (check mark) allows you to fix certain errors and will only be enabled when the caret is positioned in some text for an error. The Fix button is enabled to allow the following errors to be fixed:

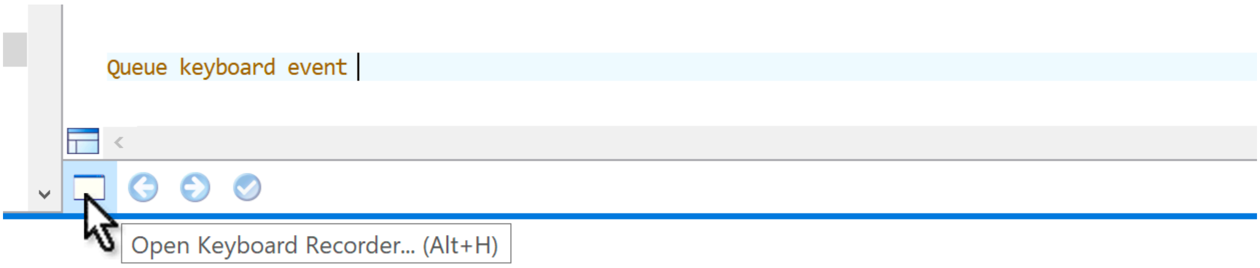
- “Unrecognized variable name, item name or attribute” and “Unrecognized variable name”: Pressing the Fix button opens the Create Variable dialog.

- ") missing": Pressing the button adds the)
- "Partly entered keyword": Pressing the button completes the keyword

In addition, the editor draws a red marker in the vertical scrollbar for each method line containing an error. The marker in the scrollbar is positioned so that when the method line containing the error is scrolled to be the first displayed line, the top of the scrollbar thumb lines up with the top of the marker. (Note that this is why the vertical scrollbar always allows scrolling even if all method lines fit within the editor window.)

Editor Helper dialog

In addition to the error reporting, there is a button to open the Helper Dialog, which is context specific. This button is disabled when the context means there is no helper dialog. If a helper dialog is available, the button is enabled, and its tooltip changes appropriately: pressing Alt+H will open the Helper Dialog.



```
Queue keyboard event
```

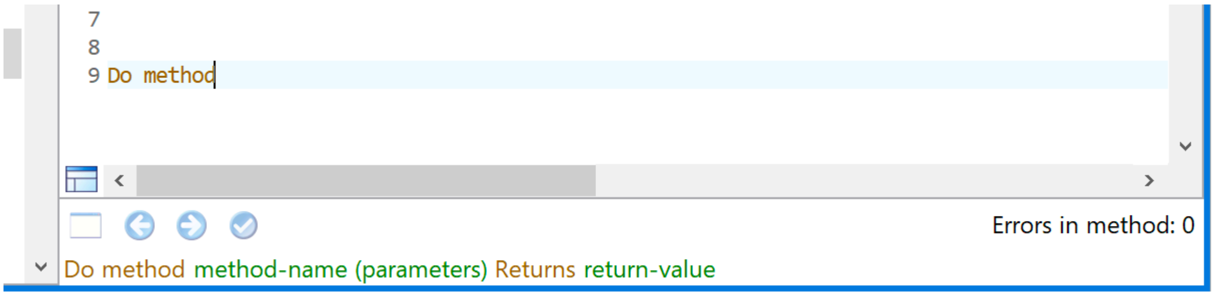
Figure 92:

The editor Helper Dialog is enabled in the following cases:

- When the caret is positioned in the parameter field of the *Queue keyboard event* command. In this case, the helper dialog allows you to record keys.
- When the caret is positioned in the title parameter of the *Working message* command. The helper dialog is the working message configuration dialog.
- When the selection includes only *JavaScript:* commands (in a client executed method). The helper dialog button will open the JavaScript editor. All JavaScript: command lines in the same contiguous block are selected, and their JavaScript is then editable using the popup editor. When the popup editor closes, Omnis replaces the selected JavaScript: commands with JavaScript: commands containing the contents of the popup JavaScript editor.
- When the selection includes only *Sta:* commands (for entering a SQL statement on multiple lines). The helper dialog opens the same external editor for JavaScript but in SQL mode allowing you to enter a SQL statement over multiple lines.

Command Syntax Help

You can view the full syntax for a command, including all its parameters and options, in the Help panel at the bottom of the editor window. This type of help is displayed once you have selected a command from the Code Assistant list, or you have typed the command name in full – as you reach the last character of the command the syntax help is shown. For example, if you type the *Do method* command, its syntax is show in the Help panel at the bottom of the editor window.



```
7
8
9 Do method
```

Errors in method: 0

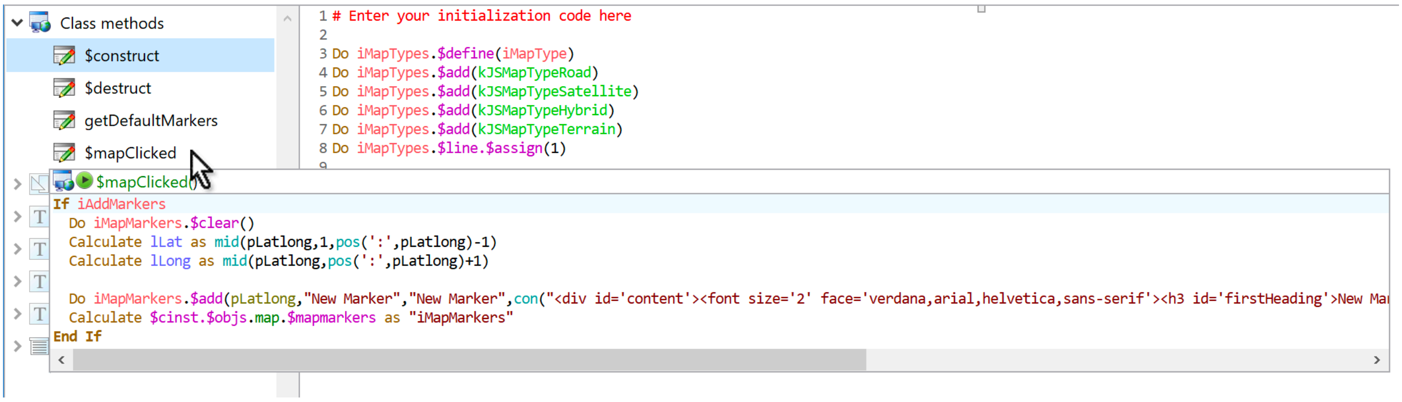
Do method method-name (parameters) Returns return-value

Figure 93:

You can hide the command syntax by unchecking the "Show Syntax Strings" option on the View menu.

Method Tips

Method Tips or tooltips are displayed when you hover the pointer or I-beam over a method name in the Method Editor, including methods listed in the Method tree list on the left of the editor window:

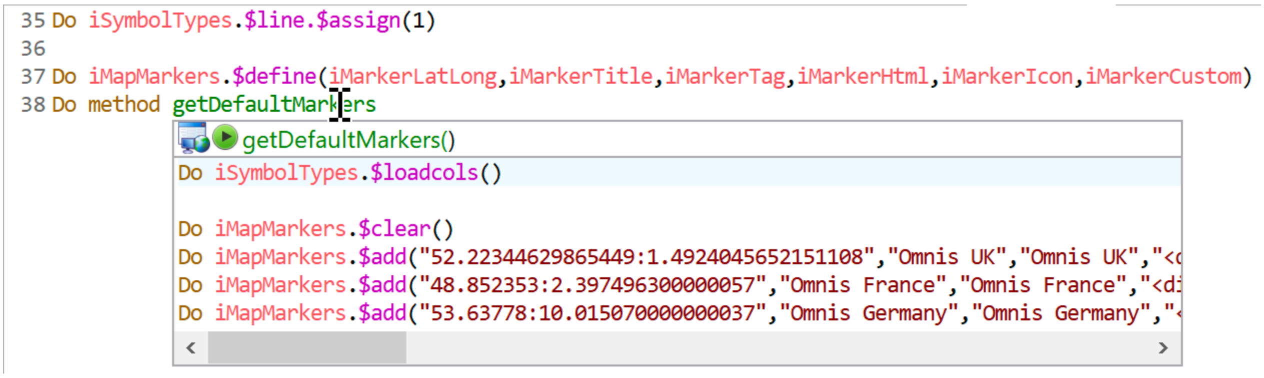


```
# Enter your initialization code here

Do iMapTypes.$define(iMapType)
Do iMapTypes.$add(kJSMapTypeRoad)
Do iMapTypes.$add(kJSMapTypeSatellite)
Do iMapTypes.$add(kJSMapTypeHybrid)
Do iMapTypes.$add(kJSMapTypeTerrain)
Do iMapTypes.$line.$assign(1)
```

```
If iAddMarkers
  Do iMapMarkers.$clear()
  Calculate lLat as mid(pLatlong,1,pos(':',pLatlong)-1)
  Calculate lLong as mid(pLatlong,pos(':',pLatlong)+1)

  Do iMapMarkers.$add(pLatlong,"New Marker","New Marker",con("<div id='content'><font size='2' face='verdana,arial,helvetica,sans-serif'><h3 id='firstHeading'>New Ma
  Calculate $cinst.$objs.map.$mapmarkers as "iMapMarkers"
End If
```

Figure 94:

or method names that are being called in your code (assuming Omnis can identify the method being called).



```
Do iSymbolTypes.$line.$assign(1)

Do iMapMarkers.$define(iMarkerLatLong,iMarkerTitle,iMarkerTag,iMarkerHtml,iMarkerIcon,iMarkerCustom)
Do method getDefaultMarkers
```

```
Do iSymbolTypes.$loadcols()

Do iMapMarkers.$clear()
Do iMapMarkers.$add("52.22344629865449:1.4924045652151108","Omnis UK","Omnis UK","<d
Do iMapMarkers.$add("48.852353:2.397496300000057","Omnis France","Omnis France","<di
Do iMapMarkers.$add("53.63778:10.015070000000037","Omnis Germany","Omnis Germany","<
```

Figure 95:

The method name and its code are displayed in the popup tip window and you can scroll longer methods using the mouse or trackpad. You can hold down the **Shift** key to keep the tooltip window open when you move the pointer, which allows you to scroll the window more easily.

The Method tips provide a useful preview of a method, without having to switch away from the current/selected method you're working on. You can dismiss the method tip by moving the mouse away from the method name and tooltip, or by pressing Escape.

There are three entries in config.json that control the size of the Method tips:

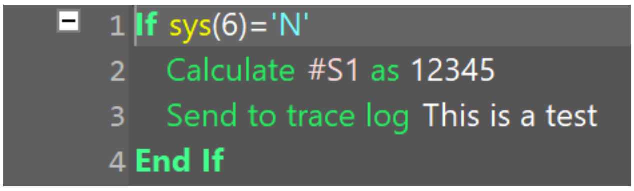
- "maxWidthOfMethodTooltip": 500 (value in pixels)
- "maxHeightOfMethodTooltipGeneralInformation": 100 (value in pixels)
- "maxVisibleMethodLinesInMethodTooltip": 20 (number of lines)

When used with the method tree list, the maximum width used is the width of the code edit text field if it is wider than the maxWidthOfMethodTooltip config item.

Code Folding

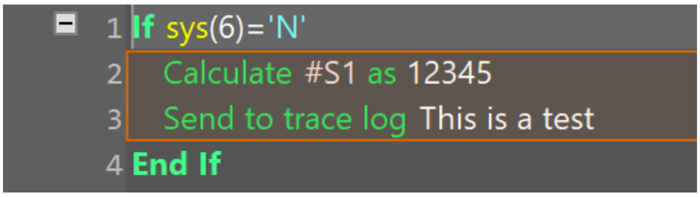
The Code Editor allows you to *fold* and *unfold* (collapse and expand) blocks of code in order to assist with readability and code manipulation in general. If a code block can be folded, a '-' icon appears in the margin at the start of the block: when a block has been folded a '+' icon is shown next to the first line of the block, and directly under this is shown a "badge" (an ellipsis icon) representing the hidden code content.

The Code Editor shows a fold icon (⊟) in the left margin which shows that a code block can be folded: you can click on the icon to fold the block, and the icon will toggle to show an unfold icon (⊞) to show that the block can be unfolded. For example, this is a code line before code folding:



```
If sys(6)='N'
   Calculate #S1 as 12345
   Send to trace log This is a test
End If
```

Figure 96:

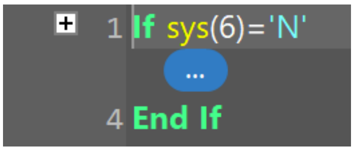


```
If sys(6)='N'
   Calculate #S1 as 12345
   Send to trace log This is a test
End If
```

Figure 97:

When the mouse is over the fold icon, Omnis highlights the block that will be folded, for example:

After you have clicked the fold icon, and the code has been folded, the content is shown as a badge (ellipsis) representing the content of the folded block:



```
If sys(6)='N'
   ...
End If
```

Figure 98:

When the mouse is over the badge icon, Omnis displays a tooltip to show its content (this is like the method content tooltips already in Studio 10.1), but note that this tooltip is always displayed, irrespective of the Show Method Content Tips option. For example:

Just like method content tips, pressing the Shift key while the tooltip is displayed locks it in place until you remove the Shift key and move the mouse away. You can select the text in the tooltip and copy it to the clipboard.

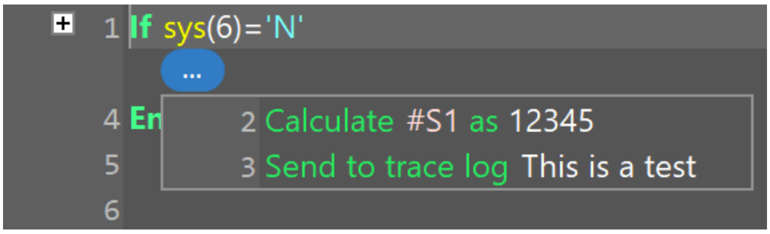
You can also press the Control (Windows) or Command (macOS) key while the mouse is over a fold or unfold icon. In this case, if the command has multiple blocks that can be folded or unfolded, Omnis highlights all the affected blocks, and pressing the fold or unfold icon while all blocks are highlighted opens or closes all the highlighted blocks. For example:

Code folding is only available in a block when there are *at least two method lines:* for a block that has a single line only, folding is not enabled for the block, so the folding icons are not shown, and the options in the folding menu are disabled.

Which Commands can be folded?

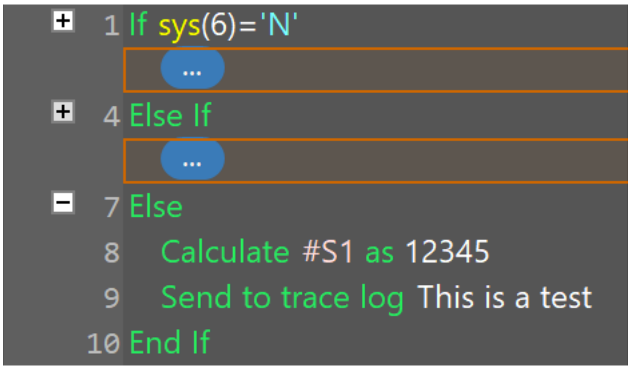
The following Omnis commands can be folded:

- All If commands, folded until the next Else, Else If or End If command in the same block.
- Else, folded until the next End If command in the same block.
- All Else If commands, folded until the next Else, Else If or End If command in the same block.
- All While commands, folded until the terminating End While command of the block.
- Both For commands, folded until the terminating End For command of the block.
- Repeat, folded until the terminating Until… command of the block.
- Switch, folded until the terminating End Switch command of the block.
- Case, folded until the next Case, Default or End Switch command in the same block.
- Default, folded until the next Case, Default or End Switch command in the same block.
- Begin reversible block, folded until the terminating End reversible block command of the block.
- Begin critical block, folded until the terminating End critical block command of the block.
- On and On default, folded until the next On or On default command, or the end of the method if there is no such command.



```
If sys(6)='N'
    ...
En
    Calculate #S1 as 12345
    Send to trace log This is a test
```

Figure 99:



```
If sys(6)='N'
    ...
Else If
    ...
Else
    Calculate #S1 as 12345
    Send to trace log This is a test
End If
```

Figure 100:

Code folding menu

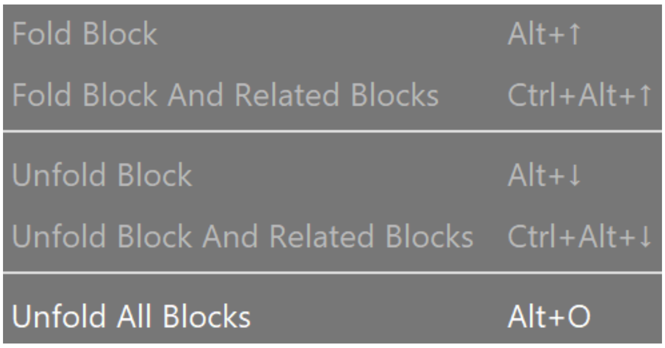
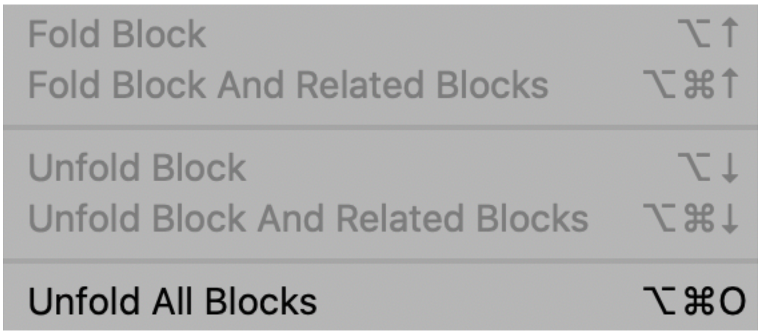
In addition to using the fold or unfold icons in the left margin, you can use the fold/unfold options on a new Code folding menu, that can be used when the code editor has the focus. In this case, most of the menu items apply to the block containing the single line of code that is currently selected.

The Code folding menu is present on the Modify menu of the Method Editor and the Remote Debugger window for a remote debugger edit session. For a remote debugger debug session, there is a new Code menu on the toolbar, containing the Code folding menu commands.

The menu commands are:

Menu command	Description
Fold Block	Equivalent to pressing the Fold icon to fold the block.
Fold Block And Related Blocks	Equivalent to pressing the Fold icon while holding the Control (Windows) or Command (macOS) key to fold the block and other related blocks that can be folded.
Unfold Block	Equivalent to pressing the Unfold icon to unfold the block.
Unfold Block And Related Blocks	Equivalent to pressing the Unfold icon while holding the Control (Windows) or Command (macOS) key to unfold the block and other related blocks that can be unfolded.
Unfold All Blocks	Unfolds all folded blocks in the method.

The menu items also have shortcuts:

Windows	macOS
	

Fold Block Alt+↑
Fold Block And Related Blocks Ctrl+Alt+↑
Unfold Block Alt+↓
Unfold Block And Related Blocks Ctrl+Alt+↓
Unfold All Blocks Alt+O

Fold Block ⌥↑
Fold Block And Related Blocks ⌥⌘↑
Unfold Block ⌥↓
Unfold Block And Related Blocks ⌥⌘↓
Unfold All Blocks ⌥⌘O

You can configure the keys for these shortcuts using the keys preference item, in the methodEditorAndRemoteDebugger group (in the keys.json file):

Preference item	Key(s)
codeFold	Opt + Up Arrow
codeFoldRelated	Cmnd + Opt + Up Arrow
codeUnfold	Opt + Down Arrow
codeUnfoldAll	Cmnd + Opt + O
codeUnfoldRelated	Cmnd + Opt + Down Arrow

Selecting Code using the pointer

You can select the badge representing a code folded block, either using the keyboard or using the mouse. When the badge is selected, the content of the block it represents is selected. In addition, double clicking on the badge selects its content.

When Omnis needs to select a line in a folded block, e.g. when hitting a breakpoint, or clicking on a stack list entry, the editor *automatically unfolds the block* (and any containing blocks) in order to display the line correctly.

Entry Behavior

As soon as an edit would affect a folded block, Omnis automatically unfolds the block (and any containing blocks) before applying the edit.

Saving the Code Folding State

Omnis stores the code folding state with the method.

When using the method editor, the state is saved back to the class with the method, provided that the editor is not operating in read-only mode. In the latter case, you can still fold or unfold methods in a read-only class, but changes to the code folding state are not saved to the class.

When using the remote debugger, changes to the code folding state are saved locally to the cache of methods loaded from the server. However, once you re-open the debug session, these changes are lost; the one exception to this is any code folding that has been applied while editing a method in a remote debug edit session.

Therefore, you should consider code folding a semi-permanent state, since as soon as Omnis needs to display the contents of a folded block for some reason, it will open the block.

Removing Code Folding

You can remove code folding from all the methods in a class or all classes in a library. All classes that can contain methods have the method $removecodefolding which removes code folding from all methods in the class, and returns the number of methods from which code folding was removed. For example, to remove code folding from all methods in all classes in a library, execute: Do $libs.library.$classes.$sendall($ref.$removecodefolding())

JSON Export

The option 'exportcodefoldingstate' in the $exportimportjsonoptions Omnis Preference ($root.$prefs) controls whether or not the code-folding state in the methods in your library is exported; the option is set to false by default so the code folding state is not exported.

If the code folding state is exported, Omnis appends the string $... to the inline comment of commands that correspond to a code folded block. This allows Omnis to regenerate the code folding state of the method when it imports the class JSON.

Word Wrapping

Long lines of code displayed in the Code Editor will wrap onto the next line automatically, and the text that wraps is drawn with an indent to make it clear that it belongs to the wrapped line (you can disable this behavior, so code lines are not wrapped, which corresponds to behavior in versions prior to Studio 10.2).

The **Word Wrap** option on the View menu of the method editor and remote debugger windows allows you to toggle Word wrapping; the option is turned on by default, and the state is saved with the window setup. When Word Wrap is enabled there is no horizontal scrollbar in the code editor window and long code lines wrap to the next line at suitable break characters, or they wrap if there is no break character.

For method content tooltips, word wrapping is always on, irrespective of the setting in the window for which the tooltip is being generated.

Inline comment wrapping & color

When word wrap is turned on and the Code editor encounters an *inline comment,* it tries to shrink the gap between the end of the code line and the inline comment *to avoid wrapping the code line if possible:* if the inline comment is still too long to fit onto the line it will wrap onto the next line, under the code line and is displayed indented.

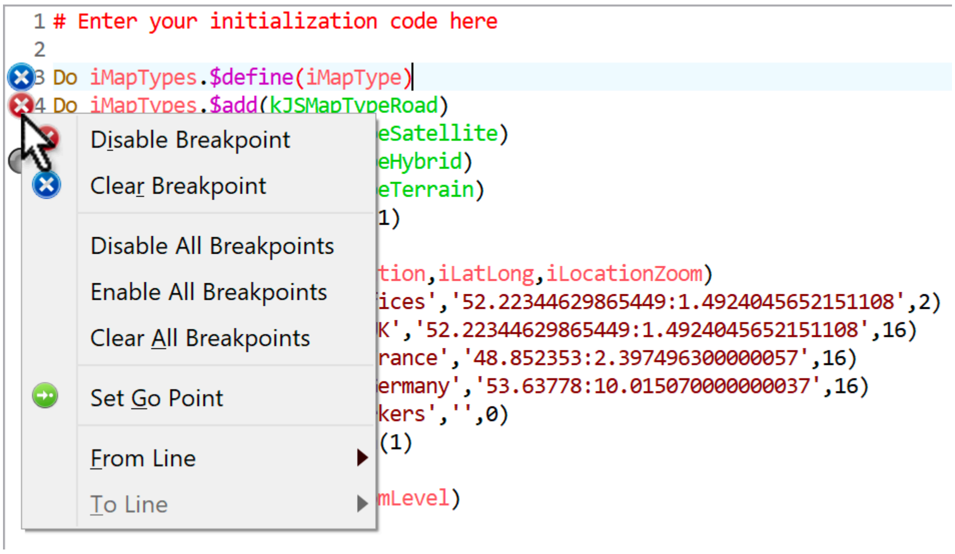
Setting Breakpoints

You can set a Breakpoint, a One-time breakpoint or the Go Point using the pointer (to click on the code margin) and the keyboard:

- You can set a **Breakpoint** using a single click in the left margin of the code editor, next to any line of code where you want the breakpoint.
- You can set a **One-time breakpoint** using Ctrl/Cmnd+click next to the line of code
- You can set the **Go point** using Shift+click next to the line of code

(Existing users should note that you can no longer set the Go point by double-clicking in the left-hand margin.)

Alternatively, you can use the **Breakpoint** context menu by right-clicking in the left margin of the code editor, next to any line of code, and selecting the option.



```
# Enter your initialization code here

Do iMapTypes.$define(iMapType)
Do iMapTypes.$add(kJSMapTypeRoad)
                                eSatellite)
                                eHybrid)
                                eTerrain)
                                1)

                                tion,iLatLong,iLocationZoom)
                                ices','52.22344629865449:1.4924045652151108',2)
                                K','52.22344629865449:1.4924045652151108',16)
                                rance','48.852353:2.397496300000057',16)
                                ermany','53.63778:10.015070000000037',16)
                                kers','',0)
                                (1)

                                mLevel)
```

Figure 101:

In addition, the Breakpoint context menu shows Delete and Disable/Enable breakpoint commands when there is a breakpoint already set for the line. It also shows the commands to Clear/Disable/Enable all breakpoints. And if there is an active stack, as well as set Go point, there is a command to Clear the stack.

Conditional Breakpoints

The **Set Condition...** option in the Breakpoint context menu (and the Breakpoints toolbar menu or Breakpoint list context menu) allows you to add conditions or a hit count to Breakpoints. You can enter a calculation that must evaluate to true (non-zero) for the breakpoint to be hit, and/or a number of hits that are to be ignored before the breakpoint is hit. The calculation and/or ignore count is displayed in parentheses in the breakpoint list.

The remote debugger displays the remote debug breakpoint, although it does not include a hit count.

The #DEBUGGER system table stores the current local debugger code breakpoint locations, which means code breakpoints (and their conditions) are restored when a library is reopened. #DEBUGGER does not appear in the Studio Browser class list, but it is included in $clib.$classes.

Menus and Keyboard Shortcuts

The Method Editor menus have been re-worked and improved for Studio 10 and include several new commands or options. The keyboard shortcut keys for some options have changed and these are listed below where they occur – there is a summary of the keyboard shortcuts at the end of this section. Where there are significant changes, an image of the menu from Studio 8 and Studio 10 is shown, so you can compare them.

View Menu

The View menu in the Method Editor has several changes or additions: some of the new options are discussed elsewhere. The **Show Debug Palette** and **Show Chroma Coding** options have been removed: the latter option has been replaced by a more comprehensive set of color options stored in the default theme in the IDE (you can change the theme in the Studio Browser Hub under **Options,** including a dark theme which may be more suited to working in the code editor).

Goto Panel

The **Goto Panel** option on the View menu lets you select a different pane in the Variables list (with keyboard shortcuts Ctrl+0 to 5). It also lets you switch the insertion point to the **Code** text entry area (Ctrl+7) ready to enter some code, or back to the Method Tree list from the code entry area (Ctrl+8).

Debug Menu

The **Debug** menu lets you run the current method via the **Execute Method** option, or test the current Remote form (or window) using the **Test Form** option.

Next are the debug options for Go, Step, Step Over, and Trace, plus you can **Set Go Point** from the Debug menu (or press Shift+F2), or use the From Line, To Line or Step Out options. The same options are available in the debug toolbar at the top of the code editing area.

As your code executes the debugger will scroll automatically to the center of the code entry area when the current line is positioned at around 75% of the visible lines.

The **Break On** <event> option allows you to select which events will stop the debugger while debugging remote form and window instances. (note IDE windows do not cause a break).

Modify Menu

The **Modify** menu contains new submenus for **Errors** and **Find And Replace.** The **Execute Method** option has been moved to the Debug menu, while the Goto panel and Fonts options have been moved to the View menu. The various Line options have been moved to the **Line** submenu.

The Comment & Uncomment options have been merged and moved to the **Selection** submenu. For classes which have an associated editor, the **Modify This Class** option opens the class editor, such as a JavaScript remote form: the shortcut key is Shift+F8.

There are additional entries that depend on the focus, as follows.

- If the focus is on the Method Tree (on the left, containing a list of methods for the class), the Modify menu contains a submenu called **Method,** which allows you to **Insert Method** (at the end of the method list, or *Before* or *After* the current method), or **Delete Selected Methods**

- If the focus is on the Code text entry area (on the right), the Modify menu contains submenus called **Line** and **Selection:** see later in this section for info.

Errors Menu

The Modify>>**Errors** submenu is new and contains **Next error, Previous error** and **Fix error** commands, that can be used instead of the buttons on the editor panel. These also have keyboard shortcuts.

Note that when the focus is on the method tree, this menu is only present when only one method is selected.

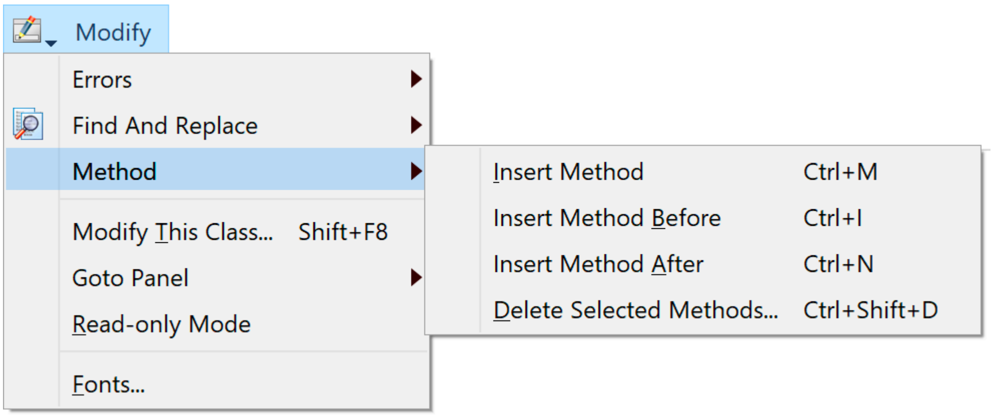

Modify
Errors
Find And Replace
Method
Modify This Class... Shift+F8
Goto Panel
Read-only Mode
Fonts...
Insert Method Ctrl+M
Insert Method Before Ctrl+I
Insert Method After Ctrl+N
Delete Selected Methods... Ctrl+Shift+D

Figure 102:

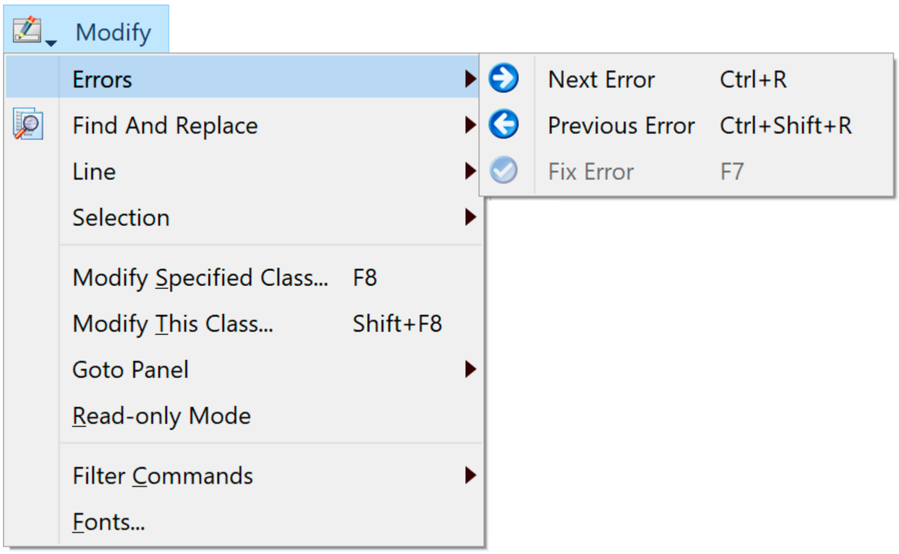


Figure 103:

Find And Replace Menu

The Modify>>**Find and Replace** submenu allows you to perform a local find and replace on the *method text for the current selected method* in the Code Editor. Note that when the focus is on the method tree, this menu is only present when only one method is selected. This menu also allows you to toggle options such as match case.

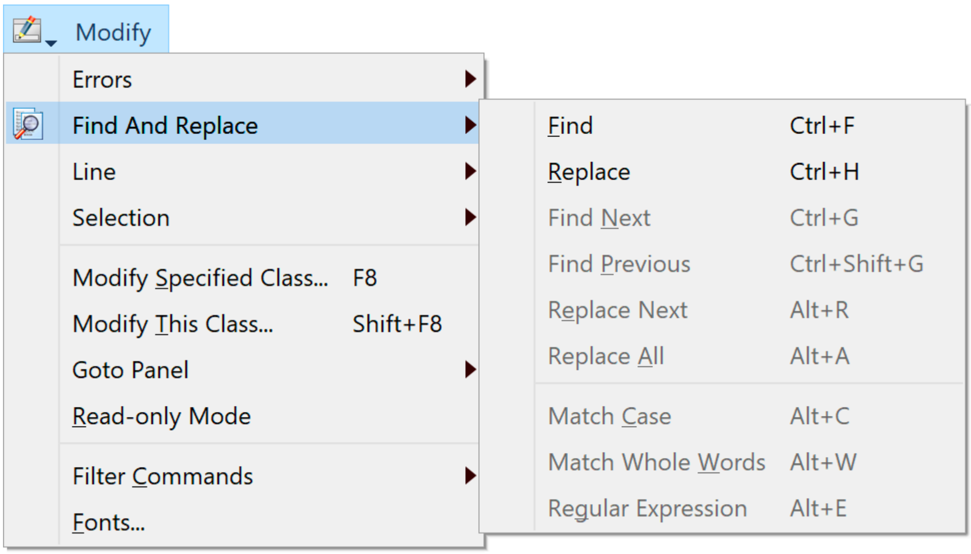
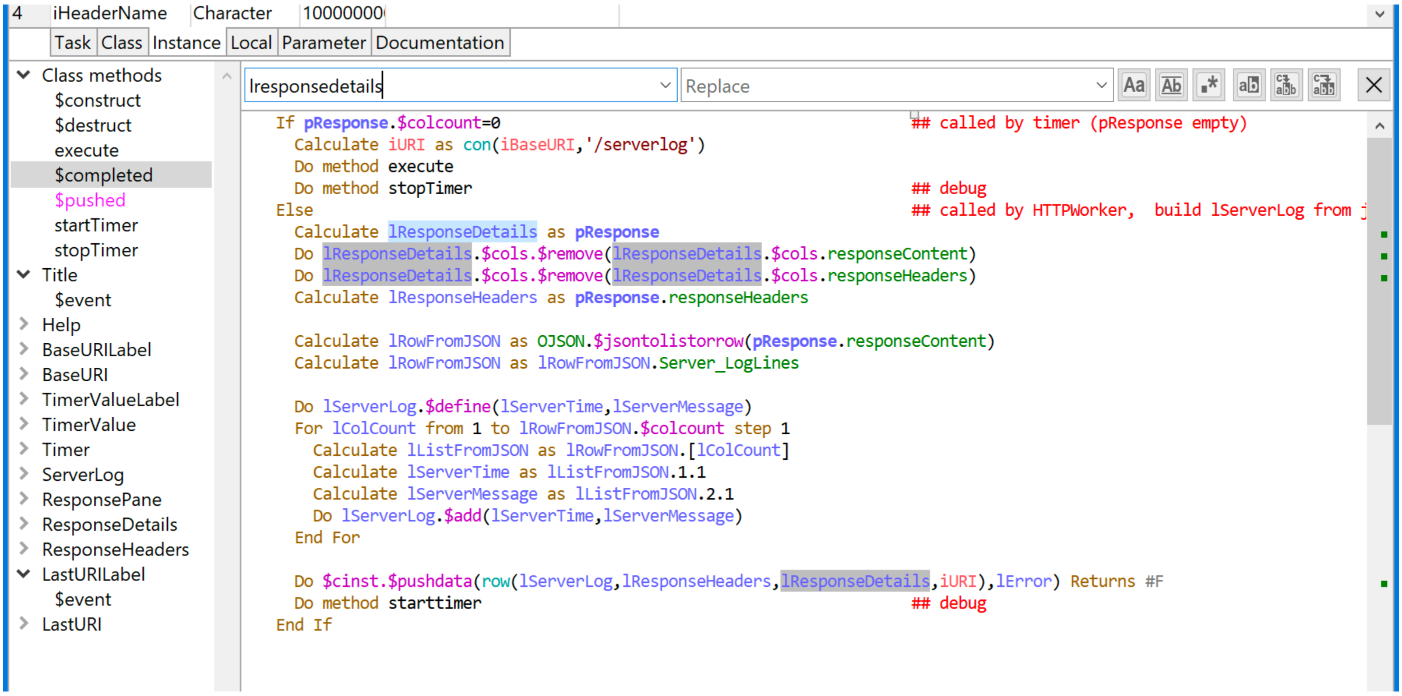


Figure 104:

The menu commands also have keyboard shortcuts, that is, Ctrl+F opens the Find panel, Ctrl+H opens Find and Replace, or Ctrl+G finds next. When you first select the Find or Replace command, the editor opens a panel immediately above the code entry field, where you can enter the find (and replace) text.



lresponsedetails

```
If pResponse.$colcount=0                                  ## called by timer (pResponse empty)
  Calculate iURI as con(iBaseURI,'/serverlog')
  Do method execute
  Do method stopTimer                                     ## debug
Else                                                      ## called by HTTPWorker,  build lServerLog from j
  Calculate lResponseDetails as pResponse
  Do lResponseDetails.$cols.$remove(lResponseDetails.$cols.responseContent)
  Do lResponseDetails.$cols.$remove(lResponseDetails.$cols.responseHeaders)
  Calculate lResponseHeaders as pResponse.responseHeaders

  Calculate lRowFromJSON as OJSON.$jsontolistorrow(pResponse.responseContent)
  Calculate lRowFromJSON as lRowFromJSON.Server_LogLines

  Do lServerLog.$define(lServerTime,lServerMessage)
  For lColCount from 1 to lRowFromJSON.$colcount step 1
    Calculate lListFromJSON as lRowFromJSON.[lColCount]
    Calculate lServerTime as lListFromJSON.1.1
    Calculate lServerMessage as lListFromJSON.2.1
    Do lServerLog.$add(lServerTime,lServerMessage)
  End For

  Do $cinst.$pushdata(row(lServerLog,lResponseHeaders,lResponseDetails,iURI),lError) Returns #F
  Do method starttimer                                    ## debug
End If
```

Figure 105:

Search Panel buttons

The panel also contains Search buttons that perform the same operations as the menu items, as follows:

Operation	keypress
Match case	Alt+C
Match whole words	Alt+W
Use regular expressions	Alt+E
Find Next or Previous	Ctrl+G or Ctrl+Shift+G(to Find Previous, you can shift click the button)

Operation	keypress
Replace next	Alt+R
Replace all	Alt+A

As you type characters into the find text field, the code text area dynamically updates to reflect the found text. It highlights the found text, and it also adds a green marker to the vertical scrollbar, in a similar way to the error marker, drawn to the right of the error marker, e.g. the text ‘lresponsedetails’ is searched and highlighted in the above image.

After closing the Find (or Find and Replace) panel, you can still use Find Next and Find Previous, although the editor no longer highlights all matches.

Jump to Search or Error Item

On Windows, you can Ctrl-click in the scrollbar to jump to that position in the code text, i.e. Ctrl-clicking on a find or error marker goes to the search item or error in the code text. On macOS, the general system preference for scrolling can be set to Jump to the position that has been clicked, or you can Alt+click to achieve the same thing.

Line Menu

The options in the **Line** submenu replace several options in the Modify menu in previous versions, including Insert Line After, Insert Line Before, and Toggle Comment. Note that you can Right-click on the current or selected lines of code to open a context menu with similar options.

The Comment and Uncomment line options available in previous versions have been merged into a single **Toggle Comment** command, which has the single keyboard shortcut Ctrl+/ for commenting or uncommenting lines.

The Line Menu contains the new option **Select Line** which selects all the text in the current line (triple-clicking on a line also selects the line), and the **Delete Current Line** option which deletes the current line (containing the cursor or word selection), or all lines where multiple lines are selected.

The **Duplicate** option duplicates the current line (if no text is selected) or all selected lines, and places the duplicate line(s) immediately below the original line(s). The command also selects the duplicate text, which then allows you to use repeated Duplicate commands to generate multiple copies.

The **Goto Line Number** option opens a box to allow you to enter a line number to go to. You can show line numbers in the code area using the **Show Line Numbers** option in the View menu.

Selection Menu

The Modify>>**Selection** submenu contains new commands **Upper Case** and **Lower Case:** note that these options only change case for text that does not have a single canonical form, e.g. text in strings.

In addition, the Selection submenu contains the option **Select Word** which selects the word containing the insertion point, or where the insertion point is at the beginning or end of a word: in the latter case the word to the right or left of the insertion point is selected.

Method Editor Context Menu

The Method Editor context menu (opened when you right-click on the Code text area) has a new hierarchical menu called **Paste as.** You can use this to paste multiple lines of text from the clipboard into Sta:, Text: or JavaScript: commands. The Paste as hierarchical menu items are enabled when the caret is positioned on an empty line.

Keyboard Shortcuts

There are many keyboard shortcuts to allow you write Omnis code from the keyboard alone, without having to use the pointer. The most significant menu options in the Method Editor have keyboard shortcuts, including most of the options in the **Modify** and **Debug** menus, as well as the **Find and Replace** options.

The following keyboard shortcuts are available, but you should be aware that *several of them are context specific* so will only work if the focus is on a certain area in the Method Editor.

Windows shortcut	macOS shortcut	Description
Alt+A	Cmnd+Opt+A	Replace all in method
Alt+B	Cmnd+Opt+B	Disable breakpoint

Windows shortcut	macOS shortcut	Description
Alt+C	Cmnd+Opt+C	Match case
Alt+E	Cmnd+Opt+E	Enable breakpoint
Alt+F	Cmnd+Opt+F	Disable all breakpoints
Alt+G	Cmnd+Opt+G	Enable all breakpoints
Alt+H	Cmnd+Opt+H	Open Edit helper dialog
Alt+I	Cmnd+Opt+I	Debugger interrupt
Alt+J	Cmnd+Opt+J	Set list selection
Alt+K	Cmnd+Opt+K	Clear method stack
Alt+L	Cmnd+Opt+L	Set list current line
Alt+M	Cmnd+Opt+M	Toggle read-only mode
Alt+N	Cmnd+Opt+N	Toggle null and empty
Alt+R	Cmnd+Opt+R	Replace next in method
Alt+S	Cmnd+Opt+S	Save modified variable
Alt+T	Cmnd+Opt+T	Set breakpoint condition
Alt+U	Cmnd+Opt+U	Duplicate line
Alt+V	Cmnd+Opt+V	Go to debugger variables
Alt+W	Cmnd+Opt+W	Whole words
Alt+X	Cmnd+Opt+X	Regular expression
Alt+Y	Cmnd+Opt+Y	Side by side
Alt+Z	Cmnd+Opt+Z	Binary edit operations
Ctrl+/	Cmnd+/	Toggle comment
Ctrl+[	Cmnd+[	Move up stack
Ctrl+]	Cmnd+]	Move down stack
Ctrl+0	Cmnd+Opt+0	Go to task variables
Ctrl+1	Cmnd+Opt+1	Go to class variables
Ctrl+2	Cmnd+Opt+2	Go to instance variables
Ctrl+3	Cmnd+Opt+3	Go to local variables
Ctrl+4	Cmnd+Opt+4	Go to parameters
Ctrl+5	Cmnd+Opt+5	Go to documentation
Ctrl+6	Cmnd+Opt+6	Go to RESTful panel
Ctrl+7	Cmnd+Opt+7	Go to code
Ctrl+8	Cmnd+Opt+8	Go to method tree
Ctrl+D	Cmnd+D	Select word
Ctrl+E	Cmnd+E	Execute method
Ctrl+F	Cmnd+F	Find in method
Ctrl+G	Cmnd+G	Find next in method
Ctrl+H	Cmnd+H	Replace in method
Ctrl+I	Cmnd+I	Insert before
Ctrl+L	Cmnd+L	Go to line number
Ctrl+M	Cmnd+M	Insert method at end
Ctrl+N	Cmnd+N	Insert after
Ctrl+R	Cmnd+R	Next error
Ctrl+U	Cmnd+U	Lower case selection
Ctrl+Shift+B	Cmnd+Shift+B	Toggle breakpoint
Ctrl+Shift+C	Cmnd+Shift+C	Clear code breakpoints
Ctrl+Shift+D	Cmnd+Shift+D	Delete selected methods
Ctrl+Shift+E	Cmnd+Shift+E	Trace
Ctrl+Shift+G	Cmnd+Shift+G	Find previous in method
Ctrl+Shift+I	Cmnd+Shift+I	Inherit and override method
Ctrl+Shift+J	Cmnd+Shift+J	Clear variable breakpoints
Ctrl+Shift+K	Cmnd+Shift+K	Delete current line
Ctrl+Shift+L	Cmnd+Shift+L	Select line
Ctrl+Shift+M	Cmnd+Shift+M	Superclass methods
Ctrl+Shift+N	Cmnd+Shift+N	Show method tree
Ctrl+Shift+O	Cmnd+Shift+O	Toggle one-time breakpoint
Ctrl+Shift+R	Cmnd+Shift+R	Previous error
Ctrl+Shift+S	Cmnd+Shift+S	Step
Ctrl+Shift+T	Cmnd+Shift+T	Step out
Ctrl+Shift+U	Cmnd+Shift+U	Upper case selection
Ctrl+Shift+V	Cmnd+Shift+V	Step over
F1	F1	Opens the Omnis Help using the syntax item *under* the pointer
F3	F3	Modify this class

Windows shortcut	macOS shortcut	Description
F5	F5	Go point
F7	F7	Fix error
F8	F8	Modify specified class
F10	F10	Method history backwards
Shift+F1	Shift+F1	Opens the Omnis Help using the syntax item *under* the pointer
Shift+F2	Shift+F2	Set Go point
Shift+F4	Shift+F4	Pin bottom panel
Shift+F5	Shift+F5	Hide bottom panel
Shift+F6	Shift+F6	Show editor panel
Shift+F7	Shift+F7	Show debug panel
Shift+F9	Shift+F9	Show Variable panel
Shift+F10	Shift+F10	Method history forwards

Keyboard Shortcut Configuration

The keyboard shortcuts are stored in the **$keys** property in the Omnis Preferences ($prefs), which you can edit in the Property Manager to change the keyboard shortcuts. Note this feature replaces the Edit Keys option on the Debug menu in previous versions, and it also contains the keyboard shortcuts for Edit fields and the Edit menu.

The first time you edit $keys *and press OK*, Omnis generates a file called **keys.json** in the Studio folder, that records the configuration of the keyboard shortcuts (as listed above): if you don't make any changes in $keys the default keyboard shortcuts will be stored in keys.json.

You can edit the Shortcut Keys options by selecting **$keys** in the Property Manager (find it under the Omnis Preferences in the Studio Browser), then select **'methodEditorAndRemoteDebugger'.**

To edit a value, you can use the **Delete** or **Backspace** key to clear the current shortcut, and then type the desired shortcut key combination. You can use all the standard Key modifiers (Ctrl, Cmnd, Alt, Option, Shift, etc) as well as all the letter and number keys, plus the numbered Function keys. In addition, you can use the **Enter** and **Return** keys in conjunction with Ctrl/Cmnd, and optionally Shift or Alt/Option, for method editor menu shortcuts.

The $keys preference also contains the shortcut keys for Edit fields (editFields), which are documented under the JavaScript Edit Control, and the Edit menu (editMenu) which has the following shortcut keys:

Shortcut Key	Description	keys.json item
Ctrl+Y	Redo last operation	Redo
Ctrl+Shift+F	Find and Replace	findAndReplace
Ctrl+Shift+G	Find Next	findNext

Word Selection

You can double-click on a word to select it, or double-click and drag the pointer to select multiple words. If you double-click on a single word that is enclosed in quotes (e.g. like the foo in *Calculate lcVar as "foo"),* the quotes *will not be* selected. In previous versions the quotes would have been selected, but if want to enable the old behavior you can set a new option "entryFieldsIncludeQuotesWhenSelectingWords" in the "defaults" section of config.json to true: the option defaults to false which enables the new behaviour.

Commenting / Uncommenting Lines

You can comment or uncomment a single method line by clicking anywhere in the line (or you can select the whole line) and selecting the **Toggle Comment** option, or press the Ctrl+/ shortcut. To comment or uncomment multiple lines, you need to select all the lines and then use the Toggle Comment option: in this case, all the affected lines will remain selected after toggling their comment state. Commenting a *single empty line* does not select the commented line: in this case (and when "Move to next line after toggle comment" is off, see below), the caret is positioned after the comment character and the space, ready for you to type the comment.

You can force the cursor to move down to the line after the commented/uncommented line or block of selected lines by enabling the **Move To Next Line After Toggle Comment** option in the Line menu (the option is off by default): the state of this option is saved with the Window Setup.

Empty method lines are not commented out when using the Toggle comment command or Shortcut key: this applies when multiple selected lines may include empty lines.

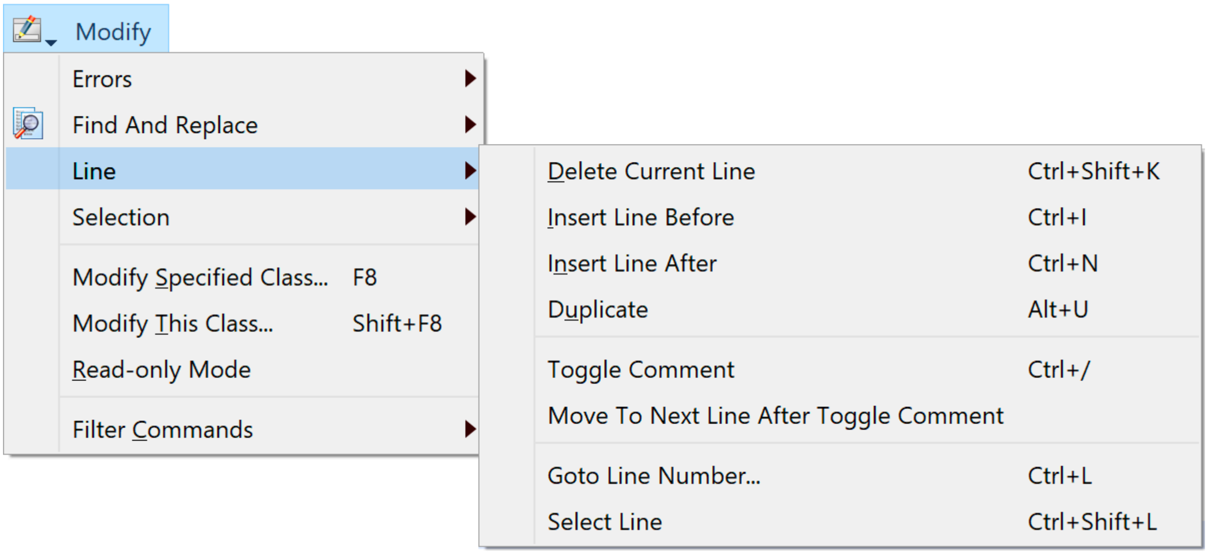

Modify
Errors
Find And Replace
Line
Selection
Modify Specified Class... F8
Modify This Class... Shift+F8
Read-only Mode
Filter Commands
Delete Current Line Ctrl+Shift+K
Insert Line Before Ctrl+I
Insert Line After Ctrl+N
Duplicate Alt+U
Toggle Comment Ctrl+/
Move To Next Line After Toggle Comment
Goto Line Number... Ctrl+L
Select Line Ctrl+Shift+L

Figure 106:

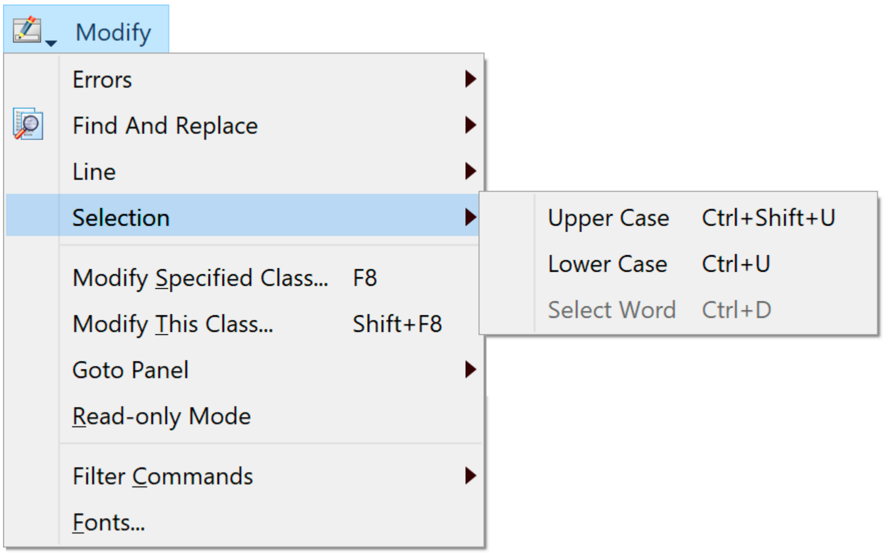


Figure 107:

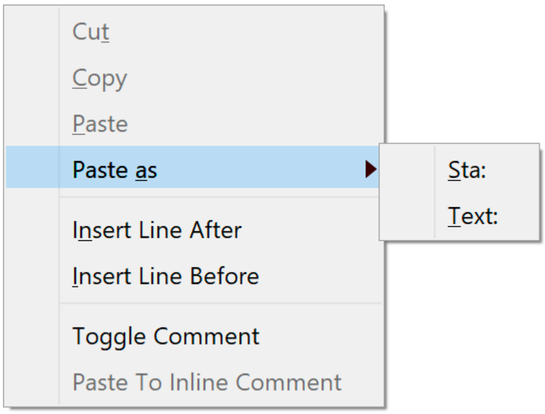
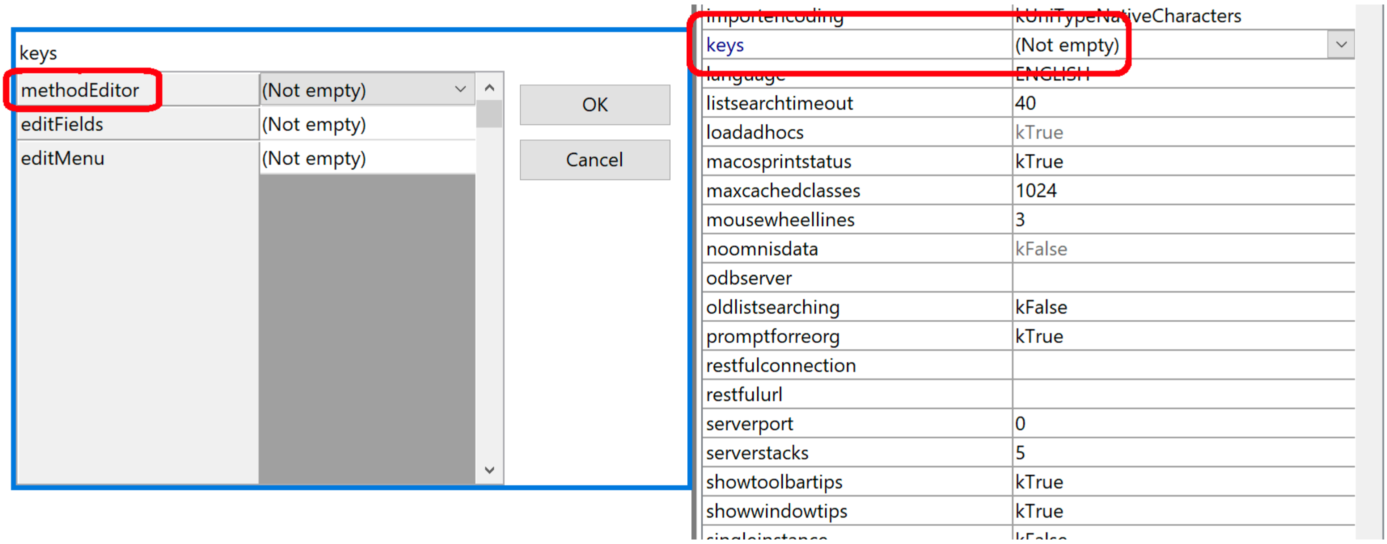


Figure 108:



keys

methodEditor	(Not empty)
editFields	(Not empty)
editMenu	(Not empty)

importencoding	kUniTypeNativeCharacters
keys	(Not empty)
language	ENGLISH
listsearchtimeout	40
loadadhocs	kTrue
macosprintstatus	kTrue
maxcachedclasses	1024
mousewheellines	3
noomnisdata	kFalse
odbserver	
oldlistsearching	kFalse
promptforreorg	kTrue
restfulconnection	
restfulurl	
serverport	0
serverstacks	5
showtoolbartips	kTrue
showwindowtips	kTrue
singleinstance	kFalse

Figure 109:

Language Syntax

There are a number of changes to the Omnis language syntax that facilitate direct text entry of commands, and which enable the new Omnis Studio 10.0 language parser to function properly.

Language Keywords

The following language keywords cannot be used as variable names:

as	at	flag
for	from	into
on	returns	sec
step	to	until

During library conversion (to Studio 10 or above), any variable names using these keywords are appended with an integer starting at 1.

Options

Omnis stores the order in which "checkbox" and "radio button" command options are specified as part of the method command (remember that the Omnis language is tokenized, and does not store raw text as entered by the developer). This allows you to enter the options as text in any order.

The "Select matches (OR)" and "Deselect non-matches (AND)" options of the Search list command have been renamed to "Select matches OR" and "Deselect non-matches AND". This prevents the parentheses in these option names from interfering with language parsing.

Braces

Braces have been removed from all commands, except for commands like OK message, which require three components (a field name or square bracket calculation, options and a calculation). For these commands, when they use square bracket calculations, you must escape ( ) { } characters in the calculation *outside square brackets* if there is no text *after* the parentheses. In this case, these characters need to be escaped using square bracket notation, e.g. ['('] escapes (.

Entering Quotes, Braces, and Square Brackets

When you have used an opening quote, or an open brace {, and then typed some parameters, the Code Editor adds the appropriate closing character.

However, when entering an open square bracket in the Sta: command, the *close square bracket* (]) is not added automatically.

When you split a text block command parameter using Return (carriage return) the "Sta:" command prefix is inserted into the text block automatically.

Text: and parenthesis

If Omnis encounters an open bracket (at the end of a command line, it prompts for options (Carriage return etc). If there is another character after the (, without a trailing comma, Omnis stops looking for options, and treats the characters as text. This leaves the special case of (on its own at the end of the text. You can enter this using square bracket notation with a constant [kOpenParen]. There is also a kCloseParen constant.

Unicode Characters

The Code Editor selects a smaller font size, if necessary, for all Unicode characters >= 0x250 contained in a string. On retina displays (on Win and macOS), the Code Editor uses the default font. On non-retina displays, it may be necessary to increase the font size to get a reasonable display of Unicode characters.

Character Constants

You can insert the # character, as well as left and right square bracket into a string/text using the constants kHash, kLeftSB and kRightSB. If you wish to create a constant for double hash, you can initialise a variable with the value con(kHash,kHash).

Comments

To enter a new comment on an empty line, you can type # and then the comment text, with or without a space after the #. (For backwards compatibility, you can also type ; to create a new comment, but the comment is marked with #).

To enter an inline comment, press the space key followed by ## at the end of a code line, and then enter the comment text. Inline comments are positioned over on the right of the code entry area: they are left-tab aligned according to a tab which is indicated by a small marker at the top of the code entry area: you can drag this marker to reset the tab position.

The Sta:, Text: and JavaScript: commands (that generate a text block) no longer allow inline comments (see note in *Library Conversion* section about inline comments for the Sta: command). This allows all text after a colon to be treated as significant text, and to be added to the text block, with the exception of the options string specifying the line delimiter for the Text: command.

If you want to include “space##” in a string you need to enter <space>## in the string and it will not be interpreted as an inline comment.

Commenting and Uncommenting code

You can “comment” or “uncomment” the current method line (containing the cursor) or any selected method line or lines using the **Toggle Comment** option in the Modify menu, or using the keyboard shortcut **Ctrl-/** (forward slash) – note the same menu option or keypress can be used to both *comment* or *uncomment* method lines or comments as appropriate. Commented lines must have valid syntax to be uncommented, otherwise they will remain commented out.

Errors

As the new Method Editor allows any text to be entered, it is possible to enter and store commands that contain errors. Internally, these are stored in the method with a new command type, and will cause an error to be reported if you try to export the method to JSON, or if you try to execute them.

The Find and replace dialog has a new option (Only search method lines containing an error), which you can use to find commands with an error. When you check this option, the dialog also checks the regular expression option, and sets the find string to the regular expression “.*”.

In general, there should not be much need to leave erroneous commands stored in a method for very long - the editor gives immediate feedback about errors, so in practice it makes sense to fix them as you code. The Find and replace dialog option provides a means to double check that all is well with a library. Omnis Studio 10.0 takes this approach (rather than for example marking all classes with an error count) since errors should be very much an exceptional case once coding of a method is complete.

Modified Commands

The step interval for the *For* command is assumed to be 1, so when entering a *For* loop and you want the step interval to be 1, you no longer need to enter this. If you need a step interval other than 1 you need to enter this into your code.

Obsolete Commands

Any commands that were marked as ‘Obsolete Commands’ in previous versions (listed in the ‘Obsolete Commands…’ group) have been removed from the Omnis language and are shown commented out in your Omnis code. The *Translate input/output* command is also obsolete and will be commented out.

The *Call method* OBSOLETE COMMAND will be replaced by the *Do code method* command and the method name.

There is a complete list of obsolete commands that have been deleted in this version in Appendix A in this manual.

Library Conversion

The changes in language syntax mean that Omnis performs a class-by-class conversion of a library created using Omnis Studio 8.1.x or earlier. The following items are converted:

- Any obsolete commands are commented out. In previous versions these commands were marked with the “OBSOLETE COMMAND” suffix and listed in the ‘Obsolete commands’ group, and some of these commands have been removed from Omnis, so cannot be used in your code. The Appendix in this manual lists all of the obsolete commands that have been removed in Studio 10.x or above and will be commented out.
- The prefix for comments is now #, converted from ;
  A space is inserted after the # at the start of comments, therefore comments are # abc rather than #abc after conversion.
- Inline comments for JavaScript:, Text:, and Sta: commands are no longer allowed, since *all the text after the :* is treated as part of the statement or text. Therefore, on conversion, all inline comments are moved to the next line and inserted as a standard comment (see below).
- Square bracket calculations (ctySquare, ctyParmlist4 etc) are converted so that any text outside square brackets does not contain unescaped characters () {}.
- Any instances of ” ##” are detected in method lines and reported as a warning (they are probably not editable as the parser will treat the text after this sequence as the inline comment).
- Any variables which are named using a *language keyword* (see earlier for a list) are renamed, by appending an integer to them (starting with 1 until a new unique name in its context is created).

Inline Comments for JavaScript:, Text: and Sta: commands

By default, the conversion process will move all inline comments from JavaScript:, Text: and Sta: commands to the *next line in the method*, after the original line containing the inline comment. There are three new options in the “ide” section of config.json to allow you to control how inline comments are treated.

- **“libConverterAppendsDiscardedInlineComments”**
  When true (the default), if the inline comment would otherwise be discarded, the converter appends a comment command after the JavaScript:, Text: or Sta: command, containing the inline comment.
- **“libConverterAddsInlineCommentToStaCommandParameter”**
  Note that if you use “libConverterAddsInlineCommentToStaCommandParameter” to convert inline comments for Sta: commands, then this option will not affect Sta: commands: see below.
- **“libConverterInsertsDiscardedInlineComments”**
  moves and inserts the inline comment *before the original line* containing the inline comment (however, if libConverterAppendsDiscardedInlineComments is set to true, libConverterInsertsDiscardedInlineComments is ignored).

Inline Comments Sta: commands

If you want to keep inline comments as part of the SQL text for Sta: commands, you can set the item “libConverterAddsInlineCommentToStaCommandParameter” in the ‘ide’ section of config.json to a formatting string, e.g. ” – %” or ” /* % */“. Omnis replaces the first % place-holder in the formatting string with the inline comment, and appends the resulting string to the parameter of the Sta: command. Note that if the resulting text does not tokenize, e.g. if the inline comment contains text like [#S333] which does not tokenize, then the comment will be discarded.

If you leave “libConverterAddsInlineCommentToStaCommandParameter” empty (or supply a string that does not include the % character), then Omnis will discard the inline comment when converting Sta: commands.

SQL comments for the Sta: command are colored, including /* */ and – comments. The “syntaxColorProbableSQLComments” option in the ‘ide’ section of config.json is enabled by default, but can be set to kFalse to disable coloring.

Call Method OBSOLETE COMMAND

The *Call method OBSOLETE COMMAND* is converted to the *Do code method* command during conversion.

Set return value OBSOLETE COMMAND

During conversion, consecutive *Set return value OBSOLETE COMMAND* ***value*** and *Quit method* commands (the latter with an empty parameter) are combined into a single command *Quit method* ***value.*** Note that when checking for consecutive commands, Omnis skips comments and empty lines.

Library Conversion Logs

The converter adds an entry to the Find and Replace log that allows you to quickly navigate to each change made by the converter by double-clicking on a line in the log. In addition, the converter writes a log file to the 'conversion' folder in the logs folder in the data part of the Studio tree. The log file provides a more permanent record of the changes applied to the converted library. Note that Omnis does not write log entries to record where spaces were inserted at the start of comments.

JSON generated libraries

When Studio 10 imports JSON generated with Studio 8.1, it parses methods using the old Studio 8.1 parser, and then applies the same conversion steps as above to the imported classes. Changes applied by this conversion are written to the Find and Replace log only.

Method Editor Coloring

The colors used in the Method Editor window can be changed by changing the theme in the Hub>>Options in the Studio Browser, or configured by editing the $appearance preference in the Property Manager (these are stored in appearance.json and the various theme files): the method editor colors are stored in the IDEmethodEditor group in the appearance.json file. The following theme colors (and settings) are available:

Color option	Description
methodcurrentlinebackgroundcolor	The background color used to display the line containing the caret in the Meth
methodeditorcodebackgroundcolor	The background color for the method editor code area
methodeditorcodereadonlybackgroundcolor	The background color for the method editor code area in read-only mode
methodeditorcodeleftmarginbackgroundcolor	The background color for the left margin in the Code Editor (where the Go poi
methodhighlightcolor	The color of selected method text in the Method Editor when the control displ focus
methodhighlightnofocuscolor	The color of selected method text in the Method Editor when the control displ have the focus
methodeditorfadealpha	Value 0-255; the fade level of the method editor when editing a variable value
overriddenmethodstyle	Overridden method style
overriddenbuiltinmethodstyle	Overridden built-in method style
syntaxwordhighlightcolor	Color used to highlight syntax elements, e.g. click on a variable name in the co of the variable
codeassistantpopupcolor	Background color of Code assistant popup
treelinesmatchingsearchcolor	Background color of unselected method editor tree lines that match the curre
executionpositioncolor	A line is drawn above and below the Go point line and Call stack return point u

Syntax Coloring

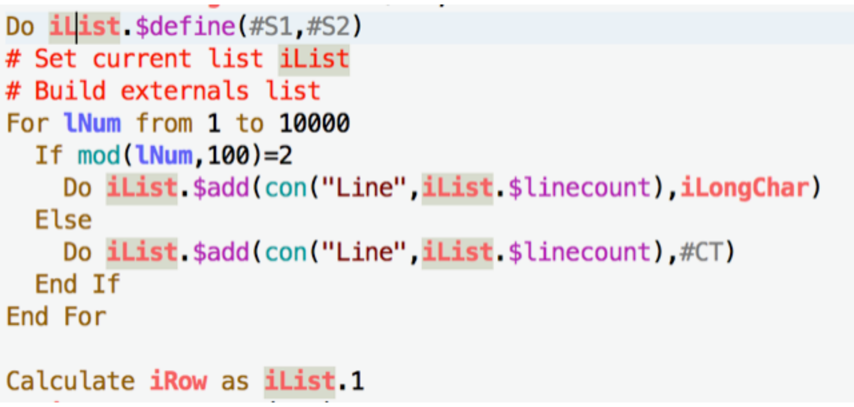
The colors used in the Chroma Coding or code syntax in the Method Editor can be changed by changing the theme in the Hub>>Options in the Studio Browser, or configured by editing the $appearance preference in the Property Manager (these are stored in appearance.json and the various theme files): the method syntax colors are stored in the IDEmethodSyntax group in the appearance.json file. The following theme colors (and settings) are available:

Color option	Description
badsyntaxcolor	bad method syntax indicators
bracketbackcolorbracketcolor	Brackets color and background color
classvariablecolorclassvairablestyle	Class variables
commentcolorcommentstyle	Comments
constantcolorconstantstyle	Constants (e.g. kTrue)

Color option	Description
ctrlkeywordcolorctrlkeywordstyle	Ctrl keyword
currentblockcolorcurrentblockstyle	Current block
eventparametercoloreventparameterstyle	Event parameter variables
functioncolorfunctionstyle	Built-in and external functions
hashvariablecolorhashvariablestyle	Hash variables
instancevariablecolorinstancevariablestyle	Instance variables
keywordcolorkeywordstyle	Keywords
localvariablecolorlocalvariablestyle	Local variables
methodothertextcolor	Color for all other text with no specific syntax color, e.g. separators, dots, etc.
notationcolornotationstyle	Built-in notation attributes
optioncoloroptionstyle	Command options (e.g. Sound bell for OK message; corresponding to check boxes o 10 editor)
parametervariablecolorparametervariablestyle	Method parameter variables
resolvednamecolorresolvednamestyleunesolvednamecolorunresolvednamestyleuseresolvednamecolorsandstyles	Field names and parameters that are “resolved” or “unresolved”; see below
stringcolorstringstyle	Strings
taskvariablecolortaskvariablestyle	Task variables
variablecolorvariablestyle	Color for other variables, including file class variables (field names), and other compo list column name

Syntax Highlighting

When you click in a syntax element (variable, notation name, command name (not block commands) or function name), the code editor performs a find and highlights instances of the element in the current method (note the find highlighting will override the syntax highlighting if the Find or Find and Replace panel is displayed).



```
Do iList.$define(#S1,#S2)
# Set current list iList
# Build externals list
For lNum from 1 to 10000
  If mod(lNum,100)=2
    Do iList.$add(con("Line",iList.$linecount),iLongChar)
  Else
    Do iList.$add(con("Line",iList.$linecount),#CT)
  End If
End For

Calculate iRow as iList.1
```

Figure 110:

The view menu contains the option “Highlight Syntax Words” which is checked by default. There is a new color option“syntaxwordhighlightcolor” in the “IDEmethodEditor” group in the $appearance Omnis preference, and stored in the appearance.json file.

Resolved Name Colors

There are colors and styles to highlight field names and parameters that are “resolved” or “unresolved” for certain commands that reference field names and notation group members. The Code Editor can (defaults to on) display names it has resolved using **resolvednamecolor** and **resolvednamestyle,** and names it has failed to resolve using **unresolvednamecolor** and **unresolvednamestyle;** all members are in the IDEmethodSyntax section of appearance.json.

If useresolvednamecolorsandstyles is true, the Code Editor tries to resolve certain names, and if successful draws them using the resolvedname color and style; if unsuccessful it draws them using the unresolvedname color and style.

Examples of where this applies are the parameters of the *Redraw* command, *Queue set current field* command, names in notation such as $cinst.$objs.name, and method names in calls such as $cinst.$mymethod().

If a name is displayed using the unresolvedname color and style it does not necessarily mean there is an issue, e.g. it could be a notation reference such as $cinst.$objs.name, where the object is dynamically added and named at runtime.

JavaScript: Editor

In addition to the main interface changes in the Method Editor, a JavaScript editor has been added to the Method Editor in Studio 10 to allow you to enter a whole block of JavaScript code directly into the *JavaScript:* command, rather than line by line as in previous versions. The new JavaScript editor will popup whenever you edit a command line containing the *JavaScript:* command. The editor also allows you to enter a SQL statement if the *Sta:* command is selected: in this case, the editor will switch to SQL mode.

To edit or enter some JavaScript, click into or tab to a *JavaScript:* command line in the text entry panel, or select a whole *JavaScript:* command line or multiple lines, and either

- Press Alt+H to open the JavaScript editor, which in this case is the same as clicking on the Helper dialog button at the bottom of the Method Editor window
- Or select *Open JavaScript Editor* from the **Modify>>Selection** submenu

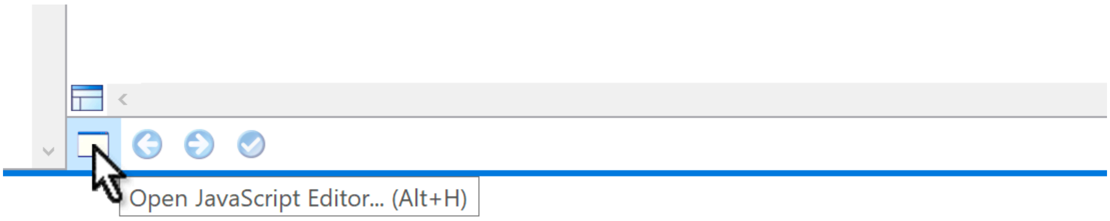
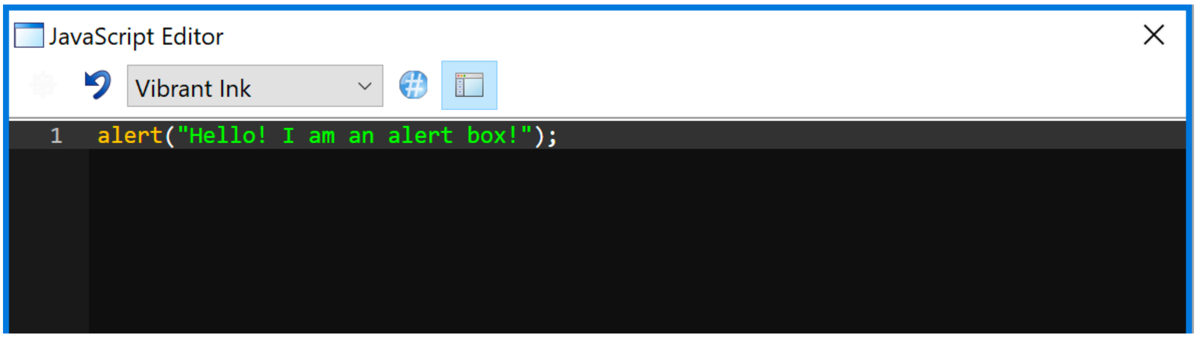


Figure 111:

The content of the JavaScript editor is formed by concatenating the contiguous JavaScript: command(s) that are selected in the list. This allows you to edit or insert a contiguous sequence of these commands as a block. Omnis selects unselected lines in this contiguous block when it opens the window, so all the lines are selected when viewed behind the editor window. When you have finished editing the JavaScript: text, you can close the editor window and Omnis replaces the selected JavaScript: commands with the new content, creating a JavaScript: command line for each line of JavaScript.



```
alert("Hello! I am an alert box!");
```

Figure 112:

The editor window allows you to change the theme of the displayed text, and to revert to the original text.

Spaces & End of Line Characters

There are two new options in the Method Editor **View** menu to allow you to show Trailing Spaces and End Of Line characters when editing text in the new popup JavaScript or SQL text editor:

- **Show Significant Trailing Spaces**
  If true, the editor displays trailing spaces for the JavaScript:, Sta: and Text: commands as the Unicode sp symbol.
- **Show Selected End of Line As Symbol**
  If true, the editor displays the end of line character as Unicode symbol cr when the end of line character is selected. This allows you to see if an end of line character will be added to the clipboard by a cut or copy, for example.

Both of the new options default to true and are saved with the window setup.

Trace Log

The *Send to trace log* command includes the name of the method that issued the command in column -Double clicking on the trace log line takes you to the code line that issued the command.

In addition, the *Send to trace log* command has the "Always log" option. If specified, the command will always log the message even if $nodebug is true for the library or the local debugger is disabled (this option is ignored for a diagnostic message).

Error Processing

There is a library preference $clib.$prefs.$errorprocessing that allows you to control how Omnis behaves when it encounters an error: Omnis either enters the debugger (if available) or reports the error with an OK message.

- **$errorprocessing**
  A kEP… constant that indicates how unhandled errors in methods belonging to this library are processed. Values of the kEP… constant are:
  kEPreport: Report the error by opening the debugger if available or by displaying a message box (the default value after converting a library to Studio 10.x)
  kEPlogStackAndReport: Log the call stack to the trace log and then report the error by opening the debugger if available or by displaying a message box
  kEPlogStackAndContinue: Log the call stack to the trace log and then continue execution with the next method command

The call stack written to the trace log is drawn using the "bad syntax" color from the appearance settings. Each line contains the error code and error text, and then the call that invoked the error (shown in one line in Studio 10.x). You can double-click on a line to open the method at the relevant method line, provided that the library is not marked as always private and the class is not protected. The call stack excludes entries from methods running in tasks marked as IDE tasks which have their code in an always private library.

Dynamic Methods & Objects

The handling of dynamically added or modified methods, and dynamically added and removed objects has been improved.

- The stack list has a new menu item, to detach the debugger from an instance. Previously this was only possible by force-closing the current debug instance.
- The debugger tree lazily updates to show new or deleted objects in the current debug instance: typically, this means it updates either when the debugger window comes to the front, or while you are stepping through code.
- When an instance closes, or an object is removed, Omnis deletes any breakpoints set in a method in a freed temporary instance field method, and removes all of its methods (if any) from the method editor history stack used by the back and forward navigation buttons.
- Omnis marks each temporary method (i.e. instance method), using a new icon so you can recognise these easily in the tree. If you edit such a method, the edits are saved with the instance, and will be lost when the instance destructs.
- You cannot rename a temporary instance object shown in the method editor tree.

Code Assistant

The **Code Assistant** is an integral part of the Code Editor, but it is described separately here since it has so many useful features to help you write better code and faster. The Code Assistant *opens automatically* at the insertion point when you type a command name, a command parameter, a variable name, or some notation in the Code Editor or you can open it at any appropriate point in your code using **Ctrl-Space.** The Code Assistant will usually drop down below the insertion point, but may pop up if space below the cursor is limited.

The Code Assistant only opens *when the caret is visible* in the method editor, i.e. it can only open when no text is selected in a line of code. Specifically, it will pop up when the caret is positioned at *the end of some text* which is either at the end of the entry field content in the method editor, or prior to some type of delimiter in the expression syntax, e.g. a function separator character. The automatic popup is delayed by a timer which is specified in an Omnis preference called $codeassistanttimer (in Omnis.$prefs).

Further highlights of the Code Assistant include:

- In addition to Ctrl-Space, you can use various special keys to navigate the popped list and request further help.

- It provides assistance entering notation relative to an item reference and functions.
- It displays method descriptions, method definitions and parameter descriptions.
- Assistance entering notation relative to a group method, as well as notation relative to $ref in the parameters of a group method.
- Intelligent generation of the list of possible values to assign to a property.
- Property and method tips in the method list.
- Assistance for initial values and when using expand entry-box in the method editor.
- An improved expand entry-box interface.
- Replaces existing data when selecting an item in the Code Assistant popup.
- Assistance entering certain commands such as *Do method*.
- Parameter highlighting, including parameters for commands such as SMTPSend.
- Parenthesis and square bracket matching.
- Assistance entering methods with overloaded definitions.

When the Code Assistant is opened, items in the help list are listed in the following order:

- Variables or names,
- Functions,
- Notation (properties & methods),
- Constants,
- Events

These are sorted alphabetically within each set. This ordering was introduced in Studio 11, so if you want to use the old sorting you can set the ‘oldSortOrder’ item in the ‘codeAssistant’ section of config.json to true (the default is false).

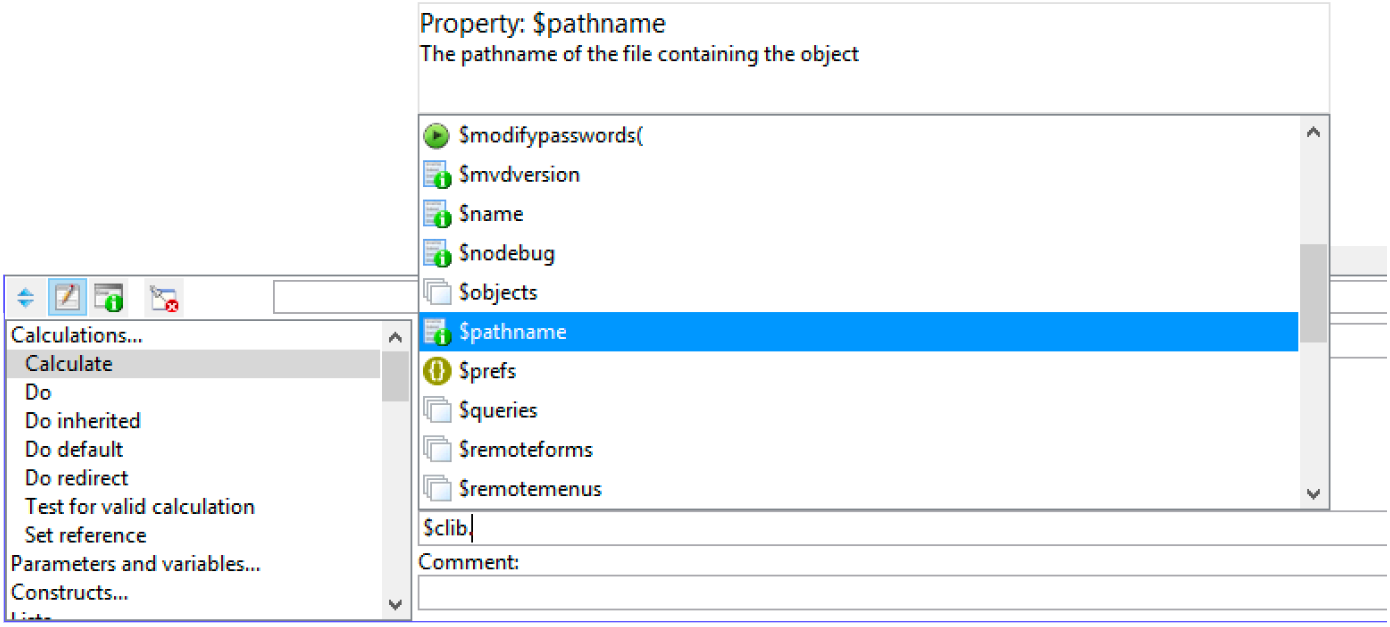
Shortcut Keys and Help

You can manually request the Code Assistant popup to open by typing **Ctrl-Space**: this will work on Windows and macOS. The Ctrl-Space shortcut key will only work if some code assistance is available for the syntax item *to the left of the current insertion point.* This short cut key is a de-facto standard used to request code assistance in many other development tools so should be familiar to developers.

The Code Assistant supports the Page up, Page down, Home and End keys, to navigate the popped up list. When you use these keys, or Up Arrow or Down Arrow, the Code Assistant displays help information about the currently selected line in a help panel above the popped list, for example, the following image shows the help text for $pathname which is a property of the current library ($clib).

Short Cut Key Summary

Key	Action
Ctrl-Space	Opens the Code Assistant
Page up, Page down	Displays next or previous ‘page’ in the popup list or Help pane
Home and End keys	Moves to the beginning or end of the popup list
Up or Down Arrow	Moves up or down the popup list
Return or Enter	Select the current line in the popup list
Shift-Space	Initiates a ‘fuzzy’ search at the insertion point



Property: $pathname
The pathname of the file containing the object

Figure 113:

Fuzzy Search

You can initiate a 'fuzzy' search at the insertion point by entering a search string and pressing **Shift-Space.** The Code Assistant will look for methods, properties, etc, that 'contain' your search string, rather than the default 'starts with'. For example, when searching for the color properties for a Description label in your code:

```
Do $cinst.$objs.Description.$col
```

you could type ".$col" then press **Shift-Space,** and the Code Assistant will popup containing the properties $backcolor and $textcolor. In this case, the Code Assistant will search for properties of a Label containing "col": without the fuzzy search shortcut key there would be no search results.

Tabbing Behavior

There is an item 'tabAlsoLeavesFieldAfterClosingAssistant' in the 'codeAssistant' entry in the Omnis congfi.json file that affects the tabbing behaviour in the Code Assistant. It is set to false by default, but if set to true (and 'oldTabReturnBehaviour' is false) then a tab will close the Code Assistant and the cursor will move to the next field in the tabbing order when tabbing out of the variable name or calculation box in the method editor.

Code Assistant Configuration

There are a number of settings in the "codeAssistant" section of the Omnis configuration file (congfig.json) that allow you to control the behavior and appearance of the Code Assistant or specifically the parameter help panel.

- **parameterHelpEnabled**
  boolean property, default is true. Enables or disables the Code Assistant.
- **maxParameterHelpWidth**
  specifies the maximum width of the parameter help popup (it defaults to half the screen width)
- **parameterHelpExclusions**
  specifies which functions or methods you wish to exclude from parameter help (the default is empty, that is, no items are excluded). For example, you might want to exclude the function "con" or the notation "$assign".
- **width**
  specifies the width of the Code Assistant window (the default is 768 pixels); the value must be between 512 and 1536 inclusive
- **openParameterHelpWithCodeAssistantPopup**
  If true (the default), the code assistant and parameter help window both open on the same side (above or below the text being entered). If false, they open on opposite sides.

- **parameterHelpSpace**
  Defaults to 40 (pixels). Space for parameter help on the same side of the text as the code assistant popup; applies when openParameterHelpWithCodeAssistantPopup is true

- **listShowsNamesFirst**
  If true (the default), method names occur in the Code Assistant list before attributes etc that start with $. When false, $ entries typically occur before names

What Help does the Code Assistant Provide?

In most cases the Code Assistant will popup automatically at the cursor if it can provide help for the current item in your code or context, however the following sections detail the behavior and function of the assistant with regards to different items or contexts in which Omnis provides you with help.

Item References and Notation

In order to provide code assistance, Omnis needs to be able to look up a notation string and map it to the table of methods and properties that apply to the current addressed item. In order to do this for notation paths that start with an item reference, Omnis needs a new piece of information that identifies the notation you intend to use with the item reference – this item is called the *item reference class* and the method editor allows you to select an item reference class as the subtype of an item reference. The class works in the same way as the subtype of an object reference, meaning that the item reference class is solely used to provide code assistance – no check is ever made to see if the item reference is being used at runtime to address items that correspond to its class.

Item reference classes use a similar hierarchical scheme to notation paths. Example classes are $iwindows.window, and $iwindows.window.$objs. There are some special classes that include * in their path. For example,

```
$iwindows.window.$objs.*
```

accumulates all possible properties and methods for the possible children of $objs (there is a child for each object type), and is used when the Code Assistant cannot isolate the class of the member of $objs to a single object type.

$iwindows.window.$objs.*.$objs accumulates all $add, $addafter and $addbefore methods for all containers, and is used when the Code Assistant cannot determine the type of a container.

Code assistance for notation works as follows:

- The Code Assistant takes the notation path (and the item reference class if necessary and available) and looks up the matching item reference class.
- If it cannot determine a class, then the Code Assistant provides no assistance.
- If the Code Assistant can determine a class, then it pops up the methods and properties that match the currently entered prefix.

Functions

Code assistance is available for functions, including the static methods implemented by external components such as the FileOps external object. The latter is provided by a two-step process, where you first select the component from a popup, such as FileOps.$, and you then select the method from an automatically popped up list of static methods.

The Code Assistant will show a full help page for the function, or a short text description if a help page does not exist for the function. If you do not want the full help pages to display for functions you can set the useOmnisHelpPagesForFunctionHelp entry in the codeAssistant section of config.json to false.

Do command

The Code assistant adds matching commands when entering what could be command parameters, e.g. when entering probable parameters for Do, the Code Assistant will also add Do method, etc, commands to the list.

Note that when typing command names, you can omit spaces and the command will be found more easily, for example, to find *Do inherited,* you can type doi.

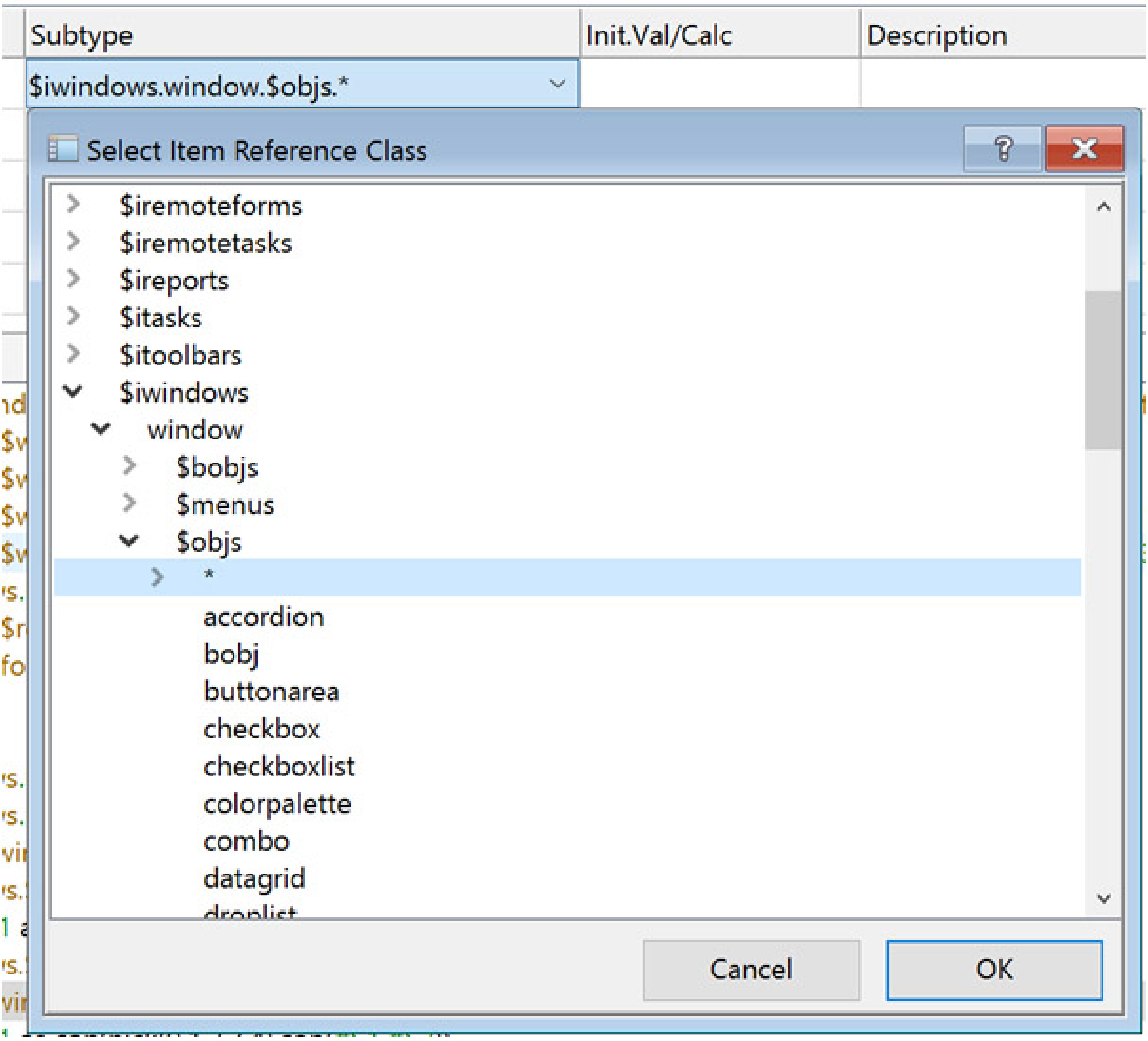
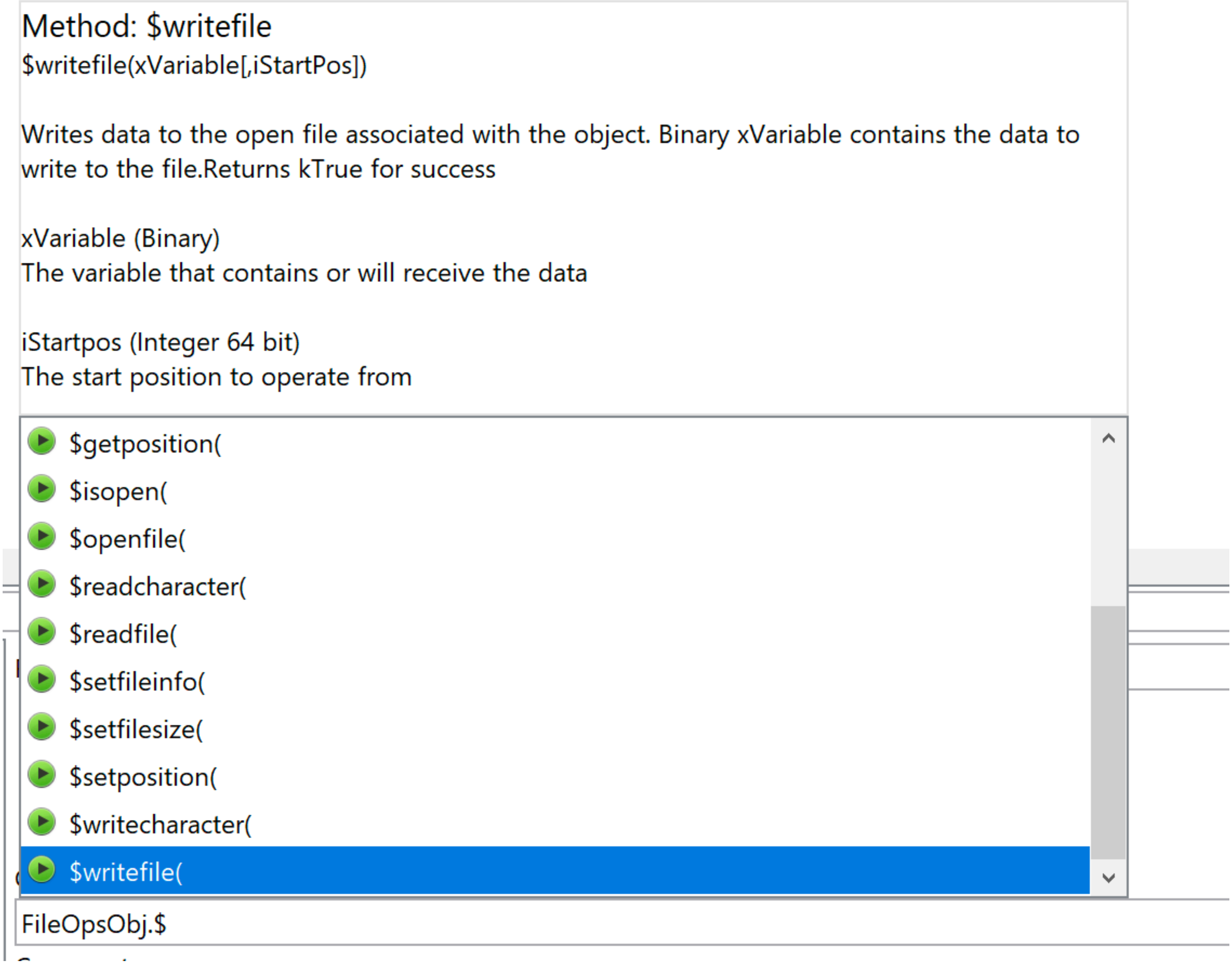

Subtype
Init.Val/Calc
Description
$iwindows.window.$objs.*
Select Item Reference Class
$iremoteforms
$iremotetasks
$ireports
$itasks
$itoolbars
$iwindows
window
$bobjs
$menus
$objs
*
accordion
bobj
buttonarea
checkbox
checkboxlist
colorpalette
combo
datagrid
droplist
Cancel
OK

Figure 114:

Method Information

The Code Assistant displays method descriptions and parameter information in the help panel when a method is selected in the popup. This information is available for all types of method, including functions, external component methods, built-in Omnis notation methods, and your own custom methods.

If you highlight the method name using the up and down arrow keys, a full description is shown in the help pane in the Code assistant popup: for example, the following shows the help for the $writefile FileOps method:



Method: $writefile

$writefile(xVariable[,iStartPos])

Writes data to the open file associated with the object. Binary xVariable contains the data to write to the file.Returns kTrue for success

xVariable (Binary)

The variable that contains or will receive the data

iStartpos (Integer 64 bit)

The start position to operate from

Figure 115:

Group Methods

Methods such as $add and $findname for a notation group return an item reference to a member of the group (assuming they work). The Code Assistant assumes that the call will work, and provides assistance for notation entered after the group method, e.g. if you enter $cinst.$objs.$add(kEntry,0,0,100,100), then as soon as you enter a dot (.) after this expression, you get assistance for all objects that could be in the group (Omnis does not parse the $add call and attempt to provide help for the specific object type added).

$obj and $field

Code assistance is provided for $cinst.$field to allow code assistance to be provided without making $cinst.$objs harder to enter; $cinst.$field is equivalent to $cinst.$obj, but $field only works for subform and subwindow instances.

The notation $field behaves the same as $obj when used with a subform or subwindow instance, for remote forms and windows only. This allows code assistance to be better targeted, and also prevents $obj from taking over as the first property in the code assistant list after typing “$cinst.$o”.

$ref

When you use group methods such as $sendall or $makelist, you use $ref in the parameters of the group method to refer to a member of the group. The Code Assistant provides help for $ref, by using the relevant item reference class for the group member (provided it can identify the item reference class of the group).

$assign

When you enter . after a property name, the Code Assistant provides $assign and $canassign as possible options. If you select $assign, you will be prompted with a popup that provides either all initial items you can enter, or the list of constants or strings which make sense to assign to the property. The latter always applies when the Code Assistant can determine the list of constants or strings which make sense.

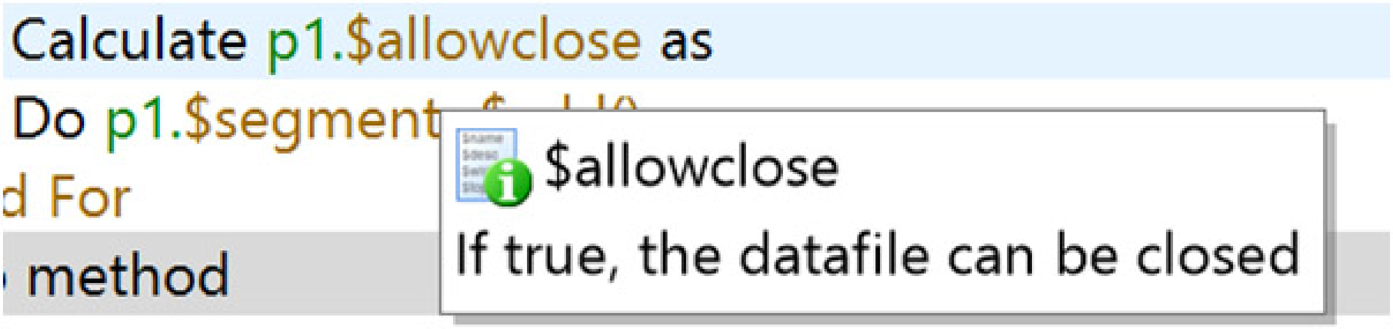
In addition, when you are coding a Calculate statement, if you enter a path to a property in the field name field in the method editor, then when you move to the calculation field, provided that the calculate field is empty, the Code Assistant will pop up the list of constants or strings which make sense to assign to the property.

For example, enter $cwind.$objs.test.$backcolor as the field name, and move to the empty calculation field. The popup will contain a list of color constants.

If you wish to assign something else, start typing that, and assistance will revert to normal. The only restriction here is that if you type k (when the values that make sense are a list of constants), you will only see the constants that make sense, rather than all constants.

Tooltips

The method editor displays a tooltip when you position the pointer over a property of a method name in the listing of the method. This shows you the property description, or the method interface and description. The tooltip for a constant also shows you the constant description.



```
Calculate p1.$allowclose as
Do p1.$segment
d For
 method
```

$allowclose

If true, the datafile can be closed

Figure 116:

Initial Values

You can use the Code Assistant in the initial value column of the variable pane of the method editor.

Expanded Entry

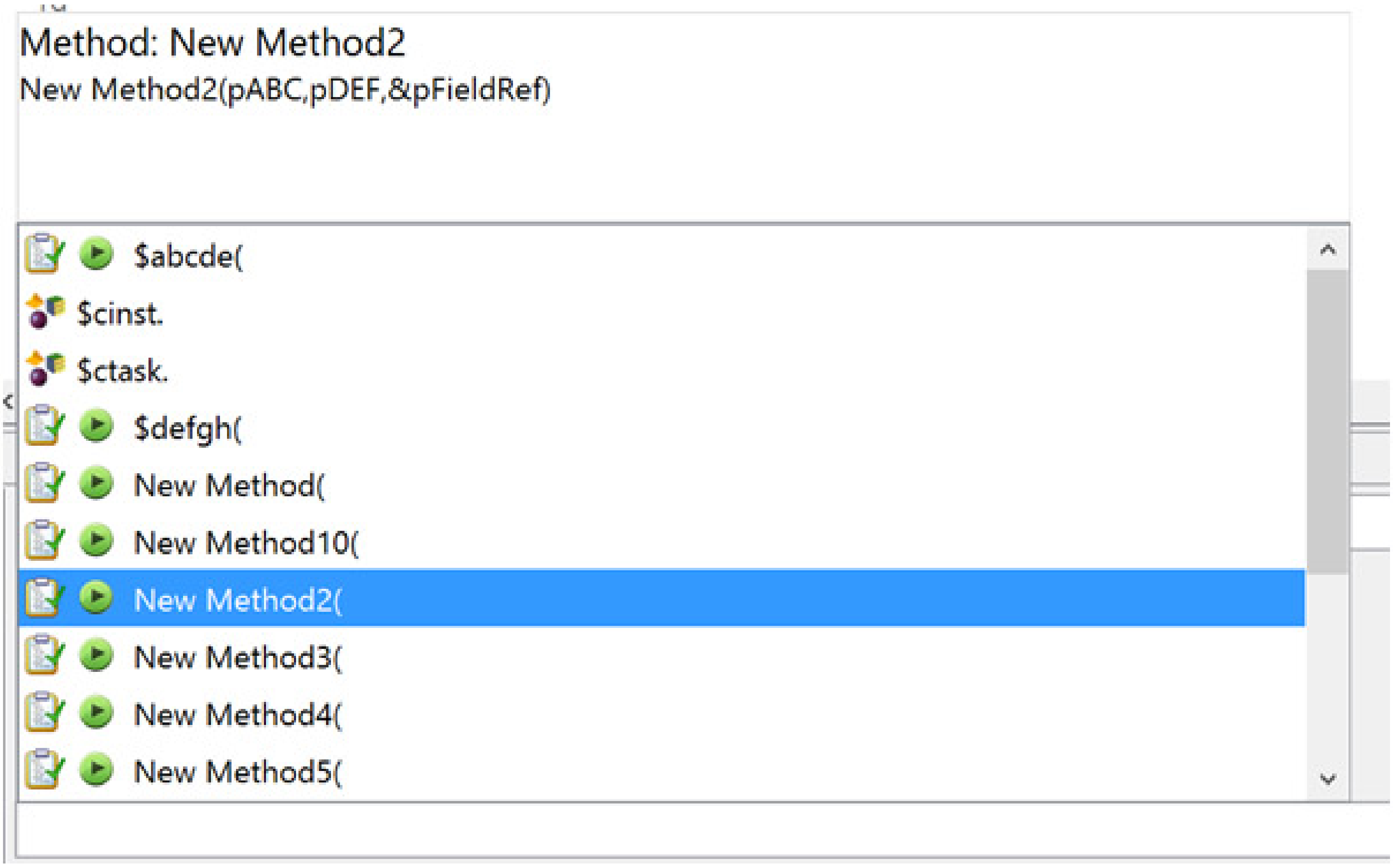
The Code Assistant is available in the expanded entry box in the method editor – it opens as an overlay over the method editor command palette. You can close it by clicking on another window, pressing return (or pressing escape to discard changes).

Replacing Data

When you select some notation from the Code Assistant popup, it replaces the entire word (if any) in which the caret is located.

Method Commands

The commands *Do method, Do async method, Do code method, Load error handler, Unload error handler, Set 'About' method, Set timer method, Start server, Install menu, Install toolbar, Open window,* and *Set report name* use a Code Assistant popup to select their method or class.



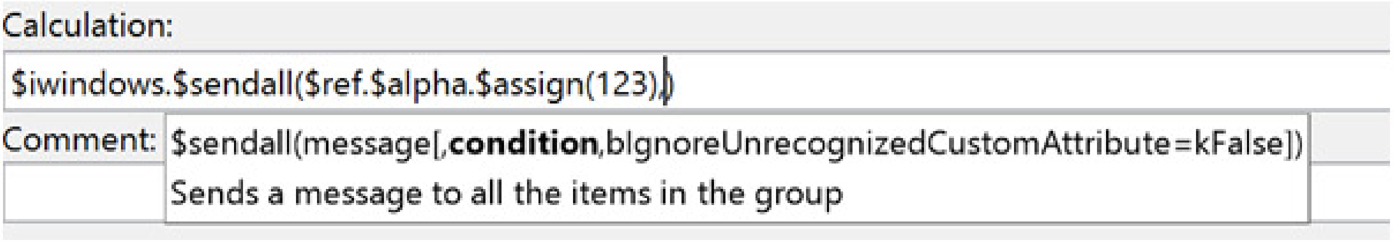
Method: New Method2

New Method2(pABC,pDEF,&pFieldRef)

Figure 117:

Parameter Highlighting

When you position the caret somewhere in the parameters of a function or method that the Code Assistant recognizes, or in a method command that has parenthesized parameters e.g. SMTPSend, Omnis displays a popup (in the opposite direction to the Code Assistant popup) that displays the method parameters and the method description. In addition, the parameter in which the caret is currently located is highlighted in bold.



$iwindows.$sendall($ref.$alpha.$assign(123),)

$sendall(message[,**condition**,bIgnoreUnrecognizedCustomAttribute=kFalse])
Sends a message to all the items in the group

Figure 118:

You can press Escape to temporarily dismiss this popup (unless “parameterHelpEnabled” in config.json is set to false).

Parenthesis Matching

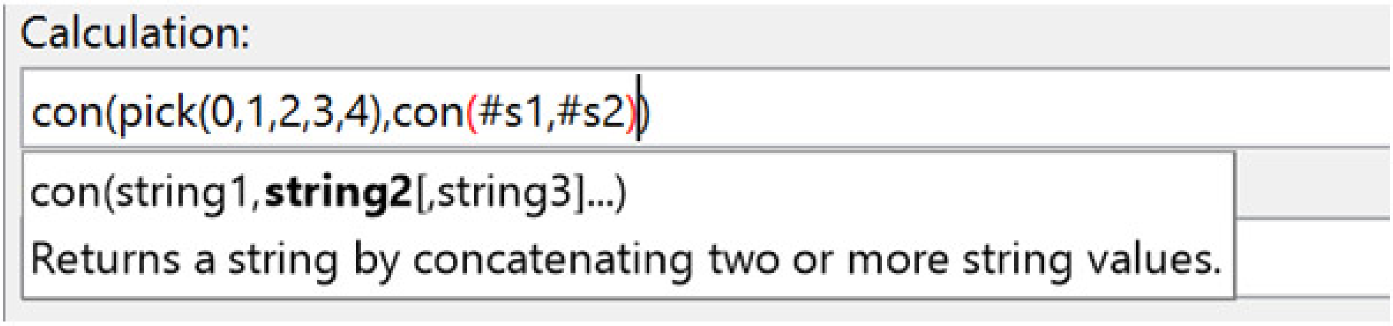
When you position the caret immediately after an open or close parenthesis in an expression, or immediately after an open or close square bracket, Omnis draws the matching parentheses or brackets using a different color.

There are two properties which control this, in the method editor chroma coding options: $bracketbackcolor and $brackettextcolor. To disable this, you can set both of these options to kColorDefault.

Overloads

Certain methods are overloaded. In simple cases, the Code Assistant shows this by using a vertical bar to separate different possibilities e.g.

```
$remove(rLine|iLineNumber|kListDeleteSelected|kListKeepSelected)
```



```
con(pick(0,1,2,3,4),con(#s1,#s2))
```

con(string1,**string2**[,string3]...)

Returns a string by concatenating two or more string values.

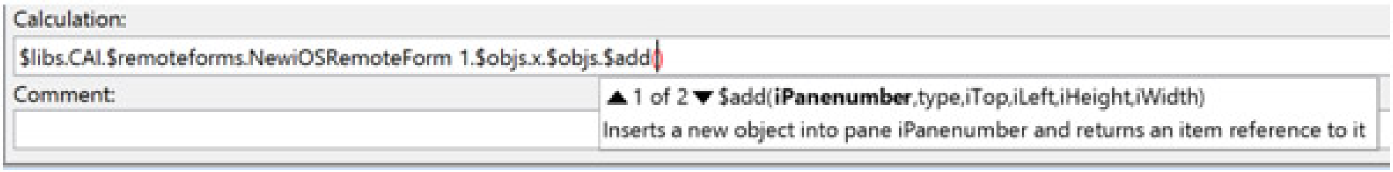
Figure 119:

However, there are other cases where this is not possible, for example:

```
$createobject for a JavaObjs\System\java\lang\String object has 15 overloads
```

$add for an unknown window object could be adding a complex grid or paged pane or scroll box, and the object being added may or may not be an external component.

In these cases the description shown for the method shows all overloads, and the parameter highlighting popup has arrow icons, indicating that you can use the up and down arrow keys to select the overload you are using, thereby resulting in sensible parameter highlighting. Omnis does not attempt to figure out the matching overload by analysing the parameters.



```
$libs.CAI.$remoteforms.NewiOSRemoteForm 1.$objs.x.$objs.$add()
```

▲ 1 of 2 ▼ $add(**iPanenumber**,type,iTop,iLeft,iHeight,iWidth)

Inserts a new object into pane iPanenumber and returns an item reference to it

Figure 120:

Custom Properties

The Code Assistant recognises custom properties, i.e. properties of an instance or an instance object, implemented using two methods, $propname and $propname.$assign. The Code Assistant combines these into a single property in the list of completions rather than showing the two methods, and provided that the Code Assistant can resolve the parent notation, it will also show $assign and $canassign as possible completions for notation relative to a custom property.

Omnis Help

While using the new Code Editor you can open the inline Omnis Help system by pressing **F1** or using the **Omnis Help Topics** option in the main **Help** menu. The Omnis Help provides comprehensive help for all commands, functions, notation, and so on: the content in the Omnis Help is the same as that provided in the Code Assistant.

What is displayed in the Omnis Help is context sensitive and will depend on what is currently selected in the Code Editor, as follows:

- If no text is selected in the Code Editor, it tries to obtain the text from the syntax item containing the caret - if there is nothing useful, no help will be displayed, otherwise it will pass the text for the syntax item to the help system, e.g. 'Calculate' for a Calculate command when the caret is in the command name.
- If some text is selected, and all selected text is on a single line, the editor passes the selected text to the Help system. If the selected text spans lines, no help will be displayed.

After performing 1 or 2, the Help system opens. If the text passed to the Help system uniquely identifies a single help page, that help page is displayed. Otherwise, the help window opens at the search tab, searching for the text passed to the Help system.

Debugging Methods

You can open most class and field methods and run them from the debugger menu bar or toolbar. Note that event handling methods will not run from the *On* command without the event, but you can try out most types of methods while you are in design mode. You cannot execute methods that contain instance or task variables at design time since these variables are only available when the objects are instantiated.

To run or execute a method

- Select Debug>>Go from the debugger menu bar

or

- Click on the Go button on the debugger toolbar

Execution will begin from the selected line. When you first open the method editor the first line of the first method is selected. You can halt execution by pressing the stop key combination Ctrl-Break/Cmnd-period/Ctrl-C. When you break into a method the debugger completes the current command and halts execution.

Along with the Execute and Test options, the basic debugging operations on the Debug menu are:

- **Go** (F5) executes from the Go point
- **Step** (Ctrl+Shift+S) executes from the Go point to the next method line, stepping into recipient methods
- **Step Over** (Ctrl+Shift+V) runs from the Go point to the next method line, executing method calls, but not stepping into them
- **Trace** (Ctrl+Shift+E) steps automatically through the method
- **Set Go Point** (Shift+F2) sets the current method line as the Go point
- **Go Point Not Set** indicates the method with the Go point when one is set
- **From Line** and **To Line** runs, steps or traces from the current line or to the current line

You can Alt-click in the left margin of the Code Editor to execute the “To line” command provided that code is executing.

The Go Point

A method normally runs from the start, but you can start execution from any method line by setting it as the Go Point.

To set the Go point

- Select the method line and choose the Debug>> Go menu option

or

- Select the method line and click the Set Go Point button on the toolbar

The debugger highlights this line and puts a yellow arrow in the left margin pointing to the method line where execution will begin. You can move around the program, changing the code, without changing the go point, which is independent of the current line. The name of the method containing the Go point is shown in the Debug menu and choosing this option from anywhere returns you to the Go point. You can clear the Go point using Stack>>Clear Method Stack.

Execution Errors

When an error occurs in a running method, Omnis takes you into the debugger. The offending method is displayed with the go point at the method line that encountered the error, and an error message is shown in the status area. Error messages include the error number and text, for example “E108139: Set main file command with no valid file name.” You can use the various inspection tools to find out why the error occurred, fix it, and continue.

You can use the Debug>>From Line submenu to run the method from the currently selected line rather than the go point. The submenu items let you Go, Step, Step Over, or Trace from the current line instead of from the go point. The To Line submenu lets you Go or Trace from the go point to the current line, which becomes a one-time breakpoint.

Stepping through a Method

Normally when debugging you will want to step through the code rather than just run it. This gives much more control over when to start and stop methods and lets you examine fields, variables, and the method stack at specific points in the program. You use stepping in conjunction with breakpoints to control the debugging of your code.

To step through a method

- Choose Debug>>Step from the debugger menubar, or click on the Step In button

Every time you click on the Step In button, Omnis executes the line at the go point and sets the go point to the next line. If a command at go point calls another method, the debugger loads the recipient method on the method stack and sets the go point to the first line in that method.

The Step In option steps into a recipient method. You can avoid this with Step Over where the debugger executes the recipient method without stepping into it. This speeds up debugging if you have a lot of method calls.

Tracing a Method

As well as stepping through your code, you can record or trace method execution.

To trace a method

- Choose Debug>>Trace from the debugger menubar, or click on the Trace button

The debugger steps through your code automatically, including stepping into recipient methods, and adds each method line and its parameters to a *trace log*. You can open the Trace Log from the Tools menu, or by clicking on the Trace Log node in the Studio Browser.

The first column in the trace log shows the name of the currently executing method, and the class it belongs to. The second column shows the method line and parameters of the currently executing command. When you double-click on a line in the trace log, the debugger goes to and highlights that method line.

You can open the trace log from within a method using the *Open trace log* command. For example, you can place the *Open trace log* command in the startup task of your library to trace method execution immediately after your library opens.

You can specify the maximum number of lines in the log in the Max lines entry field (this is not available in the Studio Browser view of the trace log); the maximum is 100000. When the log contains the specified max limit, it discards the earliest lines when new ones are added.

Copying code from the Trace Log

You can copy selected lines from the Trace log to the clipboard using the Edit menu **Copy** command or Ctrl/Cmnd-C shortcut key.

Showing the trace log in the Studio Browser

There is a node in the Studio browser which opens an alternative view of the Trace Log; the current number of lines in the log is shown. There is an option in the HUB options to select whether or not the Trace Log node is displayed in the Studio Browser: the default is on. You can use the search box at the top of the Studio Browser to search the contents of the trace log.

Contents of the trace log

The sys(193) function returns the contents of the trace log. It works in both the development and runtime version of Omnis.

Server socket bind failures

The “runtimeOpensTraceLogOnSocketBindError” option in the ‘server’ section of config.json controls whether or not the Trace Log opens in the runtime when a server socket bind fails: the default value is true, so set this to false to suppress the trace log. In the developer version, the Trace Log window never opens when this occurs as you can view the trace log in the browser.

Private Methods

When you step or trace through the methods in your library the debugger will normally enter each method that is called, even if a method is in a private library. However if you set the library property $nodebug to true, the debugger will not display methods contained in private libraries. You need to set this property each time you open the library.

Method History

Omnis stores a list of visited methods which allows you to quickly move back to a recently visited method. The toolbar in the Method Editor contains a **Back** and **Forward** button allowing you to traverse the history of visited methods. Note that inherited methods and the object nodes in the method editor do not form part of the history which can hold up to 256 items. You can also use F10/Shift-F10 to move back and forward respectively. In addition, a long press on either of the buttons opens a menu which shows the available history items in the direction of the button, up to a limit of 20 menu items.

Omnis removes affected entries from the history when a library is closed, a class or method is deleted, or when various other actions occur that would affect an entry in the history list, such as when fields are renumbered.

Inspecting Variable Values

You can inspect the value of a variable or field using the Variable menu (or Variable context menu). You can display the Variable menu for any variable or field by *Right-clicking on its name* in the method editor or the Catalog.

Variable Menu

The **Variable** context menu gives you full information about the variable or field (or the class it belongs to, if any).

To display the Variable Menu

- Right-click on the variable or field

You cannot inspect the value of instance or task variables in design mode (in a class editor) since the variables do not exist: in this case, the Variable context menu is grayed out. To examine an instance variable in a remote form, set a breakpoint in the $construct method (or somewhere else in your code), then test the form, and switch back to Omnis when the form opens in your browser – at this point you can right-click on an instance variable to examine its values etc.

The **Variable** context menu contains the variable name, its value, parent group and data type, and a series of debugging options you can apply to the variable; you can also Copy the value for a simple variable or Export a list or row variable from this menu, as well as Insert New and Delete Variable. The other options at the bottom of the context menu are discussed under Breakpoints.

Variable Value window

The first option **Variable** opens the Variable Value window, except that for Item References with a value, it opens the Notation Inspector.

This window shows the variable name and type at the bottom and displays the value, which you can edit. Omnis updates the value in the window whenever you bring the window containing the method to the top, but you cannot observe the value change dynamically through this window. Note you cannot edit a binary variable.

On the Variable Value window’s View menu

- Redraw Values redraws the variable on any window
- Single Window Mode shows subsequent variable values in the same window

The value window for a variable is valid only for as long as the variable is current.

You can rename a variable in your code directly (rather than having to go to the Variable pane) using the Variable context menu; the option applies to class, instance, local and parameter variables.

List Variables

You can show and edit a **List** variable in a value window using the **Variable** context menu option. There is a **Search** panel in the List variable window to allow you to search the contents of a large list variable while debugging. To search a list variable, click into the List variable window, press Return, and then press the Search button, or select a previously saved search in the droplist. There is also a button to navigate to the current line, which sounds the bell if pressed when there is no current line.

Search results appear in a separate panel that allows you to quickly navigate to results. The Line number and Column name for each search result is shown. If the Column name is empty, the Column number is shown.

Exporting List or Row Variables

The “Export Tab Separated...” option in the Variable menu (in the same location as the “Copy Value” option that appears for various simple data types) allows you to export the values from a List or Row variable. When selected, it prompts for the path name of a file that receives a tab-separated value representation of the list or row.

The output file is UTF-8 with a UTF-8 byte-order-marker. The first export row comprises tab-separated column names. Simple types in the list are exported as their actual value, whereas types such as lists are output as an information string, e.g. (5 Lines). If the characters tab, carriage return, linefeed or backslash occur in the data, they are exported as escapes: \t, \r, \n and \\ respectively. If a column has a #NULL value, it is exported as the text NULL.

Jump to Variable Definition

You can jump to a variable definition in the Variables pane in the method editor using the **Variable** context menu; this is useful if a method contains many variables and saves you visually scanning the variable list to find the variable.

To show the definition for a variable, Right-click on the variable name in your code and select the **Variable Definition** option. The focus will jump to the appropriate tab in the Variables panel, highlighting the variable. Alternatively, you can select the **<Variable-Type> variables** option to pop up a separate Variables panel highlighting the variable.

Variable and Event Tips

You can hover the pointer over a *variable name* anywhere in the Code Editor to display a Variable tip, showing the variable name and its value, if available.

Precedence is given to variables over functions when generating variable tips in the Code Editor, for example, when a variable name is the same as a function name (although this is generally not recommended).

You can also hover the pointer over an *event name,* e.g. evOpenContextMenu, to display its definition including all its parameters and their descriptions.

Variable Panel

The **Variable panel** allows you to view and modify variables while debugging or stepping through your code. It is only populated when execution pauses, as with a breakpoint. After you resume execution, it remains populated (but disabled) for a short time, until either execution pauses again (meaning it updates) or execution does not pause soon enough (meaning it clears). This means that the Variable panel does not flicker while stepping through code.

When execution pauses, the focus moves to the Variable panel. For example, while stepping through code the Variable panel will show $cinst, the task and instance variable values, and the values of any watched variables: see the Variable panel highlighted in red below.

Viewing Variable Data

The Variable panel displays a hierarchy of controls that allow you to drill down into the data. Each time the debugger pauses execution, it refreshes each level of the hierarchy until it reaches a level which is no longer valid, e.g. you might drill down into a local list variable, and execution pauses in a different method, so the local list is no longer valid, so in this case the panel will display the local variables of the new method.

In many cases, the panel displays variables in a grid using either the row or list representation of the grid as appropriate. The grid display for a variable or list cell shows a text representation of the value. This may be either its value, or it may be some other representation, e.g. the number of lines in a list, or an object instance name. The grid is read-only, allowing you to use the arrow keys or tab/shift-tab to move around the grid.

As you move around the grid, the current cell is highlighted, and the data type of the current cell is displayed in the status bar below the grid.

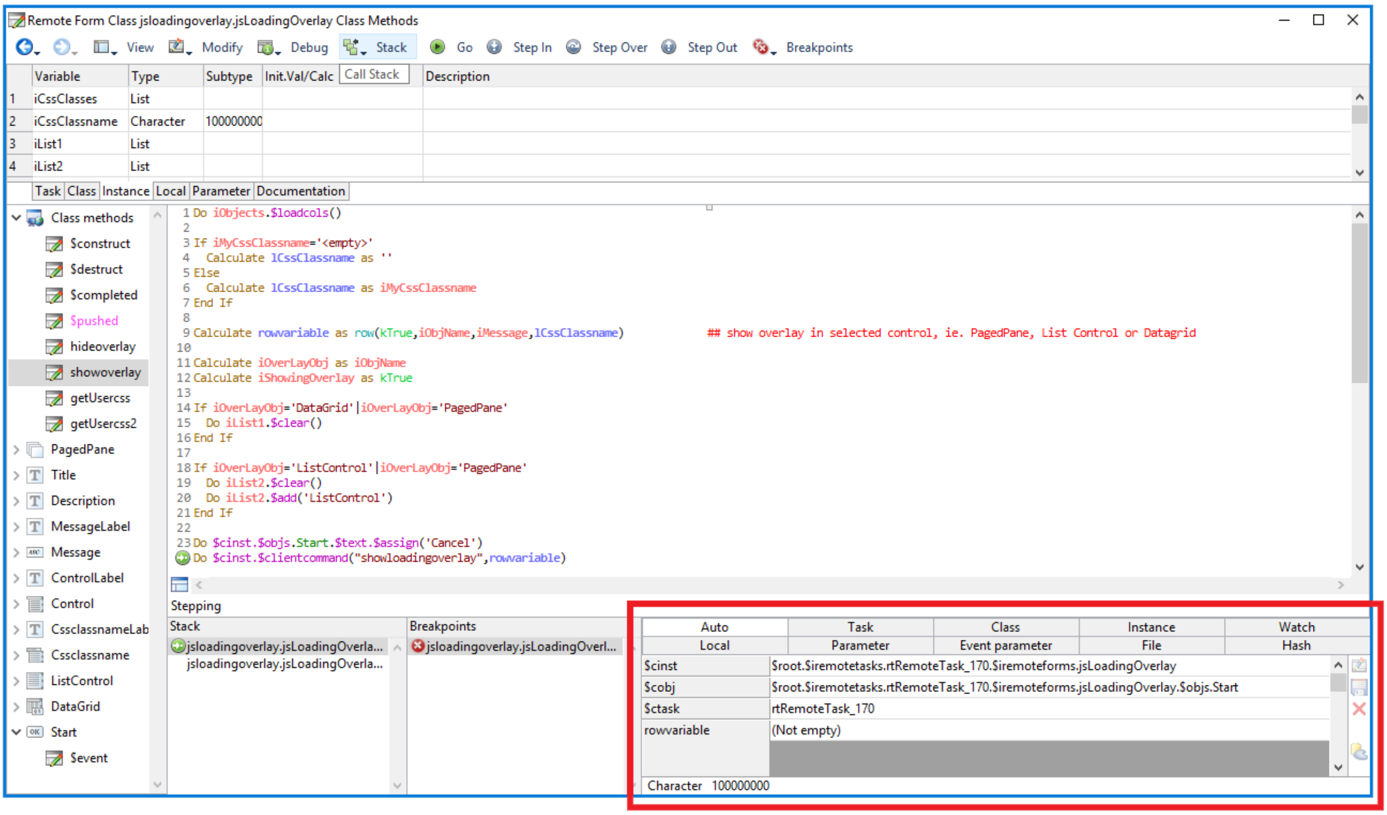
Sometimes a cell represents data such as a list or an object – in this case, you can drill down to view the contents of the cell by either clicking on the cell, or by pressing the Return key. After drilling down, a back button appears in the area above the grid, that you can use to navigate to the previous level, or alternatively you can press Backspace.

You can Ctrl/Cmd+click on a cell that would normally drill down, in order to give that cell the focus.

Buttons to the right of the grid enable, disable or check depending on what you can do with the current cell.

When enabled, you can click on the Modify button, or press the Return key, to edit the variable value. While in edit mode, the remainder of the window disables, apart from Cancel and Save buttons. You can use the Escape key to cancel, and the Return key to save the value (i.e. the key specified as saveModifiedVariable in keys.json): note that the Return key does not allow you to save the variable if it makes sense to add returns to the data being edited.

There is also a button to toggle the current value between NULL and empty.



	Variable	Type	Subtype	Init.Val/Calc	Description
1	iCssClasses	List			
2	iCssClassname	Character	100000000		
3	iList1	List			
4	iList2	List			

```
Do iObjects.$loadcols()

If iMyCssClassname='<empty>'
  Calculate lCssClassname as ''
Else
  Calculate lCssClassname as iMyCssClassname
End If

Calculate rowvariable as row(kTrue,iObjName,iMessage,lCssClassname)          ## show overlay in selected control, ie. PagedPane, List Control or Datagrid

Calculate iOverLayObj as iObjName
Calculate iShowingOverlay as kTrue

If iOverLayObj='DataGrid'|iOverLayObj='PagedPane'
  Do iList1.$clear()
End If

If iOverLayObj='ListControl'|iOverLayObj='PagedPane'
  Do iList2.$clear()
  Do iList2.$add('ListControl')
End If

Do $cinst.$objs.Start.$text.$assign('Cancel')
Do $cinst.$clientcommand("showloadingoverlay",rowvariable)
```

$cinst	$root.$iremotetasks.rtRemoteTask_170.$iremoteforms.jsLoadingOverlay
$cobj	$root.$iremotetasks.rtRemoteTask_170.$iremoteforms.jsLoadingOverlay.$objs.Start
$ctask	rtRemoteTask_170
rowvariable	(Not empty)

Character 100000000

Figure 121:

Top Level Variable panel

When you first pause execution, the debug window displays the top-level Variable panel. This allows you to view Auto, Task, Class, Instance, Local, Parameter, Event Parameter, File and Hash variables. Auto comprises variables identified from the line before the current line (if any), the current line, and up to 2 lines after the current line.

The top of the top-level Variable panel allows you to select the currently displayed scope:

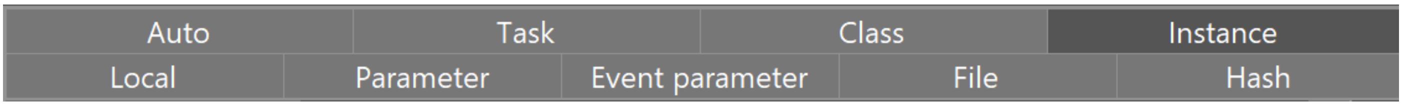


Figure 122:

You can either click on a button (heading) or type its first letter when the Variable panel has the focus, to display the scope. Save Window Setup will save the current scope.

With the exception of the File scope, each scope displays its variables in a grid. The file scope initially displays a list of file classes. You can then drill down into a file class, in order to view its values.

For task, class and instance variables, the panel shows the values for all levels of the inheritance hierarchy, with the names of inherited variables shown in the inherited color.

Table Instances

There is an entry “Table instance data” at the start of the Auto tab when debugging code in a table instance. Simple references like $cinst.name will show in the Auto tab, when name is not a variable in the normal variable scopes, e.g. a column in a row in a table instance.

Object Variable panel

When you drill down into an object or object reference, the panel displays properties and/or variables. The top of the panel looks like the following:

In the case of a non-visual object, all the buttons at the right are hidden, and the panel just shows properties. In the case of a sub-classed non-visual object, all buttons are present and enabled. In the case of an object that is not sub-classed from a non-visual object, the properties button is disabled.

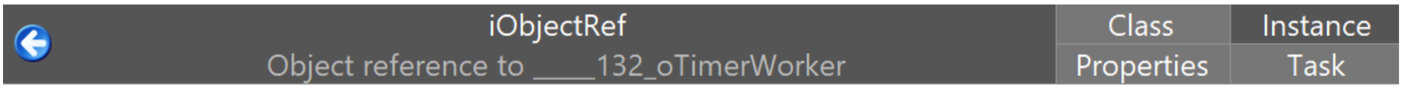


Figure 123:

As for the top-level panel, you can either click on a button, or type its first letter when the Variable panel has the focus, to display the scope.

List or Row Variable panel

For a row, this is a straightforward grid. When you drill down into a list, the panel initially displays the first 64 lines (or $linecount if less than 64) of the list. **Next** and **Previous** buttons at the top-right of the panel allow you to read more lines:

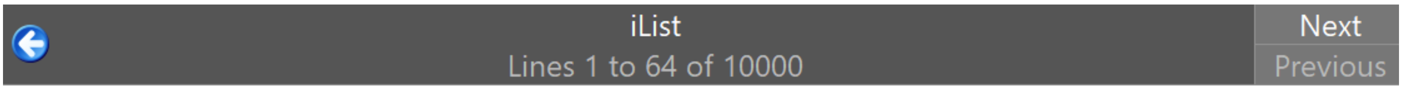


Figure 124:

If you hold the Shift key while pressing the button, the panel reads all data in the direction specified, in chunks until there is no more. While doing this, it may display a working message (if it takes long enough), which you can use to stop any further data being read.

Each time you step, and the variable remains in scope, the panel initially updates with the chunk of data from the start of the current scroll position.

You can modify the current line and selection of the list using the buttons on the Variable panel. These prompt for the new current line, or changes you want to make to the selection.

Item Reference Panel

When you drill down into an item reference that has properties (rather than an item which is a reference to a variable), the panel displays the property values of the item. You can use this panel to modify values for which $canassign is kTrue, provided that they are of a suitable data type for editing.

Integer

Integer variable values are displayed directly in the table. You can click on the H icon to display integers in Hex format. In either display mode (decimal or hex), when modifying an integer, you can either enter a decimal value or a hex value.

Large Character

Character variables containing more than 128 characters are displayed as their length followed by a preview of the start of the data. You can drill down into the variable, displaying a character Variable panel. When you first drill down, this displays up to the first 64k characters. **Next** and **Previous** buttons at the top-right of the panel allow to to read more chunks:

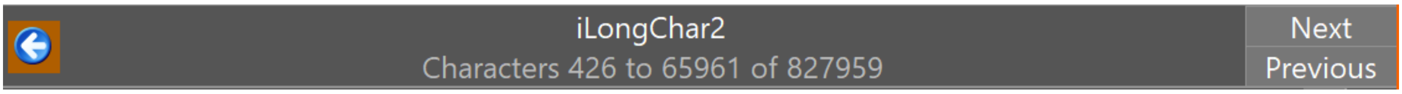


Figure 125:

If you hold the Shift key while pressing the button, the panel reads all data in the direction specified, in chunks until there is no more. While doing this, it may display a working message (if it takes long enough), which you can use to stop any further data being read.

Each time you step, and the variable remains in scope, the panel initially updates with the chunk of data from the start of the current scroll position.

If you edit the data, the edit applies to the entire variable value, i.e. the new value comprises any data on the server before the loaded data, followed by the edited loaded data, followed by any data on the server after the loaded data.

Binary

To view and edit a binary variable, you always need to drill down. You are then presented with a hex binary editor grid. When you modify the variable, a button on the right provides various binary editing operations. The binary panel works in a similar way to the character panel, with next and previous buttons.

Picture

You can drill down into a picture variable and view, edit, or save it using the Save picture button (folder icon). The new button is available when viewing image data, in the modify tool strip to the right of the image.

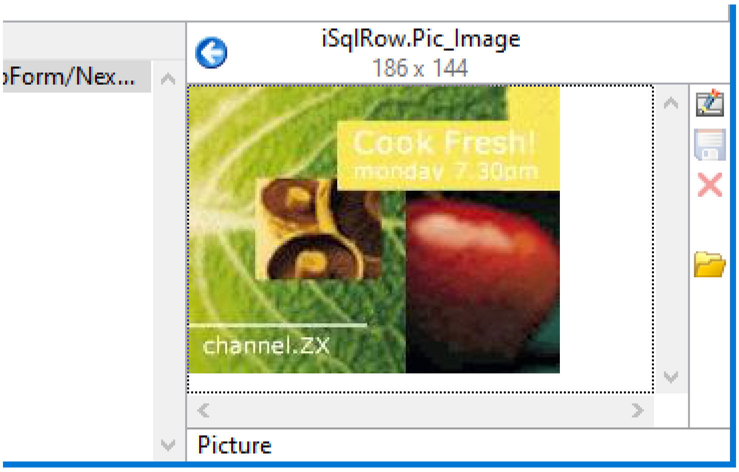


Figure 126:

The Save picture button is enabled when not modifying the variable value, and when the debugger recognizes a JPEG, GIF or PNG (the latter includes shared pictures stored as PNG, in which case the saved image is a PNG without the shared picture header). The button uses the binaryEditOperations keyboard shortcut.

Boolean

Boolean variable values can be Empty, False or True. These can be set using the variable grid drop list.

Omnis does not treat Empty and False as two different values of Boolean variables, when displaying them in the debugger. Therefore, the debugger Variable panel, variable tooltips, variable context menu and variable window all display and treat Empty as False or NO as appropriate.

The Values List

In addition to the Value window for an individual variable you can show a Values List for whole groups of variables such as task variables, class variables, and local variables.

To show the Values List for class variables

- Right-click on a class variable name in a method or in the catalog
- Select the Class Variables option from the variable context menu

The Values List for class variables appears with the different variable types on tabbed panes.

On the View menu for the window

- Redraw Value redraws the variable wherever it occurs on a window
- Show Full Value opens a scrolling Value window below for the selected variable

The Variable popup menu for a file or schema class lets you modify the class, and for file classes only the Values List shows the current values for the file class.

Sorting Variables

There is a **Sort Names** command on the **View** menu of the variables list window available in the Method Editor when inspecting variables. The sort order is always set to an ascending sort, and is not case sensitive. This item is toggled on or off when selected (the state is saved with the window setup).

Displaying Control Characters

You can display control characters in data or content when inspecting a variable in the Field Value window and Values list window, displayed when you Right-click on a variable: these tools have a menu that allows you to:

- Show characters normally
- Show all control characters (in this case no line breaking occurs on carriage return for multiline entry fields)
- Show all control characters except carriage return (in this case carriage returns break lines as usual)

The menu also allows you to increase and decrease the font size used for all content except the binary data.

The control characters are displayed using the Unicode page 0x2400 which has visual representations of control characters. In addition to characters 0-0x1f, 0x7f (del) is also treated as a control character.

In addition, the Catalog status bar, Variable value tooltips and Variable context menus always show control characters if present.

The Save Window Setup option for the Values list saves grid column widths, and the height of the value when using show full value. Save Window Setup for both the Field value window and the Values list window saves the current font size and the option controlling how or if control characters are visible.

Watching Variable Values

You can monitor or watch the value of a variable by making it a *Watched* variable. You can add task, class, local, instance and parameter variables to the Watch variables pane in the method editor. When you run the debugger you can see the value of a watched variable change in the Watch tab in the Variable panel (bottom right of the Code Editor).

To set a watch variable

- Right-click on the variable name and choose Watch Variable

or

- In the method editor, drag the variable from the variable pane into the watch pane

The Watch Variable item on the context menu is now checked. You can enlarge the watch pane by dragging its borders. The watch variable value is only updated when stepping, unless the method redraws it.

To remove a watch variable

- Right-click on the variable name and uncheck Watch Variable on the popup menu

Breakpoints

A *breakpoint* is a marker on a method line. When the debugger reaches that line, it stops execution and makes that line the go point. When a breakpoint is encountered and you have switched to a different application (such as a browser in the case of debugging JavaScript Client methods) the Omnis entry in the Windows Task bar will flash and you have to click the button to return to Omnis to continue debugging.

To set a breakpoint

- Click in the margin of the method line

or

- Click in the method line and click on the Breakpoint button

When you set a *breakpoint* for a line, a red dot appears in the left margin. A *one-time breakpoint* is a breakpoint that the debugger removes immediately after you break on it. It is marked by a blue dot in the margin.

When you close a library, you lose all breakpoints in the methods in that library. You can use the *Breakpoint* command in a method to set permanent breakpoints, but you must be careful using permanent breakpoints since you may forget to remove them and they may be encountered in your application when it is deployed to end users!

The Breakpoint menu lets you create and clear breakpoints.

- **Breakpoint** (Ctrl/Cmnd-Shift-B)
  sets a full breakpoint at the current line

- **One-time Breakpoint** (Ctrl/Cmnd-Shift-O)
  sets a one-time breakpoint at the current line

- **Clear Breakpoints** (Ctrl/Cmnd-Shift-C)
  clears all the breakpoints

- **Clear Field Breakpoints** (Ctrl/Cmnd-Shift-F)
  clears all the field change breakpoints, calculation breakpoints, and min and max settings (see below)

The rest of the **Breakpoint** menu is a list of all the current breakpoints. Choosing a breakpoint from this menu displays the line in the debugger.

You can also set breakpoints and from line to line execution, by right-clicking in the left margin of the method line or by using the appropriate tools on the toolbar.

Breaking on Variable Change

In the second half of the variable context menu, there is a group of breakpoint options that let you set breakpoints based on variable or field values.

The debugger only tests for variable or field breakpoints when methods are running, so a variable change during an enter data suspension of a method will be immediately reported if there is a control method and delayed otherwise. If there are several variable breaks at the same command, the debugger only displays one. Setting a variable value breakpoint slows down method execution considerably.

The menu option **Break on Variable Change** tells the debugger to stop the method when the variable value changes. The debugger puts a check mark against the line. Reselecting the same line toggles the break off. The status line displays the text 'Break on variable change (field)' when the break occurs.

The **One-Time Breakpoint** option puts a single-stop variable change breakpoint on the line.

Breaking on Calculation

You can also create a variable value breakpoint with a calculation. For example, to stop a method when a local variable lvLines becomes equal to the number of lines in list cvList, the calculation is entered as

```
lvLines = lst(cvList,cvList.$linecount)
```

The menu option **Break On Calculation** sets the breakpoint, and the following line **Set Calculation** prompts for the calculation. The debugger treats the calculation value as a boolean value where zero corresponds to No and anything else corresponds to Yes. Execution breaks when the calculation evaluates to Yes, but with a qualification: the break happens only when the calculation changes from No to Yes. This means that if the calculation is always Yes, the break never happens: it also means that the break happens only when the change is from No to Yes, not every time the calculation evaluates to Yes.

For example, the calculation break

```
(lvNumber<10) | (lvNumber>20)
```

ensures that local variable lvNumber stays within the range 10-20.

Each variable or field can have one calculation breakpoint. There is no requirement that the calculation refers to the variable.

The **Store Min And Max** option adds the minimum and maximum variable values to the end of the menu as execution proceeds, along with the item **Clear Min and Max** that lets you turn off the feature. If you choose either menu item, Omnis writes a line to the trace log. Turning on **Store Min And Max** slows down the debugger a good deal.

The Method Stack

A stack is a list of things that you can access in a last-in, first-out manner. When you call a method, Omnis pushes the current method onto the method stack of executing methods. The debugger adds each new method to the **Stack** menu in the method editor. The top-most menu item is the latest method, the one below it called it, and so on. When a method returns, Omnis removes the top item, also known as popping the stack, and goes to the calling method. You can examine any method on the stack by selecting it. You can also move up and down the stack with the **Stack** menu items **Move Up Stack** and **Move Down Stack**.

If you select a method in a different class while holding down the **Shift** key, the debugger opens a new method design window.

When you stop in a method with a breakpoint, an error, a step, or an explicit stop, Omnis sets the go point to the next method line and saves the stack. It marks the commands in the methods on the stack that will execute when you return to that method with a gray arrow in the left margin pointing to the method line where execution will resume.

A method can appear more than once in the method stack with a completely different set of local variables.

Debug>>To Return runs or traces the method from the go point or current line until it returns control to the method that called it. If the only method on the stack is the current method, this option is grayed out.

There are times when you may want to throw away the current stack and start over. For example, if you follow a problem to its conclusion and everything freezes up, you can restart by clearing the stack. You do this with **Stack>>Clear Method Stack**, which also grays out the **To Return** item and removes the Go point.

Method stack list

The sys(192) function returns a list representing the current method call stack, with a line for each line in the debugger stack menu. The first line in the list corresponds to the call to sys(192), and subsequent lines correspond to how the method running the previous line was called.

The list has 6 columns:

Column	Description
classitem	Item reference to the class containing the method.
object	The name of the object containing the method in the class: empty if it is a class method.
method	The name of the method.
line	The line number in the method, of the method line resulting in the method on the previous line running.
linetext	The text for the method line.
params	The parameters passed to the method. This is a two column list, with the column name and value (the value displayed as a tooltip for the parameter).

The sys(192) function works in both the development and runtime version of Omnis.

The sys(290) function returns the number of methods on the Omnis method stack.

Method stack limit

You can control the number of methods on the Omnis method stack by setting the **stackLimit** item in the “default” section of the Omnis configuration file (config.json); the default value is 30 which is adequate for most applications. Omnis fetches the value of stackLimit on startup, therefore when a library is opened, the stack limit is already in effect.

Debugger Options

The debugger **Options** menu appears with the other debugger menus.

- **Debug Next Event**
  stops at the first line of a method executed for an enter-data event with a control method (a field method, a window control method or a timer method). Note this option is not saved with other debugger options and defaults to off whenever Omnis is started

- **Trace All Methods**
  sets trace mode permanently on.
- **Open Trace Log**
  opens the trace log window or brings it to the top
- **Disable Debugger at Errors**
  stops Omnis from breaking into the debugger on program errors: this is what the end user of your application would see
- **Disable Debugger Method Commands**
  deactivates any debugger commands in the methods
- **Save Debugger Options** saves all the debugger options, and Revert To Saved Options reverts back to the last saved set of debugger options

Debugger Commands

You can control the debugger using the Debugger… commands: see the Code Assistant or the Omnis Help for a complete description of these commands. Note that none of the Debugger commands will work in client-executed methods in the JavaScript Client, however, while developing your app, you can use the *Send to trace log* command in a client method to write a line to the JavaScript console.

The *Breakpoint* command breaks the program when Omnis executes it. If you specify a message, it appears on the status line when the break happens.

Using the Trace Log

The *Trace on* command switches trace mode on, optionally clearing the trace log, and *Trace off* switches trace mode off.

Send to trace log adds a new line to the trace log containing the specified text. The text can contain variable names enclosed in square brackets. You can then use the log as a notepad for your comments, variable or field values, and bookmarks in the code. If you enable the *Diagnostic message* option in the *Send to trace log* command Omnis will send the message to the trace log substituting any variable values where appropriate. To enable diagnostic messages in the trace log, you must right-click on the trace log and select the Log Diagnostic Messages option.

You can double-click on lines in the trace log to open the method editor at the appropriate point in the method.

You can increase and decrease the size of the font in the Trace log using the Ctrl+ and Ctrl- options. The **Save Window Setup** option saves the current font size. (Note the font size of the trace log panel in the browser is not saved.)

Styled Text

The Trace log allows text styles to be added to the logged text. The *Send to trace log* command supports text styles, added using the *style()* function inside square brackets, such as kEscColor and kEscStyle. For example, you could apply colors to sections of the logged text when it is displayed in the trace log panel in the browser or the trace log window; such styles are stripped when writing the trace log line to the text log file in the logs folder.

The trace log renders the text styles if the entry **traceLogUsesStyles** in the 'defaults' section of config.json is set to true (this replaces the ide entry traceLogUsesSyntaxColors in versions before Studio 11).

Note that if you use styles other than kEscColor and kEscStyle, these styles are ignored when copying selected trace log lines to the clipboard as HTML.

For JavaScript client-executed methods, where the *Send to trace log* command sends the text to the JavaScript console (provided it is available), text styles are not supported.

Debugging Variables

The *Variable menu command* applies a variable context menu option to a list of variables. The list has the same format as *Define list*, and for fields can include file names and so on. This command has several options.

- *Set break on field change*
  sets a field change breakpoint for each field in the list
- *Clear break on field change*
  clears any field change breakpoints for each field in the list or all breakpoints if you don't specify a field list

- *Set break on calculation*
  sets a calculated breakpoint for each field in the list: set the calculation for each field with *Set break calculation*

- *Clear break on calculation*
  clears any calculated breakpoints for each field in the list or all calculated breakpoints if you don't specify a list

- *Store min and max*
  stores minimum and maximum values for each field in the list

- *Do not store min and max*
  clears store min and max mode for each field on the list or all modes set if you don't specify a list

- *Add to* watch variables *list*
  adds each specified field to the watch variables pane

- *Remove from* watch variables *list*
  removes each specified variable from the watch variables pane or all variables if you don't specify a list. Variables with breakpoints or with store min and max mode set always appear on the watch variables list

- *Send value to trace log* adds a line to the trace log for each field on the list: if you don't specify a list, adds a line for all fields

- *Send minimum to trace log* adds a line to the trace log with the minimum for each field on the list for which the debugger is storing minimums: if you don't specify a list, adds a line for all such fields

- *Send maximum to trace log* adds a line to the trace log with the maximum for each field on the list for which the debugger is storing maximums: if you don't specify a list, it adds a line for all such fields

- *Send all to trace log* adds a value line to the trace log for each field on the list; also adds minimum and maximum lines to the trace log for each field on the list for which *Store min and max* is set: with no list, adds a line for all appropriate fields

- *Open value window* opens a value window for each field on the list: with no list, opens a window for all fields with whatever limit the operating system puts on the number of window instances

- *Open values list* opens a values list containing the value for each field on the list: with no list, opens a values list for all fields, subject to the operating system limit on the number of window instances. There is one values list for each file class so if more than one field name from a particular file class appears in the list, Omnis displays only one values list for that file class

- *Set break calculation* sets up the calculation for the field breakpoint

Checking Methods

You can check the methods in your library using the *Method Checker*. The method checker is available under the **Tools>>Add-ons** menu on the main Omnis menu bar. It checks your code for syntax errors, unused variables, methods without code, and so on. It provides various levels of checking and reports errors in individual classes or all classes in the current library. Specifically, it is useful for checking libraries converted from an earlier version of Omnis.

Note that the method checker does not correct the code in your libraries automatically, it simply reports any errors and potential problems in your code.

When you open the method checker it loads all libraries that are currently open. Alternatively, you can open a particular library from within the method checker.

To check the methods in your library

- Select the Tools>>Add-ons>>Method Checker menu item from the main Omnis menu bar

- If you need to load a library, click on the Open Library button on the method checker menu bar

- Double-click on the library you want to check

- Shift-click or Ctrl/Cmnd-click to select the classes you want to check, or click on the Select all classes button to select them all

The following checking levels are available

- **Error conditions**
  this level of checking finds problems that can cause runtime errors or undesired behavior: you *must* fix these errors

- **Include Level 1 warnings**
  finds problems that you should investigate because they might result in subtle bugs and strange behavior: you *ought to* fix these problems
- **Include Level 2 warnings**
  finds problems that you should be aware of, including empty methods and/or inefficient code, potential compatibility problems, and platform-dependent code

The different levels of checking are *inclusive*, that is, if you select Level 2 Warnings (the default) this includes Level 1 and the Errors categories.

- Select a checking level, and click on the **Check** button

The method checker works through the classes you selected displaying their statistics in the Method Checker Error Log. You can cancel checking at any time by pressing Ctrl-Break/Cmnd-period/Ctrl-C.

When checking is complete, you can sort the log by clicking on one of the headers in the list. You can print the log or save it to a text file.

You can show a summary of the errors for each class by clicking on the Show Summary button.

Interpreting Errors and Warnings

The following sections detail the different levels of errors and the possible action you should take.

Fatal Errors

These are the type of errors that you *must* fix.

Encountered a construct End without a construct Begin
An ending construct was found without a matching beginning:

- *End if, End switch*, *End while*, *End for*, *Until*, *End reversible block*

Method contains a construct Begin without a construct End
A beginning construct was encountered without the proper ending:

- *If*, *Switch*, *While*, *For*, *Repeat*, *Begin reversible block*

Construct End does not match construct Begin
An ending construct was encountered that did not match the beginning construct, e.g. Begin reversible block followed by an End if.

Encountered a construct element in an invalid context
One of the following was found outside of a proper construct: *Else*, *Case*, *Default*

Encountered a command in an invalid context
One of the following commands was found outside of a proper construct: *Break to end of switch* outside of a Switch construct, or *Break to end of loop* or *Jump to start of loop* outside of a *For, While,* or *Repeat* loop.

Incomplete command
A command with no parameters set, for example, *Set reference* with no reference, *Set current list* with no list name, *Call method* with no method name.

Invalid field reference
An invalid reference to a field or variable (i.e. #???) was encountered: usually a reference to a field or variable that has been deleted.

Invalid method reference
Encountered a command containing a reference to a non-existent method, an unnamed method, or a method in a library that is not currently open. For example, *Call method* with name of non-existent method, *Enable menu line* with reference to non-existent menu or menu line.

Missing extended command or function
A missing extended command or function was encountered, either not loaded or installed: these show in your code beginning with the letter "X".

Bad library name
The library name contains one or more periods.

OK Message or **Sound bell**
Either *OK message* or *Sound bell* command in a server executed method or remote task method; these should be removed or commented out.

Level 1 warnings

These are the type of problems that you *ought to* fix.

Class variable with the same name as a library variable
Could cause precedence problems at the class level.

Optimize method command not in first line of method
The *Optimize method* command should be the first line of a method.

Code in an unnamed method
Named method with no code
Check to see if this code/method is required.

Debugging code?
You should remove all breakpoints before deploying your application. One of the following was encountered: *Breakpoint*, *Trace on/off*, *Field menu command*, *Set break calculation*, *Send to trace log.*

Obsolete command
You should not use obsolete commands: remove them from your code. For example you can replace *Call method* with *Do method* or *Do code method*.

Command removed by converter
In converted libraries certain commands are commented out: you should use another command or use the equivalent notation.

Level 2 warnings

These are the type of problems that you should investigate that might require fixing.

Unused variable
Variable defined but unused, or referenced and not defined.

Unfriendly code: Code which could affect other libraries if running in a multi-library environment
For example, *Clear method stack, Quit all methods, Close all windows, Remove all menus.*

Unfriendly code: Code which would cause the current library to be closed
The following commands will close the current library if the “Do not close other libraries” is not set: *Open library, Create library, Prompt for library.*

Class name specified in an internal method call
Inefficient code.

Code that modifies a library or class
One of the Classes… group of commands, such as *New class, Rename class, Delete class.*

Platform-dependent code
Functions which return different values depending on which platform they are executed, including sys(6), sys(10) to sys(22), sys(103) to sys(114).

Comment containing a question mark
Usually indicates code that needs to be tested, completed, or fixed.

Reference to hash variable
Avoid using hash variables: replace with variable of appropriate scope.

Method Performance

Omnis Studio allows you to collect data about the performance of method execution in your application.

Collecting Performance Data

The Omnis root preference $collectperformancedata enables method performance data collection. Its property is a kCPD… constant specifying whether or not and how Omnis will collect data about method execution performance. The data collected is stored with each method in its class, and can be accessed using the notation. Data is not collected for remote form client methods. The kCPD… constants are:

- **kCPDnone**
  Omnis does not collect method execution performance data
- **kCPDallMethods**
  Omnis collects method execution performance data for all methods

- **kCPDmarkedClasses**
  Omnis collects method execution performance data for methods in classes where the class property $collectperformancedata is kTrue: see below

Classes that can contain methods also have a new property called $collectperformancedata. This property is applied only when $root.$prefs.$collectperformancedata has the value kCPDmarkedClasses. If true, method execution performance data is collected for all methods in the class.

Assuming $root.$prefs.$collectperformancedata allows, the collected data is stored with each method in the class, with the following properties for each method:

- **$callcount**
  The total number of calls to the method. You can only assign zero to $callcount, in which case Omnis also sets $totalexecutiontime, $minexecutiontime and $maxexecutiontime to zero
- **$minexecutiontime**
  The execution time in milliseconds of the shortest call to the method
- **$maxexecutiontime**
  The execution time in milliseconds of the longest call to the method
- **$totalexecutiontime**
  The total execution time in milliseconds of all calls to the method

$minexecutiontime, $maxexecutiontime and $totalexecutiontime are floating point numbers.

Data collection does not update the $…executiontime properties when the method is being called recursively, however it does update $callcount for recursive calls.

Classes that contain methods, and can therefore collect method performance data, have two new methods:

- $clearperformancedata()
  Clears the performance data for all methods in the class
- $makeperformancedatalist()
  Returns a list containing the performance data collected for all methods in the class

There is a very small overhead when data is collected, while there is no impact on performance when performance data is not being collected.

Sequence Logging

Sequence logging allows you to record all method execution in the Development version of Omnis or on the Omnis Server which is used for deploying Omnis web and mobile applications. If your Omnis Server is running multi-threaded mode, sequence logging can log method execution on multiple server threads.

Sys(3000) turns on sequence logging, and sys(3001) turns it off. Sequence logging writes every method command executed to a file in the Omnis directory. The name of the file is reported by an OK message when logging starts. Logging is thread-safe.

The log file can be up to 2mb in size, and when that limit is reached it discards all but the most recent 256kb and continues logging. Logged lines can be of any length. The log file name includes the date. The log file is UTF-8 and has a signature to mark it as such.

Each line in the file is prefixed with “N:” where N identifies the thread. Zero is the main thread. 1-N are Omnis Server threads.

When logging starts, Omnis writes a message to the log file, to allow the developer to identify logging being started and stopped.

Remote Debugger

Remote debugging allows you to debug your Omnis code remotely over the network. You use a development copy of Omnis Studio, the *remote debug client,* to connect over the network to another copy of Omnis Studio, the *remote debug server*. References to “the debugger” in this section refer to the remote debugger, while any references to the local Omnis Studio debugger use the term *local debugger*. Some key points to note:

- The remote debug server runs the code that is to be debugged, and it can be any type of installation: development, fat client runtime, server or headless server.

- Omnis code running in the multi-threaded server, in server stacks other than the main stack, can be debugged.
- The remote debug server and client do not need to be running on the same operating system.
- The version of the client must be the same or later than the version of the server.
- Protected classes and locked libraries can be debugged (any locked classes with $canremotedebugwhenlocked set to False will not appear in the remote debugger).
- From Studio 10.2, you can edit methods and code that you are debugging remotely by opening an 'Edit Session'

Although the term remote debugger is used, the remote debugger client and server can be on the same computer, and in fact the client and server can be the same Omnis process. In the latter case, the remote debug client runs with some restrictions, which are discussed later.

Connectivity

The remote debug client and server always connect to each other over a **WebSocket.** This applies even if the client and server are running in the same Omnis process. The WebSocket connection is a direct connection from client to server, so it may require a port in the firewall to be opened on the server. As WebSocket connections start as HTTP connections, a WebSocket can be a secure TLS connection, and it can require a client certificate to authenticate the client. For the Remote Debugger, a TLS connection is always required, so the WebSocket starts as HTTPS.

An established connection between a remote debug client and server is called *a remote debug session,* or just *session*. A copy of Omnis that is running as a remote debug client or server, or both, can run *only one session* at a time.

Remote Debug Server

In the developer version of Omnis, you can configure the Remote Debug Server by clicking on the Options button at the bottom of the Studio Browser window (next to the revision number) and selecting the **Remote Debug Server** option.

In a runtime version of Omnis (not headless), if the library remotedebug.lbs is in the Startup folder, there is a menu named Remote Debug (see Remote Debug Menu below). This contains a single menu item that can be used to open the Remote Debugger window.

In the headless server you can configure remote debugging via the OS Admin window.

The Remote Debug Server window has two tabs, one to control the server, and the other to configure the server.

The **Control Server** tab has a single button, used to start or stop the server - the button text changes depending on the current state of the server. Until the remote debug server is started, it will not accept a connection from a remote debug client.

The **Configure Server** tab allows you to enter configuration details for the remote debugger server. The Configure Server tab shows fields that correspond to the entries in the configuration file. These fields are described in the following sections.

Remote Debug Menu

The Omnis preference $showremotedebugmenu ($root.$prefs) controls whether or not the Remote Debug menu is displayed. It defaults to kFalse, and is not saved. Therefore, if you want a library to display the Remote Debug menu, you must assign kTrue to this property in your Startup code.

Remote Debug Server Configuration file

The *remote debug server* configuration is stored in the file called remote_debug_server_config.json, located in the folder clientserver/server/remotedebug in the data folder of the Omnis installed tree.

You can edit this JSON file directly as a text file, or use the Remote Debug Server window, as described above.

You should note that Omnis uses a node.js server running alongside Omnis to provide the WebSocket server; this communicates with Omnis using a local in-memory socket. As a consequence, some of the configuration information stored in remote_debug_server_config.json is used by node.js. An example configuration file:

```
{
  "debugPort": 6102,
  "serverPfx": "server.pfx",
  "pfxPassPhrase": "xxxxxx",
  "ca": [ "server_cert.pem" ],
  "requestCert": false,
  "rejectUnauthorized": false,
  "userName": "myUser",
  "hashedPassword": "AAGGoAAAABBSEkknQUIeHQHu1sIyWxlSAAAAIHw9kvCVF4tE//SMpbSGVD/RKJLekoR7TlTvZVy3MbkJ",
  "startRemoteDebugServerAtStartup": true,
  "pauseAtStartupUntilDebuggerClientStartsExecution": false,
  "logConnectionSetup": false
  "excludeFolders": false,
  "inheritedMethodsFirst": false
}
```

The default server port for the remote debugger is 6102.

Debug Port

The TCP/IP port on which the WebSocket server listens for incoming connections from a client.

Server PFX

This is a file containing the server certificate and private key. It must be in the same directory as remote_debug_server_config.json. The default install tree has a self-signed certificate and key generated by the openssl command (available on any system where openssl is installed). You will need to provide your own private key and certificate. You can generate a new private key and self-signed certificate using the following openssl commands:

```
openssl req -x509 -newkey rsa:4096 -keyout server_key.pem -out server_cert.pem -nodes -days 1024 -subj "/CN
openssl pkcs12 -export -out server.pfx -inkey server_key.pem -in server_cert.pem
```

This file is set as the pfx option when calling the node.js method https.createServer(). You can find more documentation about this in the node.js documentation online:

https://nodejs.org/docs/latest-v8.x/api/https.html#https_class_https_server

[*https://nodejs.org/docs/latest-v8.x/api/tls.html#tls_tls_createsecurecontext_options*] (https://nodejs.org/docs/latest-v8.x/api/tls.html#tls_tls

PFX Pass Phrase

This is the pass phrase used to protect the Server PFX file. In the example in the previous section this is xxxxxx.

CA

See *https://nodejs.org/docs/latest-v8.x/api/tls.html#tls_tls_createsecurecontext_options* for more details. You would typically only set the CA when using a self-signed certificate, in which case it has a single entry. In the Server PFX section above, the certificate was signed using server_cert.pem. The general value of this is a comma-separated list of trusted CA certificate file names. The files must all be in the same directory as remote_debug_server_config.json.

Request Client Certificate

A Boolean option. If true, the node.js server requests a client certificate to authenticate the client. The client certificates are discussed later, in the client connectivity section.

Reject Unauthorized

A Boolean option. If true, the server will reject any connection which is not authorized with the list of supplied CAs. This option only has an effect if Request Client Certificate is true.

User Name

If not empty, the WebSocket connection also uses HTTP basic authentication to authenticate the user, in which case this field contains the user name used for HTTP basic authentication.

Hashed Password

If the User Name is not empty, this is the PBKDF2 hash of the password required for HTTP basic authentication.

Start Remote Debug Server

This Boolean option controls whether the remote debug server automatically starts when Omnis starts.

Pause Execution At Startup

If the remote debug server is configured to automatically start when Omnis starts, you can set this Boolean option to true to make Omnis pause execution at startup before it runs the startup tasks of libraries in the startup folder.

When using this option, Omnis displays a working message (Waiting for remote debug client to start execution…), and enters a loop where it waits for a remote debug client to open a session. Once a session is opened, Omnis remains in the loop, where it is now waiting for a command from the client to start execution. During this loop, the client can inspect remotely debuggable code, and set breakpoints for example.

The loop terminates either when the client sends a command to run startup, or when the remote debug session closes, or when a user clicks the cancel button on the working message displayed on the server. When the loop terminates, Omnis runs the startup tasks for the libraries in the startup folder.

You can also Exclude folders from class lists, and Show inherited methods first in the method list in the remote debugger code editor.

Remote Debug Client

The remote debug client is accessible via a new node in the Studio Browser tree, “Remote Debug Client”. It uses a similar session model to the Omnis VCS. When you click on the Remote Debug Client node in the tree, hyperlinks appear in the browser panel for Session Manager, and Open Session.

The session manager allows you to configure remote debug sessions. Each session provides the parameters that allow the remote debug client to establish a WebSocket connection to a remote debug server. These parameters are described in the following sections.

Name

A name that identifies the session.

Server

The IP address or DNS name of the remote debug server.

Debug Port

The debug port configured for the remote debug server. When connecting to the server, the client connects to a URL of the form

```
wss://Server:DebugPort
```

Client Certificate

If the server requires a client certificate, you specify this here.

You can generate a client certificate using the openssl commands:

```
openssl req -newkey rsa:4096 -keyout client_key.pem -out client_csr.pem -nodes -days 1024 -subj "/CN=192.16
openssl x509 -req -in client_csr.pem -CA server_cert.pem -CAkey server_key.pem -out client_cert.pem -set_se
```

Note that this uses the server key and server certificate generated in the example for the Server PFX field of the remote debug server configuration. The client certificate needs to be installed on the client machine.

On Windows, generate a client.pfx file:

```
openssl pkcs12 -export -out client.pfx -inkey client_key.pem -in client_cert.pem
```

Import client.pfx into the windows certificate store: double click on the pfx, add to Personal certificates for the current user.

On macOS, generate a pkcs12 file:

```
openssl pkcs12 -export -out client.p12 -inkey client_key.pem -in client_cert.pem
```

Double click on the file to add it to the keychain.

You can find more details about this in the CURL documentation at:

https://curl.haxx.se/libcurl/c/CURLOPT_SSLCERT.html

Note the Client Certificate parameter is the value passed to the CURL option CURLOPT_SSLCERT.

On Windows, the client certificate parameter is a path expression to a certificate store e.g.

```
CurrentUser\MY\afe2179599460d20da08c12e8c328d84bd300735
```

where afe2179599460d20da08c12e8c328d84bd300735 is the thumbprint viewed by double clicking on certificate in the MMC (MMC certificate snap-in view, details tab, thumbprint field).

On macOS, you can specify either the path of the p12 file, or the keychain name of the client certificate.

User Name

If the server uses HTTP basic authentication, the user name required for that.

Password

If the server uses HTTP basic authentication, the password required for that. Alternatively, you can leave this empty, and the client will prompt for the password when it is required.

Server Connection Logging

You can monitor the client connection to the Remote debugging server, which allows you to highlight any connection problems. You can enable logging in the remote debug client window (or in the config file in the logConnectionSetup item).

If enabled, the Remote Debug Client writes a log file named <session name>.htm to the logs/remotedebug folder, containing a log of what occurred when attempting to connect to the remote debug server. Note that the log is not written until the connection closes.

Preparing Code For Remote Debugging

You have to enable remote debugging in the library, and in the task instance (remote or standard for fat client), by setting the $remotedebug property.

Library

By default, a library cannot be debugged by the remote debug client, meaning that when the remote debug client connects to a server, the library will not appear in the client interface. If you wish code in a library to be debugged with the remote debugger, you need to set a new library property, $clib.$remotedebug: if true, remote debugging of this library is allowed, but it cannot be set to true in an always private library, which means you must set this property to true before making the library always private.

Task

Setting $clib.$remotedebug allows the library and its classes to appear in the remote debug client interface. This allows you to browse the code and set breakpoints. However, only tasks and remote tasks marked for remote debugging will react to these breakpoints. This provides more control over debugging, and specifically in the multi-threaded server, it prevents one breakpoint from stopping every client that hits it.

To mark a task or remote task for remote debugging, set the $remotedebug property of the task instance to kTrue.

In addition, you can set this property of a remote task by adding a query string parameter to the URL used to open a JavaScript client form or execute an ultra-thin request: omnisRemoteDebug=1, for example:

```
http://127.0.0.1:5981/jschtml/jsDragDrop.htm?omnisRemoteDebug=1
```

Locked Classes

You can debug methods in a locked class if the $canremotedebugwhenlocked property in the class is set to true. Locked classes for which this property is kFalse do not appear in the remote debug client list of classes for the library.

Opening a Session

To use the remote debugger client after configuring a session, click on the Remote Debug Client node in the Studio Browser tree, click on either **Open Debug Session** (read-only), or **Open Edit Session** hyperlink, and then click on the hyperlink for the session you want to use. This will cause the client to establish a WebSocket connection to the server. While the connection is being established, progress is displayed in the browser panel, although this is usually very quick. In addition, a Cancel Open Session hyperlink is displayed while the connection is being established.

Edit Session

An **"Edit session"** allows you to edit methods via the remote debugger, that is, you can apply edits, and you can create new variables via the fix error dialog. Note that until you try to save the method back to the server, you will not know for sure if the method will be accepted, since only part of the library is available when editing - you can use instance, class, local, task and parameter variables (from the class or a superclass) or any file class variable in a file class used by the method. Using variables from other file classes, or using notation, functions or commands available on the server (but not the client), will be displayed as an error if you edit a line containing something only available on the server. However, you can still save the method successfully in this case.

In edit mode, methods default to read-only in the remote debug window. You need to explicitly press "edit method" in the toolbar to edit the method, after which you cannot do anything else with the window until you press Save or Cancel or close the window.

Debug (read-only) Session

A **"Debug session" (read-only)** allows you to view and step through code in the remote debugger, but you cannot edit the methods or add variables. The remainder of this section describes remote debugging for a debug session, but an edit session has the same operation except for the above caveats.

Browsing Libraries

After the session opens, libraries marked for remote debugging appear in both the browser tree and the browser panel. The Remote Debug Client child nodes have similar behavior to the Libraries node child nodes, so they include both libraries and folders within the libraries. When you select a child node, the browser panel updates to show the content of that node - this comprises a list of all classes that can contain code.

While any node in the remote debug client sub-tree is selected, a Close Session hyperlink is displayed. In addition, if a single class is selected in the browser panel, a hyperlink named "Open debug window" is displayed. You can click on this (or double click on a class in the panel) to open the remote debug window for the class.

If the server is paused, waiting to run startup, the hyperlinks include a link named "Run Startup" that can be used to tell the server to carry on and run its startup processing.

Save Window Setup for the browser window remembers column positions for the list view of the remote debug client panel.

The status bar of the browser window includes the name of the currently open session.

Excluding Folders

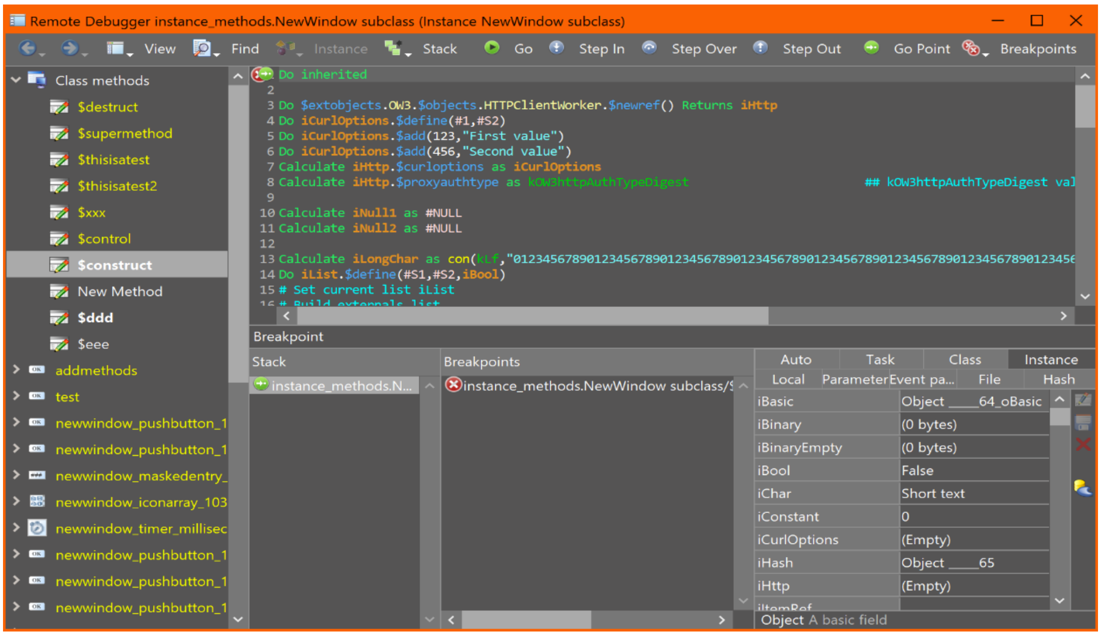
The Remote Debug Server configuration has the 'Exclude folders' option which controls whether or not folder names are returned by the server to the client, and therefore displayed in the browser. The remote debug server dialog allows you to edit this option.

The Remote Debug Window

The remote debug window has a similar layout to the Method Editor. The main difference is that it always shows the debug panel (there is no editor panel). The window shares both its fonts and keyboard shortcuts with the method editor. So if you open the Fonts... dialog from either of these windows, you are editing the same configuration information (stored in keys.json under methodEditorAndRemoteDebugger).

Remote Debugger Toolbar

From left to right, the toolbar controls are as follows. Note that there is no configuration mechanism to change these.



```
Do inherited

Do $extobjects.OW3.$objects.HTTPClientWorker.$newref() Returns iHttp
Do iCurlOptions.$define(#1,#S2)
Do iCurlOptions.$add(123,"First value")
Do iCurlOptions.$add(456,"Second value")
Calculate iHttp.$curloptions as iCurlOptions
Calculate iHttp.$proxyauthtype as kOW3httpAuthTypeDigest                    ## kOW3httpAuthTypeDigest val

Calculate iNull1 as #NULL
Calculate iNull2 as #NULL

Calculate iLongChar as con(kLf,"0123456789012345678901234567890123456789012345678901234567890123456
Do iList.$define(#S1,#S2,iBool)
# Set current list iList
# Build externals list
```

instance_methods.N...

instance_methods.NewWindow subclass/$

iBasic	Object ____64_oBasic
iBinary	(0 bytes)
iBinaryEmpty	(0 bytes)
iBool	False
iChar	Short text
iConstant	0
iCurlOptions	(Empty)
iHash	Object ____65
iHttp	(Empty)
iItemRef	

Object A basic field

Figure 127:

Back

Navigates to the previous method in the history stack for the window. Note that unlike the method editor, there is a separate history stack for each remote debug window. This allows operations such as open superclass methods and open specified class to operate within the context of a single window, and also works more appropriately when you have several windows all paused at a breakpoint.

Forward

Navigates to the next method in the history stack for the window.

View

Open specified class is similar to the Modify Specified Class command in the method editor - when a method line is selected, it opens the method referred to by the command, if one can be identified.

In addition to a context menu command in the tree, the View menu also has a command to go to the superclass methods if relevant (also available on the Modify menu for the method editor).

Find

This allows you to perform find operations on the currently selected method.

Instance

The remote debug window can be associated with an instance. This allows you for example to view instance specific methods or objects that have been added during runtime execution. The instance menu allows you to close the instance, detach the debugger from the instance, or attach the debugger to an instance. Note that this menu is disabled as soon as the remote debug window becomes associated with some executing code (by hitting a breakpoint).

Stack

When execution is paused this allows you to select items on the call stack, or clear the stack.

Go, Step In, Step Over, Step Out

Like the standard method editor, these commands allow you to start execution (Go) and step through your debuggable code. The step commands step until the next debuggable command, so if code which is not debuggable is encountered execution will not pause there.

Go Point

This allow you to set the go point to a different line in the method at the top of the call stack.

Breakpoints

This allows you to manage breakpoints. Note that the method editor has also been changed to remove individual Breakpoint and One-time breakpoint buttons, and use a similar menu to this for consistency.

The Set Condition… command allows you to set a condition on a breakpoint. The condition is a calculation that must evaluate to true for the breakpoint to pause execution. The condition dialog provides some code assistance, by using variable names (of task, class, instance, local, parameter and event parameter variables) present in the currently displayed method.

Note that the code panel and the breakpoint panel both provide alternative ways to work with breakpoints, in a similar way to the method editor - so the left column of the code panel can be used to set and clear breakpoints, and the breakpoints panel has a context menu to do this. Set Condition… is not available in the breakpoints panel context menu, because the method affected may not currently be displayed.

Variable panel

The Variable panel in the Method Editor will be populated while debugging your code remotely, and allows you to view and modify variables.

Keeping the Client in Step with the Server

You should bear in mind that the set of libraries and instances being debugged can change on the server. Omnis keeps the client up to date with the server using a combination of notifications sent from the server, and lazily applied updates to the client. For example, if a library closes on the server, the client is informed, and it updates the interface to reflect this - this means it removes it from the browser tree, and closes any remote debug windows for classes in the library. However, if a method changes on the server, the client will not receive the updated method until it requests it again - note that each time the client performs a debug operation, e.g. step over, the client will request the method when the action completes - if the method has changed on the server, the client will receive a new copy as part of the step action.

Execution

Execution Contexts

An execution context is either the main thread or a remote task instance. When execution suspends for an execution context, the remote debug client looks for **the** debug window associated with the context, and uses that. If there is no debug window for that context, the client looks for a suitable open remote debug window for the class, and if one exists that is not associated with a context it will use it, and associate the window with the context; otherwise the client opens a new window and associates it with the context. Once a window is associated with an execution context, all debugging for that context occurs in that window. A window associated with an execution context can only be closed if execution is not currently paused.

This approach means you can be simultaneously debugging several remote task instances for example. Each execution context uses a single window.

All in one process

As stated earlier, the remote debug client and server can be the same process. In this case:

When execution suspends in the main thread, the remote debug window for the main thread context becomes fully modal.

You cannot debug code running in a critical block in the multi-threaded server.

Errors

If an error occurs during Omnis code execution, e.g. Open window instance with a bad window name, and the line of code causing the error is remotely debuggable, the remote debugger pauses execution at the line causing the error.

Local Debugger

While the remote debugger is attached to a copy of Omnis, the local debugger is disabled in that copy. This also affects the ability to right click and view variable values.

Omnis Language

There is a new sys() function, sys(238) that returns a Boolean which is true if the remote debug server has been started.

Remote Debugger In Control

When the remote debug server and client are not the same process, and execution is suspended for the main thread on the server, the following window appears on the server (this does not apply to the headless server):

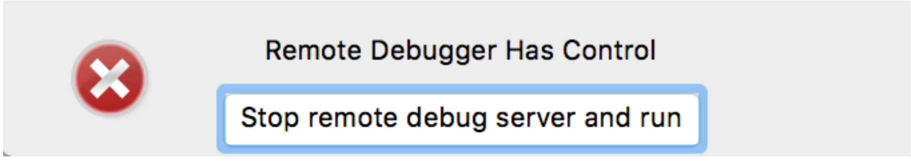


Figure 128:

While this window is displayed, the only action that can be performed on the server is a click on the button to stop the remote debug server, and run (meaning execution continues from where it was paused).

Note that if you do choose to stop the server, then the session will close on the client, and all remote debug windows open on the client will close. If you subsequently restart the server (without restarting Omnis), and open a new session from the client, any breakpoints set for the previous session will still be set.

Chapter 5—Object Oriented Programming

This chapter discusses the object-oriented features used in Omnis Studio, including inheritance, custom properties and methods, and creating and using object classes and external objects.

Note that most of the example code in this chapter is generic and can be applied to all programming tasks; however, some of the example code may relate to window classes only, but the code may be easily adapted to work with remote forms.

Inheritance

When you create a new class, you can derive it from an existing class in your library. The new class is said to be a “subclass” of the existing class, which is in turn a “superclass” of the new class. You can make a subclass from all types of class, *except schema, query, remote menu, code classes, file or search classes*. The general rule is that if you can open or instantiate a class, such as a remote form, then you can make a subclass of that type of class. Omnis does not support mixed inheritance, that is, you cannot make a subclass of one type from another type of class.

Why should I use inheritance?

When you make a subclass, by default, it inherits all the variables, methods, and properties of its superclass. For example, remote form subclasses inherit all the fields and objects, as well as all the methods, from the remote form superclass. Therefore, inheritance saves you time and effort when you develop your application, since you can reuse the objects, variables, and methods from a superclass. When you make a change in a superclass, all its subclasses *inherit the change automatically*. From a design point of view, inheritance forces uniformity in your UI by imposing common properties, and you get a common set of methods to standardize the behavior of the objects in your library.

Making a Subclass

You can make subclasses from the following types of class:

- **Remote Form**
  inherits variables, methods, and properties from its superclass, as well as all fields on the remote form superclass
- **Report**
  inherits variables, methods, and properties from its superclass: note that a report class *does not* inherit the fields, sections, and graphics from its superclass

- **Remote Task**
  inherits variables and methods from its superclass, but none of its properties
- **Task**
  inherits variables and methods from its superclass, but none of its properties
- **Table**
  inherits variables and methods from its superclass, and only some of its properties
- **Remote Object**
  inherits variables and methods from its superclass, but none of its properties
- **Object**
  inherits variables and methods from its superclass, but none of its properties

You can make subclasses from the following types of class, but they apply to desktop apps only (so are not available in the Community Edition):

- **Window**
  inherits variables, methods, and properties from its superclass, as well as all fields on the window superclass; you can inherit the status bar (right-click on the status bar and select Inherit)
- **Menu**
  inherits variables, methods, and properties from its superclass, as well as the menu lines in the menu superclass
- **Toolbar**
  inherits variables, methods, and properties from its superclass, as well as the toolbar controls in the toolbar superclass

To make a subclass

- Open your library in the Browser and show its classes
- Right-click on a class and choose **Make Subclass** from its context menu (if a class does not support subclasses the option will not appear)

When you make a subclass Omnis creates a new class derived from the selected class. The new class inherits all the objects, variables, and methods from its superclass. Omnis supports up to 10 superclass levels, that is, a single class can inherit objects from up to ten other superclasses that are directly in line with it in the inheritance tree. If you create further levels of subclass they do not inherit the objects from the superclasses at the top of the tree.

You can view and edit a subclass, as you would any other class, by double-clicking on it in the Browser. When you edit the methods for a subclass in the method editor, you will see its inherited variables and methods shown in a color. When you view the properties of a subclass its inherited properties are shown in a color in the Property Manager. You can set the color of inherited objects using the **inheritedcolor** Omnis preference. Note that all the Appearance and Action properties for a window are inherited too.

Standard properties for the class, such as the window name and class type, are not inherited, neither are the grid properties. The properties to do with the general appearance of the window, such as **title** and **hasmenus**, are inherited and shown in a color, which defaults to bright blue. You cannot change inherited properties unless you overload them: their values are grayed out in the Property Manager. If you change the properties of the superclass, the changes are reflected in the subclass when you next open it in design mode.

There are some general properties of a class that relate to inheritance, as follows:

Property	Description
$superclass	the name of the superclass for the current class; the superclass can be in another library, in which case the class name is prefixed with the library name
$inheritedorder	for window classes only, determines where the inherited fields appear in the tabbing order: by default inherited fields appear before non-inherited fields on your window
$issupercomponent	if true the class is shown in the Studio Browser as a superclass
$componenticon	icon for the superclass when it is shown in the Studio Browser

Window Status Bar

A window subclass can inherit window status bar panes from a superclass. In design mode, when you right-click on the status bar, the **Inherit** option allows you to inherit the status bar panes from a superclass (the option allows you to toggle the panes on or off). When inherited, you cannot change any of the pane or bar properties in design mode, and these properties are displayed in the Property Manager using the inherited colour.

Subclass Editors

Class editors for subclasses update immediately when the superclass has been changed, so you do not have to close and re-open a subclass to see the changes made to the superclass.

Making a Subclass Manually

You can set the **superclass** property for a class manually, to make it a subclass of the specified class, either using the notation or in the Property Manager.

However, when you make a subclass in this way it *does not inherit* any of the properties of the superclass: *only objects are inherited from the superclass*. You have to open the Property Manager and inherit the properties manually using a context menu.

To inherit a property manually

- View the properties of the subclass in the Property Manager
- Right-click on the property and select Inherit Property from its context menu

If the property cannot be inherited the context menu will display Cannot Inherit and will be grayed out. If the class does not have a superclass the inheritance context menu does not appear or is grayed out.

To inherit a method manually

- Open the method editor for the subclass
- Right-click on the method and select Inherit Method from its context menu

When you inherit a method in this way, Omnis will delete the current method. Any comments in the inherited method will be shown on the left-hand side of the 'Notes' tab in the Variable pane in the Method Editor.

Inherit or Override method Shortcut

The Inherit method or Override method options are present in the method editor Modify menu when it is appropriate to include the command. Both have the shortcut Ctrl+Shift+I to inherit or override the current method.

Overloading Properties, Methods, and Variables

Having created a class from another class using inheritance you can override or overload any of its inherited properties, methods, and variables in the Property Manager or the Method Editor as appropriate. All inherited objects are shown in a color. To overload an object, you can Right-click on the object and select Overload from the object's context menu.

(Note that for windows, menus, and toolbars you cannot overload or delete inherited fields, menu lines, or toolbar controls. If you don't want certain objects to be displayed in a subclass you can hide them temporarily at runtime using the notation.)

To overload a property

- View the properties of the subclass in the Property Manager
- Right-click on the inherited property and select Overload Property from its context menu

When you overload a property, its inherited value is copied to the class and becomes editable in the Property Manager. You can overload any number of inherited properties in the class and enter values that are local to the class.

To reverse overloading, you Right-click on a property and select Inherit Property: the local value will be overwritten, and the value inherited from the superclass will appear in the Property Manager.

To override a method

- View the methods for the subclass in the method editor

- Right-click on the inherited method and select Override Method from its context menu

When you override a method it becomes like a non-inherited method in the class, that is, you can select it and add code to it.

To reverse this process and inherit the method with the same name from the superclass, you can right-click on the method and select Inherit Method: the code in the non-inherited method will be deleted and the inherited method will now be in place. Alternatively, if you override a method and then delete the local one, the inherited method will reappear when you close and reopen the class.

To override a variable

- View the methods for the subclass in the method editor and click on the appropriate tab in the variable pane to find your variable
- Right-click on the inherited variable and select Override Variable from its context menu

When you override a variable it becomes like a non-inherited variable in the class and is only visible in that class.

To reverse this process and inherit the variable with the same name from the superclass, you can Right-click on the variable and select Inherit Variable.

Inheritance Tree

The Inheritance Tree shows the superclass/subclass structure of the classes in your library. All classes below a particular class in the hierarchy are subclasses of that class. When you select a class in the Browser and open the Inheritance Tree it opens with that class selected, showing its superclasses and subclasses above and below it in the tree.

To open the Inheritance Tree for a class

- Select the class in the Browser
- Right-click on the class and select Inheritance Tree from its context menu

Showing Superclasses in the Studio Browser

You can show any class that supports inheritance in the Studio Browser by setting its $issupercomponent property to kTrue. You can specify the icon for a superclass by setting its $componenticon property. If you create a class from a superclass displayed in the Studio Browser, the new class will be a subclass of the class in the Studio Browser automatically.

To make a subclass from a supercomponent

- Select the target library in the Studio Browser
- Click Class Wizard option
- Click on the Class Type option
- Click on the Subclasses option

At this stage, all the supercomponents of that type (i.e. classes that have their $issupercomponent set to kTrue) should be displayed in the wizard.

- Click the supercomponent (superclass) you want to create a subclass of
- Enter a name for the subclass and click on the Create button

Such supercomponents will only appear in the Studio Browser if the library containing it is open, since the class actually remains in your library and is only displayed in the Studio Browser.

Note that classes that appear in the Studio Browser in this way cannot be made the default object for that class.

Inheritance Notation

You can use the $makesubclass() method to make a subclass from another class. For example

```
Do $windows.Window1.$makesubclass('Window2') Returns ItemRef
# creates Window2 which is a subclass of Window1 in the
# current library, ItemRef contains a reference to the new class
```

You can test if the current class can be subclassed by testing the $makesubclass() method with the $cando() method, as follows

```
If $cclass.$makesubclass().$cando()
  Do $cclass.$makesubclass(sClass) Returns ItemRef
  # creates a subclass of the current class
  ...
```

The $makesubclass() method has a Boolean optional second argument (bAddLibPrefixForClib, default kFalse), which when kTrue, causes the $superclass property of the new subclass to include the library prefix for the current library.

You can test if a particular class is a superclass of another class using the CLASS.$isa(SUPERCLASS) method as follows

```
Do $windows.window2.$isa($windows.window1) Returns lresult
# returns true if window1 is a superclass of window2
```

You can change the superclass of a class by reassigning $superclass

```
Do $cclass.$superclass.$assign('DiffClassName') Returns lresult
```

You can test if a property can be inherited using

```
Do $cclass.$style="font-variant: small-caps;">PropertyName.$isinherited.$canassign() Returns lresult
```

If a property can be inherited, you can overload or inherit the property by assigning $isinherited. For example, to overload a property

```
Do $cclass.$style="font-variant: small-caps;">PropertyName.$isinherited.$assign(kFalse) Returns lresult
```

A superclass can be in a different library to a subclass. If you open an instance of a subclass when the superclass is not available, perhaps because the library the superclass belongs to is not open or has been renamed, a kerrSuperclass error is generated.

Libraryname prefix for subclasses

By default, Omnis does not prefix $superclass with the library name (if both classes are in the same library). You can manually change the $superclass property to include the library name if you wish.

Calling Properties and methods

When a property or method name is referenced in a subclass, Omnis looks for it first in the subclass and progressively up the chain of superclasses in the inheritance tree for the current library. Therefore, if you haven't overridden the property or method in the subclass, or at any other level, the property or method at the top of the inheritance tree will be called. For example, the following command in a subclass

```
Do $cinst.$MethodName()
# will call $MethodName() in its superclass
```

However, if you have overridden a property or method in the subclass the method in the subclass is called. You can still access the inherited property or method using the $inherited property. For example, assuming $MethodName() has been overridden in the subclass

```
Do $cinst.$inherited.$MethodName()
# will call the superclass method

Do $windows.MySubWin.$MethodName().$inherited
# will call $MethodName() in the superclass
```

$inherited is present at the top level, and in $cinst for all types of instance (while $default is available for table instances only).

Referencing Variables

When a variable is referenced in a subclass, Omnis looks for its value first in the subclass and progressively up the chain of superclasses in the inheritance tree for the current library. Therefore, if you haven't overridden the variable in the subclass, or at any other level, the value at the top of the inheritance tree is used.

However, if you have overridden a variable in the subclass the value in the subclass is used. You can access the inherited variable using $inherited.VarName, in which case, the value at the top of the inheritance tree is used.

A superclass cannot use the instance and class variables of its subclasses, although the subclass can pass them as parameters to superclass methods. References to class and instance variables defined in a superclass are tokenized by name and the Remove Unused Variables check does not detect if a variable is used by a subclass. If an inherited variable does not exist, its name is displayed as $cinst.VarName in design mode and will create a runtime error.

Inherited Fields and Objects

All inherited window fields on a window subclass are included in the $objs for an instance. Since some field names may not be unique, when $objs.name is used Omnis looks first in the class containing the executing method, then in the superclasses and lastly in the subclasses. The $objs group for report fields, menu lines, and toolbar controls behave in the same way as window fields.

You should refer to fields at runtime by name, since at runtime Omnis assigns artificial $idents to inherited fields. The $order property of a field may also change at runtime to accommodate inherited fields into the tabbing order.

Do inherited Command

You can use the *Do inherited* command to run an inherited method from a method in a subclass. For example, if you have overridden an inherited $construct() method, you can use the *Do inherited* command in the $construct() method of the subclass to execute the $construct() method in its superclass. You could use this command at the end of the $construct() method in the subclass to, in effect, run the code in the subclass $construct() and then the code in the superclass $construct() method.

Inherited Object Notation

You can access the methods of inherited objects in remote forms, windows, menus, toolbars and reports in your Omnis code. Inherited objects are exposed as a new group within the notation group called $inheritedobjs. The members of this group are the inherited objects from all of the superclasses of the class.

Each member of the $inheritedobjs group has three properties: $name, $ident and $isorphan (true when the object no longer belongs to a superclass, but has methods so it cannot be removed without developer approval). In addition, each member has a child $methods group which are the methods implemented in the class for the inherited object. This is just like any other methods group, and methods can be manipulated as you would expect. To override a method from the superclass, simply add a method with the same name as the inherited object method. To inherit a method, delete the method with the same name from the inherited object methods.

Object Classes

Object classes let you define your own structured data objects containing variables and methods. You can create an object variable based on an object class which contains all the variables and custom methods defined in the class. When you reference an object variable an instance of the object class is created containing its own set of values. You can store an object variable instance and its values on a server database (or Omnis datafile). The structure and data handling capabilities of the object instance is defined by the types of variables you add to the object class; similarly, the behavior of an object variable is defined by the methods you add to the object class.

Object classes have the general properties of a class and no other special properties. They can contain class and instance variables, and your own custom methods. You can make a subclass from an object class, which inherits the methods and variables from the superclass.

The HTTP Push example app in the Hub in the Studio Browser uses Object classes to create an HTTP Worker object; see this example app for code using object classes.

To create an object class

- Open your library in the Browser
- Click on New Class and then the Object option
- Name the new object class

- Double-click on the object class to modify it

Alternatively, you may want to create an object class using one of the wizards. In this case, you should click on Class Wizard, then Object, and select one of the wizards.

When you modify an object class, Omnis opens the method editor for the class. This lets you add variables, and your own custom properties and methods to the class.

When you have set up your object class you can create any other type of Omnis variable based on your object class: an object variable has type Object and its subtype is set to the name of your object class. For example, you can create an instance variable of Object type that contains the class and instance variables defined in the object class. When you reference a variable based on an object class you create an instance of that object class that belongs to the current task at the point of their creation (this provides consistency with object instances created via $new). You can call the methods in the object class with the notation ObjVarName.$MethodName(), where $MethodName() is any custom method you have defined in the object class.

You can store object variable instances in a server database that stores binary values (or an Omnis data file). When you store an instance of an object variable in a database, the value of all its contained instance variables are also stored. When the data is read back into memory the instance is rebuilt with the same instance variable values. In this respect you can store a complete record or row of data in an object variable.

You can store object instances in a list. Each line of the list will have its own instance of the object class. Object instances stored in a task, instance, local, or parameter variable belong to the same task as the instance containing that variable. Similarly object instances stored in a list or row belong to the same task as the instance containing the list or row variable. All other object instances have global scope and are instantiated by the default task and belong to the default task.

You cannot make an object instance a private instance. If you delete the object class an object variable uses, the object instance will be empty.

An object instance stored as a class variable in a task is destroyed as soon as its original task is destroyed.

To add variables and methods to an object class

- Open your object class in design mode
- Right-click in the variable pane of the method editor and select the Add New Variable option from the context menu
- Name the variable, give it a type and subtype as appropriate
- Right-click in the Method Names pane of the method editor and select the Add New Method option from the context menu
- Name the method, including the dollar prefix

If you right click on an object variable, and use the Variable <name>… entry in the context menu, the variables list window opens, initially showing instance variable values.

Missing Object Variables

An error is generated when you open a library that tries to construct an Object variable and the Object class it is based on does not exist or is in a library that is not open. In this scenario, you will get a runtime error and execution blocks with the following error:

```
E100101: Class not found when constructing an object variable
```

The class name is TESTB.oTestB
Either the class does not exist or it has the wrong type e.g. remote object

In versions prior to Studio 10 in this case, you would have received the error "Class not found", but code execution would have continued which may have led to further errors in the application.

$cando and Error Handling

Since Studio 10.1. the default behavior when Omnis attempts to construct an object variable when its class does not exist, is to report a debugger error when code attempts to use the object, e.g. via $cando. From Studio 10.2, you can override this behavior, for example, $cando will return kFalse for notation like iObject.$message.$cando.

To override this behavior, you can use the $nofatal property of object variables:

```
Calculate iObject.$nofatal as kTrue
```

- $nofatal
  If true, and the object instance could not be constructed because the object class does not exist, treat this error as a warning when trying to use this object (meaning Omnis will not enter the debugger or abort execution)

When using $cando after setting $nofatal to kTrue, the $cando will return false.

Variable Count

The $usage property reports the current number of object variables that are sharing the underlying external component object. A NULL value means the object is neither an external component object, nor is it subclassed from an external component object.

Note that copies of the object such as $statementobject and $sessionobject contribute to the count.

Using Object Classes

This section describes an invoices example and uses an object class and simple invoices window; it is intended to show what you can do with object classes, not how to implement a fully functional invoices application. You can create the data structure required for an invoice by defining the appropriate variables and custom methods in an object class.

The following example uses an object class called o_Invoice, which contains the variables you might need in an invoice, such as the invoice ID, quantity, value, and a description of the invoice item. The o_Invoice object class also contains any custom methods you need to manipulate the invoice data, such as inserting or fetching invoices from your database. The methods in the object class contain the following code

```
# $SaveInvoice() method contains
# local var lv_InvoiceRow of type Row and
# local var lv_Bin of type Binary and
# parameter var pv_Object of type Field reference
Do lv_InvoiceRow.$definefromtable(t_Invoices)
Calculate lv_bin as pv_Object
Calculate lv_InvoiceRow.InvoiceObject as lv_bin
Do lv_InvoiceRow.$insert() Returns #F
# $SelectInvoice() method contains
# local var lv_Row of type Row
Do lv_Row.$definefromtable(t_Invoices)
Do lv_Row.$select()
Do lv_Row.$fetch() Returns #S1
Quit method lv_Row.InvoiceObject
# $FetchInvoice() method contains
# local var lv_Row of type Row
Do lv_Row.$definefromtable(t_Invoices)
Do lv_Row.$fetch()
Quit method lv_Row.Inv_Object
# $total() method

Quit method iv_QTY * iv_Value
```

The invoice window can contain any fields or components you want, but would contain certain fields that reference the instance variables in your object class, and buttons that call the methods also in your object class.

The invoice window contains no class methods of its own. All its functionality is defined in the object class. The window contains a single instance variable called iv_Invoice that is based on the o_Invoice object class.

The instance variable would have the type Object and its subtype is the name of your object class, o_Invoice in this case. If the object class is contained in another library, the object class name is prefixed with the library name.

When you open the invoice window an instance of the object class is created and held in iv_Invoice. Therefore you can access the instance variables and custom methods defined in the object class via the instance variable in the window; for example, iv_Invoice.iv_QTY accesses the quantity value, and iv_Invoice.$SaveInvoice() calls the $SaveInvoice() method. Each field on the invoice window references a variable in the object class; for example, the dataname of the quantity field is iv_Invoice.iv_QTY, the dataname of the item or description field is iv_Invoice.iv_Item, and so on.

The buttons on the invoice window can call the methods defined in the object class, as follows.

```
# $event() method for the Select button
On evClick
  Do iv_Invoice.$SelectInvoice(iv_Invoice.iv_ID) Returns iv_Invoice
  Do $cwind.$redraw()

# $event() method for the Fetch button
On evClick
  Do iv_Invoice.$FetchInvoice Returns iv_Invoice
  Do $cwind.$redraw()

# $event() method for the Save button
On evClick
  Do iv_Invoice.$SaveInvoice(iv_Invoice)
```

When you enter an invoice and click on the Save button, the $SaveInvoice() method in the object class is called and the current values in iv_Invoice are passed as a parameter. The $SaveInvoice() method receives the object instance variable in the parameter pv_Object and executes the following code

```
Do lv_InvoiceRow.$definefromtable(t_Invoices)
Calculate lv_bin as pv_Object
Calculate lv_InvoiceRow.InvoiceObject as lv_bin
Do lv_InvoiceRow.$insert() Returns #F
```

The row variable lv_InvoiceRow is defined from the table class t_Invoices which is linked to the schema class called s_Invoices which contains the single column called InvoiceObject. The binary variable lv_bin, which contains the values from your object instance variable, is assigned to the row variable. The standard $insert() method is executed which inserts the object variable into your database. The advantage of using an object variable is that all the values for your invoice are stored in one variable and they can be inserted into a binary column in your database via a single row variable. If you want to store object variables in an Omnis database you can create a file class that contains a single field, called InvoiceObject for example, that has Object type, rather than Binary, and use the appropriate methods to insert into an Omnis data file.

Libraryname prefix for object variables

There is an “Include library prefix” check box on the object class selection dialog. This allows you to add the library name to the object class name for the variable, but when the object class is in the same library you don’t need to add the libraryname prefix.

Dynamic Object Instances

The object class has a $new() method that lets you create an object instance dynamically and store it in an object variable, for example

```
Do $clib.$objects.objectclass.$new(parm1,parm2,...) Returns objectvar
```

where parameters parm1 and parm2 are the $construct() parameters for the object instance. When the instance is assigned, any existing instance in the object variable is destroyed. It would be normal practice to put no class in the variable pane for object variables which are to be set up dynamically using $new(), but there is no class checking for instance variables so no error occurs if the class shown in the variable pane is different from the class of the new instance.

You can do a similar thing with an external function library if it contains instantiable objects, such as Fileops. For example

```
Do Fileops.$objects.fileops.$new() Returns objectvar
Do objectvar.$openfile(pFilename)
```

The following example uses an Object variable ivSessionObj which has no subtype defined in the method editor. When this code executes ivSessionObj is instantiated based on the object class SessionObj which was created using the Session Wizard in Omnis. Once the object instance exists the $logon() method is called.

```
Do $clib.$objects.SessionObj.$new() Returns ivSessionObj
# runs $construct() in the SessionObj object class

Do ivSessionObj.$logon()
# runs the $logon() method in SessionObj
```

Self-contained Object Instances

Object classes have the $selfcontained property. If set to kTrue the class definition is stored in all instances of the object class. An instance of such an object class is disconnected from its class and instead relies on its own class data for method and instance variable definitions. When the object instance is stored on disk the class definition is stored with the instance data and is used to set up a temporary class whenever the instance is read back into memory. Any changes to the original object class have no effect on existing instances, whether on disk or in memory.

Once an instance is self-contained it is always self-contained, but you can change its class definition by assigning to $class, for example

```
Do file1.objvar1.$class.$assign($clib.$objects.objectclass1)
```

causes the class definition stored inside objvar1 to be replaced by the current method and variable definitions for objectclass1. The instance variable values are maintained provided the new instance variable definitions are compatible with the old ones (Omnis can handle small changes in the type of variables but won't carry out substantial conversions). Note that the old instance variables are matched to the new ones by $ident and not by name, so to avoid problems the new class should be a descendant of the original class with none of the original variables having been deleted.

Only the main class is stored with the object instance, inheritance is allowed but any superclasses must exist in open libraries whenever the instance is present in memory. Assigning to $class does not change the superclass structure of self-contained instances.

Object References

The Object reference data type provides non-persistent objects that you can control using notation. Non-persistent means that objects used in this way cannot be stored on disk, and restored for use later.

You can use the Object reference data type with local, instance, class and task variables. Object references have no subtype. To create a new Object instance, referenced by an Object reference variable, you use the methods $newref() and $newstatementref(). These are analogous to the $new() and $newstatement() methods used to create object variables, and they can be used wherever $new() and $newstatement() can be used. When created, an object reference will belong to the current task, and are no longer valid after the task closes.

Once you have associated an Object instance with an Object reference variable, you can use it to call methods just like you would with an Object variable.

The instance associated with an object reference variable will be destroyed in the following circumstances:

- When they are no longer required, object references are deleted automatically in order to free up memory (when a variable or list column no longer contains the reference, for example).
- You can use the $deleteref() method to delete the object reference, if you want to release memory sooner than would otherwise occur under the automatic process.
- The task that created the variable is closed.
- The library is closed.

Until one of these occurs, Object reference variables can be passed as parameters, copied, stored in a list column, and so on. All copies of the variable address the same single instance of the object. As soon as the object is deleted, other references to the object become invalid. Prior to deleting the object, Omnis calls the destructor method.

Note that this approach means that you must delete objects when you have finished with them, otherwise significant memory and resource leaks can occur.

There is also a method $listrefs() which you can call to list the instances of an object class, external object, or in fact *all* instances created in this way. This is useful both for leak-checking, and for clean-up.

You can use $newref() in conjunction with session pools. The only differences are that $deleteref() returns the object instance to the pool, rather than destroying it, and $listrefs() does not include objects from a session pool.

There are two further methods that can be used with object reference variables. $copyref() creates an object instance which is a copy of the instance referenced by the variable, and $validref() returns a Boolean which is true if and only if the variable references a valid object instance. Note that if $copyref() takes a copy of a DAM session object, both copies reference the same DAM session.

The method $objref() can also be used. If there is an object reference associated with the instance, the method returns the object reference, otherwise returns #NULL, that is, the object MUST have been created with a $newref().

Constructing new objects

...from an object class

```
Calculate myObjectReference as $clib.$objects.myObject.$newref("Test")
```

...from an external object

```
Calculate myObjectReference as $extobjects.ODBCDAM.$objects.ODBCSESS.$newref()
```

...from a session object

```
Calculate myObjectReference as mySessionObjectReference.$newstatementref()
```

...storing the reference in a list column of type Object reference

```
Do theList.$add($extobjects.ODBCDAM.$objects.ODBCSESS.$newref())
```

Listing instances created using an Object reference

...instances of an object class

```
Calculate myList as $clib.$objects.myObject.$listrefs()
```

...instances of an external object

```
Calculate myList as $extobjects.ODBCDAM.$objects.ODBCSESS.$listrefs()
```

...all instances

```
Calculate myList as $listrefs()
```

Destructing objects

...using an object reference variable:

```
Do myObjectReference.$deleteref() ## Calls the destructor of the object instance
```

...all instances referenced by a list column of type Object reference:

```
Do myList.$sendall($ref.1.$deleteref())
```

...using the $listrefs() method:

```
Do $extobjects.ODBCDAM.$objects.ODBCSESS.$listrefs().$sendall($ref.1.$deleteref())
Do $libs.DAMTEST.$objects.Doracle8.$listrefs().$sendall($ref.1.$deleteref())
```

Testing an object reference for validity

...using an object reference variable:

```
If myObjectReference.$validref()
```

...using a list column of type Object reference:

```
If myList.1.myObjRef.$validref()
```

Copying an object reference

…using an object reference variable:

```
Calculate objectReference as myObjectReference.$copyref()
```

…using a list column:

```
Calculate objectReference as myList.1.myObjRef.$copyref()
```

External Objects

External objects are a type of external component that contain *methods* that you can use by instantiating an object variable based on the external object. External objects can also contain *static functions* that you can call without the need to instantiate the object. These functions are listed in the Catalog under the Functions pane.

Some external objects are supplied with Omnis Studio; these include equivalents to the FileOps and FontOps externals, a Timer object, as well as the multi-threaded DAMs. Writing your own external objects is very similar to writing external components. The FileOps and FontOps functions are documented in the Omnis Help.

External objects are created and stored in component libraries in a similar manner to external components, and in future releases are intended to replace external functions and commands, although to maintain backward compatibility, the old external interface is still supported at present.

External object libraries are placed in the XCOMP folder, along with the visual external components. They must be loaded in the same way as external components using the External Components option, available in the Browser when your library is selected.

Using External Objects

You can add a new object in the method editor by inserting a variable of type Object and using the subtype column to select the appropriate external object. You can click on the subtype droplist and select an external object from the Select Object dialog. This dialog also appears when you create an object elsewhere in Omnis. An icon in the variable subtype cell shows whether the variable is based on an object class or an external object.

When an instance of the external object has been constructed, you can inspect its properties and methods using the Interface manager.

To use the object's methods in your code, you can drag the method you require from the Interface Manager into the Code Editor.

For some objects it is important to note that for the Interface manager to interrogate an object it will need to be constructed. For example, if the Interface Manager was used on an Automation object, the Automation server needs to be started.

External objects are contained in the notation group $extobjects, which you can omit from notation.

External Object Events

External objects do not support events in the GUI sense. They can however define notification methods which they call when certain events occur.

You can subclass an external object and then override the notification method so your code is informed of the event. The Timer object supplied in Omnis is an example of this. To subclass an object, you can either set the superclass property in the Property Manager, or use the New Subclass Object wizard available in the Browser (using the Class Wizard>>Object option).

External Object Notation

All the components available to Omnis are listed in the $root.$components group. Individual components have their own methods, in addition to the $cmd() method which allows you to send a component specific command to the component or object.

The following examples show some commands that you can send to an ActiveX component to manage timeouts:

```
Do $components.ActiveX.$cmd("RequestPendingTimeout",10000)
# Set timeout in milliseconds

Do $components.ActiveX.$cmd("ServerBusyTimeout",10000)
# Set timeout in milliseconds
```

```
Do $components.ActiveX.$cmd("ResponseDialog",kFalse)
# Prevent response dialog from appearing

Do $components.ActiveX.$cmd("BusyDialog",kFalse)
# Prevent busy dialog from appearing
```

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 from a database, 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 a list 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, and Data grids, as well as Bar and Pie 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*.

Note that most of the example code in this chapter is generic that can be applied to all list handling, including lists containing SQL data and list variables contained in Remote forms used for creating web and mobile apps; however, some of the example code may relate to window classes only, but the code may be easily adapted to work with remote forms.

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, or in the Create Variable dialog in the Code 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 va library
Instance variable	on opening instance	within the instance only	on closing instance
Local variable	on running method	within the method only	when method term
Parameter variable	on calling the method	within the recipient method	returning to the cal

To declare a list or row variable in the Variable pane

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

To declare a list or row variable in your code

You can declare a list or row variable directly in your code, by first naming it in your code and then using the **Create Variable** dialog, as follows:

- Type the variable name in your code in the Code Editor, e.g. iCustomerList; note it will not be recognized as a variable and will be underlined to indicate an error

When you type the name of a new variable in your code, you can specify the initial *scope* and *type* for the variable using a predefined *prefix* and *suffix*, respectively, so in this case you can type iCustomerList to specify an instance variable of List type.

- Click into the variable name, and click on the **Fix** button at the bottom of the Code Editor window (a blue button with a check icon)
- In the **Create Variable** dialog, specify the **Scope** (e.g. Local or Instance), then select *List* or *Row* from the **Type** list (note you can select a Schema, Query or Table class as the **Subtype** if required) and click on **Create Variable;** if you have used a predefined prefix and suffix you may not have to change the scope and variable type

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

- from variables
- from a schema, query, or table class

If you want to use a list in a 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.

(A list can be defined from a File class, but this class type is only available for backwards compatibility in legacy apps using Omnis datafiles.)

Defining Lists from Variables

To define a list from a number of variables you can use the $define() method (the equivalent of the old *Define list* command in previous versions). For example

```
# The variables for the list columns need to be declared
# cvList1 of List type
# cvCol1 of Short integer type
# cvCol2 of Character type
# cvCol3 of Date Time (Short date 1980..2079) type

Do cvList1.$define(cvCol1, cvCol2, cvCol3)
```

This method will define the list cvList1 with the columns cvCol1, cvCol2, cvCol3. You can define a list with up to 32k columns, although you should limit the size of the list to only the columns that are required. The data type of each field or variable defined in the list determines the data type of the corresponding column in the list.

When **$defineresolvesfieldrefs** is set to kTrue (default is kFalse), if a field used to define a list is a field reference, Omnis resolves the field reference and defines/redefines the list using the resolved field.

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 $definefromsqlclass() method (the equivalent of the old *Define list from SQL class* command in previous versions). Alternatively, you can create the variable in the Variables pane in the Method Editor, or the Create Variable dialog, and set a SQL class name as the Subtype of the list or row variable.

Defining a list or row based on one of the SQL classes binds the 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 can have its **$sqlclassname** property set to the associated schema or query class to get the definition from the SQL 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')
```

You can however use a table class without any schema or query class assigned in $sqlclassname. In this case, the table class has a $load method that has a manual SQL statement, i.e. with Begin and End SQL. The list variable will then define the list columns from the select result automatically.

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.

- **$select()**
  issues a select statement to the server

- **$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

- **$insert**()
  inserts a row into the server database (row variables only)

- **$update(old_row**)
  updates a row in the server database (row variables only)

- **$delete**()
  deletes a row from the server database (row variables only)

- **$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 SQL Worker Lists.

Defining Lists from File classes

For legacy apps using Omnis datafiles, 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 $createinstance-whensubtype 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:

- **cName**
  the name of the new column
- **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)
- **subtype**
  the subtype of the new column; only applies to some major types
- **maxlen**
  for some major types such as Character you can specify the maximum length

Legacy List Commands

All the commands in the **Lists** and **List Lines** groups, such as *Define list* and *Search list,* have been deprecated in Omnis Studio 11 and are no longer visible in the Code Assistant in the Code Editor (they will not appear when you type the first few characters); however, they are still present in Studio 11 and will continue to function in legacy code. You can show these commands by disabling the appropriate Command Filter in the **Modify** menu in the Code Editor.

You should use the equivalent methods where available, such as $define() instead of *Define list*, $search() instead of *Search list*, $sort() instead of *Sort list*, and so on, to manipulate the contents of list variables. Various sections in this chapter will have example code for both the list methods and the old list commands, and for all new applications you should use the list methods.

Building List Variables

You can build lists from SQL data using a SELECT statement and the $fetch() method.

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:

- $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 "Variable <var_name>" option (the first option in the context menu), which will open a table containing the list contents. You can do this wherever the variable name appears in Omnis, including the Variable pane in the method editor, the Code editor or 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:

- 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)
- 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).
- 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.
- 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 using the $loadcols() method. The following example for the Sidebar window control uses the $loadcols() method to load values based on the chosen icon.

```
# $event method for sidebar component, incl pLinenum (Integer)
On evIconPicked
  Do iSidebarList.$line.$assign(pLinenum)
  # selects the list line according to item selected in sidebar
  Do iSidebarList.$loadcols()
  # loads the values in the selected list line, including iID

  Switch iID ## branches according to value of iID
    Case 1
      Do something..
    Case 2
      Do something..
    Case 3
      Do something..
  End Switch
```

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, row, ColumnName)
```

You can address cells directly by referring to them as ListVarName.ColumnName for the current row or ListVarName.RowNumber.ColumnNa 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 remote form or 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)  ## this legacy code works, but the above is the preferred method
```

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
```

As part of resolving list notation (e.g. List.C1), when Omnis accesses a list it will automatically convert a NULL list variable to empty.

List Column Calculations

To allow for expressions like myList.col or myList.10.col where the list line does not exist, perhaps because the list is empty, you can set the library preference $validcolumninbadrowisnull ($clib.$prefs) to true. If true, non-existent list columns in calculations evaluate to #NULL rather than an empty character string.

List line commands

The *List Lines* group of commands (that includes the *Load from list* command) are now obsolete, and they are no longer visible in the Code Assistant in the Code Editor (although they are still present in Studio 10 and will continue to function in legacy code). You can show these commands by disabling the appropriate Command Filter in the Modify menu in the Code Editor.

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.

- **$linecount**
  returns the number of lines in the list; you can change this property or use *Set final line number* to truncate the list
- **$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
- **$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 $search()
- **$colcount**
  returns the number of columns in the list
- **$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

- **$class**
  returns the schema, query, or table class for the list, or is empty if it is not based on a SQL class

- **$cols**
  group containing 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)

- **$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.

- **$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

- **$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

- **$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

- **$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

- **$clear**()
  clears the data for the list, but keeps the list definition

- **$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

- **$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

- **$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

- **$addbefore**()
  $addbefore(list row or row number,col1 value[, col2 value]…) inserts a row before the specified row

- **$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)

- **$remove**()
  $remove(list row or row number) deletes the specified row

- **$search**()
  $search(calculation [,bFromStart=kTrue, bOnlySelected=kFalse, bSelectMatches=kTrue, bDeselectNonMatches=kTrue]) searches the list; behaves the same as the *Search list* command; bOnlySelected restricts the search to selected lines. If bFromStart is kTrue, Omnis searches all of the lines in the list, starting at line 1; otherwise, Omnis starts the search after the current line ($line + 1).

- **$count**()
  $count([bSelectedLinesOnly=kFalse]) returns the count of lines in a list, optionally passing bSelectedLinesOnly as kTrue to only count selected lines (you can also use LISTCOL.$count() to return the number of lines in a list column)

- **$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).

- **$merge**()
  $merge(list or row[, by name, only selected]) merges the two lists; note $merge() cannot use search criteria to merger data

- **$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):

- **$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

- **$addafter()**

  $addafter(rColumn|iColumnNumber,{fieldname|cName[,type,sub-type,iMaxlen]}) adds a column to the list and returns an item reference to it

- **$addbefore()**

  $addbefore(rColumn|iColumnNumber,{fieldname|cName[,type,sub-type,iMaxlen]}) adds a column to the list and returns an item reference to it

- **$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:

- **$name**
  returns the simple name of the column

- **$dataname**
  returns the dataname of the list column; empty for a list defined from a SQL class

- **$coltype**
  returns the data type of the column; changing this clears the list data

- **$colsubtype**
  returns the data subtype of the column; changing this clears the list data

- **$colsublen**
returns the length of character and national columns; changing this clears the list

List columns have the following methods:

- **$clear**()
Clears the data for a list or row, or a column in a list or row; executing List.$clear() for a smart list sets $smartlist to kFalse, meaning that it is no longer a smart list
- **$average**()
$average([bSelectedLinesOnly=kFalse]) Returns the average of the non-null list column values
- **$minimum**()
$minimum([bSelectedLinesOnly=kFalse]) Returns the minimum of the non-null list column values
- **$maximum**()
$maximum([bSelectedLinesOnly=kFalse]) Returns the maximum of the non-null list column values
- **$count**()
$count([bSelectedLinesOnly=kFalse]) The count of non-null values in the list column (you can also use LIST.$count() to return the number of lines in a list)
- **$removeduplicates()**
$removeduplicates([bSortNow=kFalse,bIgnoreCase=kFalse]) removes all list rows with duplicate values in the column; you must sort the list on the column before using this method, or you can pass bSortNow as kTrue to force the list to be sorted prior to running the method. bIgnoreCase affects character values only.
Note: The bSortNow parameter is ignored and always treated as kFalse in client methods.
- **$selectduplicates**()
$selectduplicates(listname.column) selects all list lines with duplicate values in the column; you must sort the list before using this method; the list selection state of non-duplicate lines is cleared; this can be used in client-executed remote form methods, as well as in server methods
- **$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.

Properties and Methods of a List Row

A list row has the following properties:

- **$group**
returns the list containing the row
- **$selected**
returns true if the row is selected

A list row has the following methods:

- **clear**()
clears the value of all the columns in the row
- **$loadcols**()
$loadcols(variable1[, variable2]...) loads the column values for the row into the specified variables
- **$assigncols**()
$assigncols(column1 value[, column2 value]...) replaces the column values for the row with the specified values
- **$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.

- **$group**
  returns the list row containing the list cell
- **$ident**
  returns the column number for the list cell
- **$name**
  returns the column name for the list cell
- **$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.

- 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
- List.$cols.$add(colname, type, subtype, length)
  adds a column to the right-hand end of the list using the specified definition
- 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
- 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
- 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
- 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, bDeselectNonM
```

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 bSelectMatches and bDeselectNonMatches are kFalse, the list's current line is set to the first matched row.

$search is optimized to operate on a single line at a time, so your calculation cannot contain multiple line conditions.

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 $multipleselect property. When the user highlights list lines with the mouse, the $selected property for those lines is set. If the field does not have $multipleselect set, 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(list, 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:

- the *normal* list containing the list data, and
- 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:

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

- **$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:

- **$status**
  the status of the row: not assignable
- **$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 $rowpresent and override your changes (deleting a row, applying or rolling back a filter, etc.)
- **$rownumber**
  the row number of the row in the normal list, or zero if $rowpresent is false; not assignable
- **$filterlevel**
  the number of filters applied to the history list, up to 15: not assignable (see filtering below)
- **$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

- **$currentcontents**
  the current contents of the row in the normal list: not assignable
- **$errorcode**
  the error code for the row; assignable and initially zero
- **$errortext**
  the error text for the row; assignable and initially empty
- **$nativeerrorcode**
  native error code generated by last statement command
- **$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:

- Updating a row of status kRowUnchanged changes it to kRowUpdated; updating a row with any other status leaves the status unchanged
- Inserting a row always sets the status to kRowInserted and makes the row present in the normal list
- 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:

- **$savelistdeletes**()
  removes rows with status kRowDeleted from the history list, and also from the normal list if $rowpresent is kTrue
- **$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 $rowpresent
- **$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 $rowpresent
- **$savelistwork**()
  quick and easy way to execute $savelistdeletes(), $savelistinserts() and $savelistupdates()

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

- **$revertlistdeletes**()
  changes the status of all kRowDeleted rows to kRowUnchanged or kRowUpdated (depending on whether the contents have been changed); for these rows $rowpresent is set to true
- **$revertlistinserts**()
  removes any inserted rows from both the normal list and the history list
- **$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

- **$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.

- **$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

- **$doinserts**(), **$dodeletes(), $doupdates()**
  inserts, deletes, or updates any rows in the list with the row status kRowInserted, kRowDeleted, or kRowUpdated, respectively
- **$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.

- *Search list* and equivalent notation selects only lines in the normal list.
- *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.
- 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.
- 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.
- 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.

Chapter 7—SQL Programming

The SQL Browser lets you connect to a wide range of server databases, and the SQL Form wizard lets you build the interface to your server database, quickly and easily. However, you may want to modify the SQL forms created automatically or create forms from scratch to enhance your web and mobile apps. To do this, you need to use or customize the *SQL methods*.

The type of database you can access in Omnis Studio will depend on the edition of Omnis Studio you have; all versions allow you to access the following databases:

- **PostgreSQL** 8 and later
- **SQLite** data files v3 and later

In addition to those above, other editions, including the Professional Edition, allow access to:

- **Oracle** 9i R2 and later
- **Sybase** Adaptive Server Enterprise 12 and Sybase SQL Anywhere 9 and later
- **DB2** Universal Server / DB2 Express 9 and later
- **MySQL** 4.2 and later
- Plus all **ODBC**-compliant databases, such as MS SQL Server, and other file systems such as **SAP HANA**
- You can access an Omnis database (data file) using the Omnis SQL DAM, but this is provided for backwards compatibility in legacy apps only and should not be used for new applications

This chapter covers features of SQL programming that are common to all supported databases, but any differences are highlighted where appropriate. For information about features that are specific to individual supported databases, see the Server-Specific Programming chapter.

(Note that support for JDBC has been removed in Studio 10 or above, but the supporting files can be obtained by contacting Omnis Support.)

Overview

The *Object DAMs* (Data Access Modules) provide an object-oriented mechanism for establishing connections to a variety of SQL databases and enable you to perform multi-threaded database access as part of an Omnis Server in a web application. The DAM interface uses objects to represent a database session and session statements. These objects provide properties and methods that let you invoke the required database functionality. Using the object-oriented approach, an application creates object variables of a particular DAM class that are instantiated when the object is created by the component. This is known as a *session object* and represents a session of one of the supported DAM types. There is a group of common operations that apply to all session objects and a set of database specific operations based on the type of session object. For example, if an application requires an ODBC session, it uses an ODBCSESS session object. The session object controls the connection environment used to pass commands to the database server. The application creates a *statement object* in order to issue SQL. A statement object can be used for all types of statements, e.g., SQL, PL/SQL, cursors and remote procedures. There can be multiple statement objects which share a common context provided by a single session object.

In the Omnis Server multi-threaded environment there will be multiple clients each accessing session objects on the same server. Although it is possible to allow each client to use its own object this may use large amounts of system resource. Therefore, additional functionality is provided to create and maintain session object pools where each is of the same type. An application can allocate session objects from this pool. Each object in the pool is connected with the same attributes when the pool is created. This has an advantage in performance when a client requests an object since the connection is immediately available.

Setting up a Database Connection

Clientware

To connect to one of the supported databases, you may need to install the appropriate clientware on each machine that will be connecting to your server. The Omnis DAM uses the clientware to connect to the remote server.

We recommend which third party clientware you can use for each of the supported platforms and DAM connections. This information is available on the Omnis DAMs developer web site. For each database type, a recommended driver name and version is provided. This is the driver which we have tested and certified against and which we recommend you to use.

For most platform/database combinations however, there will be other drivers which work comparably. In the event of technical support issues, it is the drivers listed on our web site which we ask faults to be verified against.

64-bit DAMs

The DAMs provided with the 64-bit version of Omnis Studio use 64-bit architecture. This means that you will need to install separate 64-bit clientware where appropriate. The 64-bit DAMs are not interoperable with 32-bit client libraries and vice-versa. For single-tier and embedded DAMs, including DAMPGSQL, DAMSQLITE, DAMMYSQL and DAMAZON, all necessary changes have been made (and DAMOMSQL for legacy apps only). The 64-bit ODBC DAM requires the 64-bit ODBC Administrator library and should be used with 64-bit ODBC Drivers to ensure compatibility.

INI files under macOS

Certain object DAMs available for macOS, namely DAMODBC, DAMSYBSE and DAMORA8, make use of ".ini" files in order to set system environment variables to be required by their associated client libraries. These files are named after the DAMs to which they apply and reside inside the Omnis package; in the Contents/MacOS/xcomp/ini folder.

Please note that for Studio 10.0 and later, ".ini" files are no longer used. Instead, please refer to the "macos" section inside the studio/config.json file which now contains *keys* for "odbcdam.ini", "oracledam.ini" and "sybasedam.ini". Each corresponding *value* can consist of one or more comma-separated values, for example:

```
"oracledam.ini": "TNS_ADMIN=/instantclient_12_2, NLS_LANG=AMERICAN_AMERICA.WE8ISO8859P1"
```

If you are running Omnis from the command line (i.e. using the "open omnis.app" command), you can set environment variables from the context of the terminal window before starting Omnis, hence negating the need for these files. If used however, their values will override any existing values. For example:

```
cd /Applications
export TNS_ADMIN=/instantclient_12_2
export NLS_LANG=AMERICAN_AMERICA.WE8ISO8859P1
open "Omnis Studio 10.2.app"
```

Connecting to your Database

Aside from *hostname*, *username* and *password* information, the session templates in the SQL Browser contain all the necessary information required to connect to your server database automatically. However, to connect to your database programmatically you need to create a session object and log the session object onto the database.

The session object is the primary object that controls the session environment from which statement objects are created. This includes the connection, the transaction mode and any common statement properties such as the size of large object data chunks. The instance of an object is created either explicitly via the $new() external object method, or where it is first used. This initialises the object with default property values. When the object goes out of scope or is closed explicitly, depending on the session state, any statements are closed and the connection is logged off.

Creating a Session Object

Create a variable of data type Object in the Variable Pane of the method editor and set the subtype to the session type required for the database connection.

Connection	Session Type
PostgreSQL	PGSQLSESS
SQLite	SQLITESESS
Oracle	ORACLE8SESS
Sybase	SYBASESESS
DB2	DB2SESS
MySQL	MYSQLSESS
ODBC	ODBCSESS
Omnis SQL	OMSQLSESS

Note that most of these will only appear in the list of available session types if the relevant client software has been installed on your computer. DAM objects that fail to load due to missing or incompatible client software will leave an entry in the Trace Log when Omnis is started; the Trace log can be viewed in the Studio Browser or via the Tools menu or toolbar.

Any type of variable may be used to create a session object. However, a task variable is particularly suitable since it enables a variety of objects within your library to easily access the database session.

See the section on External Objects in the *Object Oriented Programming* chapter for further information about creating object variables.

Logging on to a Session Object

The $logon() method is used to log on to the host server using a valid username and password. This establishes a connection that this session can then use to send commands to the server.

You can log on to a database using a method call of the form:

```
Do SessObj.$logon(pHostName,pUserName,pPassword,pSessionName) Returns #F
```

The parameters pHostName, pUserName, pPassword contain information required to connect to the host database. The values of these parameters vary according to the session object and database. If the pUserName and pPassword parameters are left empty, and depending on the session object type, the user may be prompted by the database client to enter the information. See the Server-Specific Programming chapter for more information.

The optional parameter pSessionName is used to name the new session object and results in the session appearing under the SQL Browser tab and in the notational group; $sessions.

If the $logon() method is successful, it returns a value of kTrue, a connection to the database is created and the $state property is set to kSessionLoggedOn. The read-only properties $hostname, $username and $password of the session object are set to the values that were supplied.

For example, to log on to a session named “MySession” to a SQL Server data source named “MyDb” using an object variable “SessObj”

```
Do SessObj.$logon('MyDb','','','MySession') Returns #F
```

There is no limit to the number of session objects that you can create, except the limits imposed by memory resources and server restrictions.

Using Object References

There are two new properties $sessionobjref and $statementobjref of a list or row defined from a SQL table, and one new property of a session in $sessions called $sessionobjref. These are equivalent to $sessionobject and $statementobject, except that they work exclusively with object references.

Logging Off from a Session Object

You need to log your session off from the server when you have finished with it using the $logoff() method.

The connection is dropped and the session $state property is set to kSessionLoggedOff. Depending on the session state, statements are cleared, cursors used by statements are closed and pending results are cleared. This call will fail if the session is not currently logged on. If it fails for any other reason, the connection may be in an undefined state and the current object should be destroyed. If there was an active transaction on this session, the behavior of the DBMS will determine if that transaction is committed or rolled back. The $hostname, $username and $password properties are cleared.

Interacting with your Server

Once a user is logged into a server, they can make use of all the tables and views to which they have been granted access. To send SQL commands to the server, a statement object must first be created.

Creating a Statement Object

A new statement object is created using the $newstatement() method. To return a new statement object in variable StatementObj with name “MySql” use:

```
Do SessObj.$newstatement('MySql') Returns StatementObj
```

The statement object can then be used to send commands and process results.

The variable StatementObj is defined as an object variable with no subtype. Again, any type of variable may be used but a task variable is convenient to enable a variety of objects within your library to easily access the statement and its results.

If successful a statement object is returned otherwise this call has no effect possibly due to insufficient resources. The new object has a $statementname property value of “MySql” and defaults for all other properties.

If the parameter to $newstatement() is omitted, each statement name is automatically generated. These names are of the form “statement_1”, “statement_2”, etc.

Mapping the Data

Before a client application can get any data from a server, it must set up a corresponding place in Omnis to hold the data. This involves mapping the structure of the data, including column names and data types. Typically, you do this using Omnis schema classes. You can define a schema to include all columns of the server table or view, or any subset of the columns. In addition, you can create query classes that use columns from one or more schema classes.

You can use schema and query classes to define list and row variables to handle your server data. Information on creating schema, query, and table classes will be found earlier in this manual, as will details on using list and row variables.

Mapping Character Columns

The definition of Character columns in Schema classes has changed in Studio 10 and now allows lengths from 0xffff to (100000000 - 1) to be stored correctly. In previous versions, the column sublen of 65535 or greater would have been mapped to 100000000.

Sending SQL to the Server

To send SQL to the server, you can either write your own methods, or use the table instance methods that generate SQL automatically and handle both single row and bulk SQL transactions. SQL statements must first be prepared and then executed. If the statement returns results these may then be fetched.

Preparing a SQL Statement

When a SQL statement is prepared, it is sent to the server for verification and if valid is ready to be executed. A single line SQL statement may be prepared using the $prepare() method. Omnis sends to the server whatever you pass as a parameter to the method. It can be standard SQL or any other command statement the server can understand. For example

```
Do StatementObj.$prepare('SELECT * FROM authors ORDER BY au_lname,au_fname') Returns #F
```

A value of kTrue is returned if the statement was successfully prepared, kFalse otherwise indicating that an error occurred. See the section on *Error Handling* for more information. Once a statement has been successfully prepared, the value of the $state property is set to kStatementStatePrepared.

A SQL statement may also be built up using a number of lines of method code as follows:

```
Begin statement
Sta: SELECT * FROM titles
If iPrice>0
  Sta: WHERE price>=[iPrice]
End if
Sta: ORDER BY title
End statement

Do StatementObj.$prepare() Returns #F
```

The *Begin statement* and *End statement* block contains the SQL statement each line of which is contained in an *Sta:* command. Other method commands such as *If*, *End if* etc. may be included inside the block in order to build up the SQL statement using logic.

Note that in this case a $prepare() method with no parameters is used to prepare the statement block.

When an Omnis web server is operating in a multi-threaded mode, the $prepare() method uses the statement buffer of the current method stack.

Once a SQL statement has been prepared it is possible to obtain the contents of the statement as a text string using the $sqltext property, for example

```
OK message The current SQL is {[StatementObj.$sqltext]}
```

The *Get statement* command returns a copy of the statement buffer created by the *Begin statement, End statement* and *Sta:* commands. Get statement also replaces the bind variable place-holders in the copy of the statement it returns, with the normal Omnis syntax for bind variables (“@[...]”).

Executing a SQL Statement

After a SQL statement has been prepared, using the $prepare() method, it can be executed. This is done using the $execute() method. For example

```
Do StatementObj.$execute() Returns #F
```

A value of kTrue is returned if the statement was successfully executed, kFalse otherwise indicating that an error occurred. If the statement $state property is kStatementStateClear prior to the execution of this method it will fail. Once a statement has been successfully executed, the value of the $state property is set to kStatementStateExecuted.

If the SQL statement generates results (e.g. a SELECT command), the $resultspending property of the statement is set to kTrue. These results may be retrieved using the $fetch() method.

Once a statement has been executed it may be re-executed as many times as is required using the $execute() method. There is no need to prepare it again unless a different SQL command is required. Re-executing a statement that has $resultspending set to kTrue will clear the results set however.

Alternatively, you can prepare and execute a SQL statement with a single command using the $execdirect() method. For example

```
Do StatementObj.$execdirect('SELECT * FROM authors ORDER BY au_lname,au_fname') Returns #F
```

This method effectively performs a $prepare() followed by a $execute() method and if successful returns a value of kTrue. Once a statement has been successfully executed, the value of the $state property is set to kStatementStateExecDirect. It is not possible to re-execute a statement using $execute that has previously been executed using $execdirect().

Fetching Results

When a SQL statement is executed that returns results (e.g. a SELECT statement), these can be retrieved using the $fetch() method in the form

```
Do StatementObj.$fetch(pTableRef,pRowCount,pAppend) Returns lFetchStatus
```

The results are placed in pTableRef, which may be a list or row variable. The pTableRef parameter may be omitted only if a previous $fetch() defined the row or list to use for returning data.

The pRowCount is a positive integer used to specify the number of results rows to return. It can also be set to kFetchAll to signal that all remaining rows in the result set should be returned. When pTableRef is a list, the number of rows returned is pRowCount or less. If pTableRef is a row variable, $fetch() always returns 1 row. If pRowCount is greater than the number of rows in the current result set, only the available rows are returned. If there are no more rows left in the current set, kFetchFinished is returned, otherwise kFetchOK is returned. If the pAppend parameter has a value of kTrue, the rows returned are appended to the list, or if kFalse, replace the previous contents of the list.

If the row or list is undefined, its column definition is created based upon the names of the columns in the result set and the results are retrieved. This can be forced by defining the variable as empty prior to fetching any rows. For example

```
Do iResultsList.$define()
Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus
```

If the variable was previously defined and the definition of the list or row columns do not match the data returned, any valid conversions are performed. If there are no more results pending, the $resultspending property of the statement is set to kFalse. An attempt to fetch when $resultspending is kFalse will result in a return status of kFetchError.

To limit the number of rows returned to a list use set the value of pRowCount to a value that will not make the list too big. For example

```
Do StatementObj.$fetch(iResultsList,100,kTrue) Returns lFetchStatus
```

The number of rows to be returned is in this case up to 100. If there were more than 100 rows in the results set, the extra rows will remain waiting to be fetched. The pAppend parameter is set to kTrue indicating that the results rows are to be added to the end of the list, preserving existing rows.

You can use the $linemax list property to limit the size of the list regardless of the number of rows in the results set. For example to limit the size of a list to 50 rows

```
Calculate iResultsList.$linemax as 50
Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus
```

Any results generated by a statement will be available until either they have all been fetched or another query is executed using the same statement object, in which case the previous results set is destroyed.

To limit the result set based on memory rather than the list size, you can use the statement object's $maxresultsetsize property which defaults to 100MB. $fetch() stops fetching when this limit is exceeded and lFetchStatus will be set to kFetchMemoryUsageExceeded. To remove any memory limit, set $maxresultsetsize to zero (Studio 10.2 and later).

Fetching directly into Omnis Variables

To return a row of data without using a list or row variable, the $fetchinto() method can be used in the form

```
Do StatementObj.$fetchinto(vVar1, vVar2,… vVarN) Returns lFetchStatus
```

$fetchinto() returns a row of data directly into the parameters supplied. A variable number of parameters can be supplied corresponding to each column in the result set .

Fetching into an External File

The $fetchtofile() method lets you fetch a results set to an external file. It is the object DAM equivalent of the old-style command *Retrieve rows to file*, allowing one or more rows from a result set to be written directly into an external text file (export file).

$fetchtofile() implicitly opens and closes the export file, so also encompasses the old-style commands: *Set client import file name, Open client import file, Close client import file* and *Delete client import file*.

As with the $fetch() and $fetchinto() methods, a result set must exist before you call $fetchtofile(). The syntax used for $fetchtofile() is:

```
Do StatementObj.$fetchtofile(cFilename [,iRowCount=1][,bAppend=kTrue]    [,bColumnNames=kTrue][,iEncoding=k
```

The parameters are:

- **cFilename**
  is the full path of the export file, formatted as required by the curremt operating system. The filename will usually be given the extension ".txt"
- **iRowCount**
  is an optional parameter which specifies the number of rows to write to the export file. If iRowCount is less than the number of rows in the result set, $fetchtofile () can be executed again in which case the result set will be fetched from the last unread row. To retrieve all rows in the result set to the file, specify kFetchAll. If omitted, a single row is written.
- **bAppend**
  is an optional parameter which specifies whether the existing file contents should be appended to or over-written. The default is to append the data (kTrue).
- **bColumnNames**
  is an optional parameter which specifies that the first row of the file should contain the column names of the result set. This has no effect when bAppend is set to kTrue. The default behavior will automatically write column names to a file which is empty or being overwritten.
- **iEncoding**
  is an optional parameter which specifies the type of encoding to be used. iEncoding should be one of the Unicode type constants and defaults to kUniTypeUTF8. The corresponding Unicode byte order marker (BOM) is written to the beginning of the file when the file is empty or when bAppend is set to kFalse.

$fetchtofile() automatically creates the export file if it does not exist or otherwise opens the file prior to writing the pending result rows into it. The file is closed immediately on writing the last row.

$fetchtofile() will return kFetchOk or kFetchFinished if the requested rows are successfully written to the export file, otherwise kFetchError is returned with the resulting error message being written to statement.$errorcode and statement.$errortext.

Two additional statement properties have been added which work in conjunction with the $fetchtofile() method. These are $columndelimiter and $rowdelimiter which are used to specify the characters that delimit columns and rows. It may be necessary to change these properties to match the format expected by an external application. $columndelimiter and $rowdelimiter accept multiple characters if required (15 characters maximum). The default value for $columndelimiter is kTab (chr(9)), and $rowdelimiter is kCr (chr(13)).

Batch Fetching

Where the DBMS is capable, it is possible to reduce network traffic when fetching multiple rows of data from the database by fetching several rows simultaneously (as a single *batch*), thus reducing the number of fetches required to return the entire result set.

There are three statement properties to manage batch fetching; $batchsize, $effectivebatchsize and $maxbuffersize.

By default $batchsize is set to 1, but this can be set to any long integer value. $batchsize should be set before the SQL SELECT statement is prepared. Changing the $batchsize automatically clears any results which may be pending on the statement object. When increasing the batch size, it should be noted that the memory resources required to receive the row set will increase proportionately.

$maxbuffersize can be used to limit the batch size such that for a given column of result data, the buffer size calculated as columnSize * batchSize will not exceed the imposed maximum value. If the maximum buffer size would be exceeded, the batch size is revised downward at prepare time to accommodate the maximum number of result rows per fetch. The default value of $maxbuffersize is 32KB but will accept any long integer value greater than 255.

The resulting batch size is assigned to the $effectivebatchsize property (which is read-only).
The value assigned to $batchsize is stored so that when the statement object is re-used, $effectivebatchsize can be calculated again.

It is not possible to return non-scalar data types using batch fetching. Such types include Oracle CLOBs and BLOBs which are not stored in the database directly but referenced by pointers (LOB locators). Thus, the raw data for each column value has to be fetched using a separate network transaction- negating the potential benefit of using batch fetching.

Batch fetching of large binary data is also prohibitively expensive in terms of memory requirements; only one or two rows could be fetched in most cases.

For similar reasons, batch fetching is also not supported for column data which is retrievable via chunks (chunking). If the database supports chunking of fetched data, the session property $lobthreshold should be set high enough to prevent possible chunking of any of the result columns.

When preparing a select statement which includes one or more non-scalar or chunk-able columns, $batchsize will be set back to 1 automatically. For example

```
Do mylist.$define()
Do tStatement.$batchsize.$assign(3000) Returns #F ## Not implemented by all servers
Do tStatement.$prepare('select * from mytable') Returns #F
OK message {$batchsize will be limited to [tStatement.$effectivebatchsize] for this result set}
Do tStatement.$execute() Returns #F

Do tStatement.$fetch(mylist,kFetchAll) Returns lStatus
```

The following object DAMs currently support batch fetching: DAMODBC, DAMORA8, DAMSYBSE & DAMDB2. For database servers which do not support batch fetching, the $batchsize and $maxbuffersize properties are also read-only.

Debugging Slow Queries

The **$trackslowqueries** allows you to track and debug slow queries. The value of $trackslowqueries represents the number of seconds that an EXECUTE or FETCH from the database has to reach before being considered slow. The default is 0 which means the query tracking is off.

For example, if the value of $trackslowqueries is 2 and a query takes 3347 milliseconds to finish, the query will be reported to the trace log alongside its execution time. If $debuglevel is set to 2 and a $debugfile is set, the slow query will be reported in both trace log and file specified in $debugfile.

Part of the SQL executed will be included in the message, up to 80 characters, in order to keep the logs clean; this should be enough to identify the query in the code.

Measuring Data Transfer

You can measure the amount of data (in bytes) that is received and sent through a session object since logon using the session properties $bytesreceived and $bytessent. The values can be reset by assigning zero to them. These properties apply to all DAMs.

Describing Results

After executing a SQL statement, you can use the $results() method to obtain information about the columns contained in the current results set.

The only parameter required is a list variable where the information is to be placed. For example

```
Do StatementObj.$results(iResultsList) Returns #F
```

The returned list will be defined with the following columns and will contain one *row* for each *column* of the pending results set.

Col	Name	Meaning
1	ColumnName	Name of column
2	OmnisData typeText	Omnis data type (description)
3	SQLData type	Equivalent standard SQL data type (CHARACTER, NUMBER, DATETIME, ...)
4	Length	Column width (for character columns)
5	Scale	Number of decimal places (for numeric cols), empty for floating numbers

Col	Name	Meaning
6	Null	Nulls allowed (kTrue or kFalse)

Substituting Variables into SQL Commands

You can substitute variables into the SQL statement using concatenation. You must supply quoted literals for character columns according to the rules imposed by your SQL server. For example

```
Do StatementObj.$prepare(con("SELECT * FROM authors WHERE state='",iState,"' ORDER BY au_lname,au_fname"))
```

Note the use of double-quotes around the statement string, since it is necessary to embed single quotes around the value for the character column.

If you are using the *Sta:* command to build up a statement, a variable may be substituted using square bracket notation. For example,

```
Begin statement
  Sta: SELECT * FROM authors
  If iState<>''
    Sta: WHERE state='[iState]'
  End if
  Sta: ORDER BY au_lname,au_fname
End statement

Do StatementObj.$prepare() Returns #F
```

Note that using these types of substitution you can only substitute variables containing data that can be represented by characters (i.e. character and numeric variables.) For other types of data including binary, pictures and lists you must use bind variables.

Bind Variables

A *bind variable* allows an Omnis variable to be passed to a SQL statement. Instead of expanding the expression, Omnis associates, or binds the memory address of the variable to a SQL variable. To specify a bind variable, you place an @ before the opening square bracket. Omnis evaluates the expression and passes the value to the server directly rather than substituting it into the SQL statement as text. You can also use this syntax to bind large fields such as binary, pictures and lists into a SQL statement. For example

```
Do StatementObj.$prepare('SELECT * FROM authors WHERE state = @[iState] ORDER BY au_lname,au_fname') Return
# or
Do StatementObj.$prepare('INSERT INTO authors (au_lname,au_fname) VALUES (@[iRow.au_lname],@[iRow.au_fname]
```

Alternatively using the *Sta:* command:

```
Begin statement
  Sta: SELECT * FROM authors
  If iState<>''
    Sta: WHERE state=@[iState]
  End if
  Sta: ORDER BY au_lname,au_fname
End statement

Do StatementObj.$prepare() Returns #F
```

Note that you do not need to place quotes around a value to be substituted by a bind variable. You must include quotes when using square bracket notation to substitute character variables, but you don't need to when using bind variables.

Not all database systems allow bind variables; in these cases, Omnis will behave as though they do, but will instead perform literal expansion as though you had entered square bracket notation instead of bind variables

Generally, using bind variables results in better performance than using square bracket notation and is more flexible with respect to data representation. You should use square bracket notation only when the notation expression evaluates to a part of a SQL statement other than a value reference- such as an entire WHERE clause, or where you know that simple value substitution is all you need.

If you are inserting NULL data into the database, you should use bind variables, to ensure that SQL nulls, rather than empty strings are inserted.

Be careful to ensure when using a variable such as a local or instance variable, that it will still exist at the time when the SQL is executed using the $execute() method. This is not usually a problem when the $execute() method follows immediately after the $prepare, but may be if the $execute() method is located elsewhere (i.e. in another method.) It does not matter if an instance variable is not in scope in the method where the $execute() is located so long as the variable has not been destroyed by closing the instance that owns it.

Once a statement containing bind variables has been prepared, it can be repeatedly executed using the $execute() method. The values of the bound variables may be changed before each call to $execute(), allowing different data to be passed each time. This can greatly speed up the process of, say, inserting many rows into a server table within a loop.

Constructing SQL Queries from Row Variables

Several methods are provided that allow you to construct SQL queries based on the definition of the columns of a row or list variable. These methods return partially complete SQL text that varies according to the session object in use and always includes all columns from the row or list variable.

Pre-V30 SQL Functions

Existing users should note: The Pre-V30 SQL functions insertnames(), selectnames(), updatenames(), and wherenames() have been removed from the Catalog (F9/Cmnd-9) in Omnis Studio 6.x onwards, but they can still be used in your code. In addition, the createnames() and server() functions have been removed from Omnis, including the Catalog, and will no longer work in code in Omnis Studio 6.x onwards.

$createnames()

To create a new table on the server, you can use the $createnames() method. $createnames() returns a text string which is a comma delimited list of column names and data types. Use is of the form:

```
Calculate lColList as SessObj.$createnames(pRowRef)
```

Parameter pRowRef is a row or list variable.

This can be used in a SQL CREATE TABLE statement, for example to create a new database table called “publishers” based on the definition of the columns in row variable “iPublishersRow”

```
Do StatementObj.$execdirect(con('CREATE TABLE publishers (', SessObj.$createnames(iPublishersRow),')')) Ret
```

Note that you need to include the parentheses around the column list because they are not provided by the method.

Depending on the session object, this will create a SQL command similar to

```
CREATE TABLE publishers (pub_id varchar(4),pub_name varchar(40),city varchar(20),state varchar(2),country v
```

$insertnames()

To insert a row into a table, you can use the $insertnames() method. $insertnames() returns a text string that is a comma delimited list of column names and values to be inserted. Use is of the form:

```
Calculate lColList as SessObj.$insertnames(pRowRef)
```

Parameter pRowRef is a row or list variable. If it is a list variable, the values from the current line (pRowRef.$line) are used.

This can be used in a SQL INSERT statement. For example, to insert a row into the table “publishers” containing data from the row variable “iPublishersRow”:

```
Do StatementObj.$execdirect(con('INSERT INTO publishers ', SessObj.$insertnames(iPublishersRow))) Returns #
```

Note that the method provides the parentheses around the column and values lists and the “VALUES” clause automatically.

This will create a SQL command similar to

```
INSERT INTO publishers (
    pub_id,pub_name,city,state,country)
    VALUES (@[iPublishersRow.pub_id],
    @[iPublishersRow.pub_name],
    @[iPublishersRow.city],
    @[iPublishersRow.state],
    @[iPublishersRow.country])
```

$updatenames()

To update a row in a table, you can use the $updatenames() method. $updatenames() returns a text string that is a comma delimited list of column names and values to be updated. Use is of the form:

```
Calculate lColList as SessObj.$updatenames(pRowRef)
```

Parameter pRowRef is a row or list variable. If it is a list variable, the values from the current line are used.

This can be used in a SQL UPDATE statement. For example, to update a row in the table “publishers” containing data from the row variable “iPublishersRow”:

```
Do StatementObj.$execdirect(con('UPDATE publishers ', SessObj.$updatenames(iPublishersRow),' WHERE pub_id =
```

Note that the method provides the “SET” clause automatically.

This will create a SQL command similar to

```
UPDATE publishers SET pub_id=@[iPublishersRow.pub_id],
    pub_name=@[iPublishersRow.pub_name], city=@[iPublishersRow.city],
    state=@[iPublishersRow.state], country=@[iPublishersRow.country]
    WHERE pub_id = @[iPublishersRow.pub_id]
```

For a list with a table instance, there are some additional parameters that allow the Where clause to be omitted and the updated columns to be determined from the data that has changed. In this case, the definition for $updatenames() is:

- **list.$updatenames**([cOldrowName] [,cRowName=”, **bExcludeWhere**=kFalse, **wOldrow**=#NULL, **wRow**=#NULL])

Pass bExcludeWhere as kTrue to exclude the Where clause. This is false by default. If wOldRow is supplied, then wRow can also be supplied, or if not, the current line of the list will be used (so if wRow is omitted and there is no current line, $updatenames fails and returns #NULL).

When wOldRow is supplied, in addition to the usual behavior of omitting columns marked as $excludefromupdate, $updatenames also excludes columns where the value in wOldRow equals the value in wRow (or the current list line if wRow is not supplied).

$selectnames()

To select all columns in a table, you can use the $selectnames() method. $selectnames() returns a text string that is a comma delimited list of column names to be selected. Use is of the form:

```
Calculate lColList as SessObj.$selectnames(pRowRef,pTableName)
```

Parameter pRowRef is a row or list variable and pTableName is an optional parameter (default empty) that will prefix column names with the specified table name.

This can be used in a SQL SELECT statement. For example, to select all columns in the table “publishers” using the column definition of the list variable “iPublishersList”:

```
Do StatementObj.$execdirect(con('SELECT ', SessObj.$selectnames(iPublishersList),' FROM publishers ORDER BY
```

This will create a SQL command similar to

```
SELECT pub_id,pub_name,city,state,country FROM publishers ORDER BY pub_name
```

$wherenames()

To locate a row or rows in a table to be updated or deleted you can use the $wherenames() method. $wherenames() returns a text string that is a comma delimited list of column names and values. Use is of the form:

```
Calculate lColList as SessObj.$wherenames(pRowRef,pTableName,pComparison,pOperator)
```

Parameter pRowRef is a row or list variable. If it is a list variable, the values from the current line are used. The remaining parameters are optional, pTableName will prefix column names with the specified table name, pComparison (default "=") is used to specify an alternative comparison operator (e.g. >, <, >= etc.), pOperator (default "AND") is used to specify an alternative logical operator (i.e. OR.)

This can be used in a SQL UPDATE or DELETE statement. For example, to delete a row in the table "publishers" where the column values exactly match all columns in the row variable "iPublishersRow":

```
Do StatementObj.$execdirect(con('DELETE FROM publishers ', SessObj.$wherenames(iPublishersRow))) Returns #F
```

This will create a SQL command containing a WHERE clause similar to

```
DELETE FROM publishers WHERE pub_id=@[iPublishersRow.pub_id]
    AND pub_name=@[iPublishersRow.pub_name]
    AND city=@[iPublishersRow.city]
    AND state=@[iPublishersRow.state]
    AND country=@[iPublishersRow.country]
```

Table and Column names

Table instances and session variables have a property called $quotedidentifier which determines whether or not table and column names are contained in quotes. If set to kTrue, table and column name identifiers returned from the $createnames(), $insertnames(), $updatenames(), $selectnames() and $wherenames() methods will be quoted "thus", facilitating case-sensitive names and names containing spaces. The new property affects table instance methods as well as session object methods.

SQL Errors

It is possible for an error to occur when a session or statement method is executed. The error may originate from the session object or in the database. The session or statement method will indicate that an error has occurred by returning the value kFalse. The value returned by these methods may be placed into any variable. The following example places the return value in the Omnis flag variable:

```
Do StatementObj.$execute() Returns #F
```

The methods will also return a generic (i.e. database independent) code and message describing the error in the properties $errorcode and $errortext. For example, in the event of an error occurring, SessObj.$logon() would return an error code in the property SessObj.$errorcode and an error message in SessObj.$errortext whilst StatementObj.$execute() would return an error code in StatementObj.$errorcode and message in StatementObj.$errortext. You can create a method to display the error code and message. For example:

```
OK message SQL Error {Code = [StatementObj.$errorcode], Msg = [StatementObj.$errortext]}
```

If smart lists are used, the history list row properties $errorcode and $errortext contain the value of StatementObj.$errorcode and StatementObj.$errortext for each row updated.

Additionally, you can obtain the native error code and message from the server using the session and statement properties $nativeerrorcode and $nativeerrortext. Some servers may return more than one combination of native error code and message for a single error. Currently only Sybase and Microsoft SQL Server behave in this way. When an error occurs the $errorcode and $errortext will contain a generic error from the session object and the $nativeerrorcode and $nativeerrortext will contain the first error reported by the server. If there are further error messages, $nativeerrorpending (a Boolean property of the statement object) will be set the kTrue. In order to access the next native error code and message, use the $nextnativeerror() method that will set the $nativeerrorcode and $nativeerrortext properties to the next error. You can repeatedly call the $nextnativeerror() method until all errors have been processed ($nativeerrorpending=kFalse). For example

```
While StatementObj.$nativeerrorpending
 Do StatementObj.$nextnativeerror()
 OK message SQL Error {Code = StatementObj.$nativeerrorcode], Msg = [StatementObj.$nativeerrortext]}]}
End While
```

All server errors cause the statement method to return a value of kFalse. If you get an error from a method it does not prevent execution of further methods. You should always test the return value after execution of a session or statement method and take an appropriate action.

Data Type Mapping

Omnis converts the data in an Omnis field or variable into the corresponding SQL data type. Since each DAM maps to a wide variety of data types on servers, each DAM determines the correct conversion for the current server. See the Server-Specific Programming chapter for details on how each DAM maps SQL data types to Omnis data types and vice versa.

DAMs support for 64-bit Integers

The Studio 6.0 DAMs perform additional data type mappings between Omnis 64-bit integers and the corresponding large integer data type on the database server. For most databases this will be BIGINT. The notable exception is Oracle which uses NUMBER(19,0) instead.

Note also that BIGINT UNSIGNED columns will be converted to signed 64-bit Integers when fetched into Omnis. In order to preserve such values, a *CAST(column as CHAR)* function can be used to fetch the value into a character field.

Where schemas and lists are defined using Integer 64-bit columns, the session object's $coltext() and $createnames() methods now return the appropriate SQL data type. Integer 32-bit columns retain their previous behavior.

Clearing Statements and Sessions

It is possible to clear individual statement objects back to their default state using the $clear() method. For example

```
Do StatementObj.$clear() Returns #F
```

When used with a statement, the $clear() method will cancel any pending results or operations on the statement object and set all properties to their default values (except for $statementname, $usecursor.) The statement object is placed in the kStatementStateClear state. If the statement is using a cursor, it will be closed.

When used with a session object the $clear() method will clear all statements controlled by the session object instance. For example:

```
Do SessObj.$clear() Returns #F
```

All of the statement objects are placed in a kStatementStateClear state. Depending on the session state this will clear all SQL text, close all cursors and destroy pending results of all statement objects owned by the session.

Listing Database Objects

Omnis provides statement methods that enable you to access data dictionary information about any database to which you can connect using a session object. Using these methods, you can create database-independent code to list objects contained in your server database, regardless of its type. These methods work by creating results sets as though you had queried the information from the database. You then use the $fetch() method to read the results into Omnis.

Listing Tables and Views

The $tables() method generates a results set containing details of tables available to the statement object. Use is of the form:

```
Do StatementObj.$tables(pType,pOwner) Returns #F
```

The parameter pType is used to indicate what types of object are to be listed and may contain kStatementServerTable to obtain details of all tables, kStatementServerView to obtain details of all views, or kStatementServerAll (default) to obtain details of all tables and all views. The parameter pOwner is used to list only objects belonging to a single named owner and defaults to all owners. For example, to create a list of all available tables and views for all owners:

```
Do StatementObj.$tables() Returns #F
Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus
```

To create a list of all available views owned by DBO:

```
Do StatementObj.$tables(kStatementServerView,'DBO') Returns #F
Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus
```

The result set contains the following columns:

Col	Name	Description
1	Owner	Name of user that owns the database object
2	TableOrViewName	Name of table or view
3	TableType	Object type (kStatementServerTable or kStatementServerView)
4	Description	Remarks or description for the object where available
5	DamInfoRow	A row of database specific information about the table or view. This may be empty for some session objects

Listing Columns

The $columns() method generates a results set containing details of columns for a specified table or view.

The only parameter required is the name of the database table or view for which column details are required. For example, to create a list of columns in the “authors” table:

```
Do StatementObj.$columns('authors') Returns #F
Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus
```

The results set contains the following columns:

Col	Name	Description
1	DatabaseOrCatalog	Name of database or catalog that contains the object
2	Owner	Name of user that owns the database object
3	ColumnName	Name of column
4	OmnisData typeText	Omnis data type (description)
5	OmnisData type	Omnis data type (notational)
6	OmnisDataSubType	Omnis data subtype (notational)
7	SQLData type	Equivalent standard SQL data type (CHARACTER, NUMBER, DATETIME, ...)
8	Length	Column width (for character columns only*)
9	Scale	Number of decimal places (for numeric cols), empty for floating numbers
10	Null	Nulls allowed (kTrue or kFalse)
11	Index	Index exists for column (kTrue or kFalse)
12	PrimaryKey	Column is the primary key (kTrue or kFalse)
13	Description	Remarks or description for the column where available
14	DamInfoRow	A row of database specific information about the column. This may be empty for some session objects

*As of Studio 6.1, $columns() accepts an additional *flags* parameter which can be used to return lengths for other column types. Refer to Statement Methods for details.

Listing Indexes

The $indexes() method generates a results set containing details of index columns for a specified table. Use is of the form:

```
Do StatementObj.$indexes(pTableName,pType) Returns #F
```

The parameter pTableName is the table of which indexes are to be listed. The parameter pType is used to indicate what types of indexes are to be listed and may contain kStatementIndexUnique (default) to obtain details of unique indexes, kStatementIndexNonUnique to obtain details of non-unique indexes, or kStatementIndexAll to obtain details of all indexes. For example, to create a list of all indexes for the table "authors":

```
Do StatementObj.$indexes('authors',kStatementIndexAll) Returns #F
Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus
```

The results set contains the following columns:

Col	Name	Description
1	DatabaseOrCatalog	Name of database or catalog that contains the object
2	Owner	Name of user that owns the database object
3	ColumnName	Name of column contained in index
4	IndexName	Name of index
5	Unique	Unique index (kTrue or kFalse)
6	ColumnPosition	Position of column (integer) in index (1 for normal index, 1,2,3… for column in compound index)
7	DamInfoRow	A row of database specific information about the index. This may be empty for some session objects

Building Schema Classes

Using the $makeschema() session method you can make a schema class automatically that matches the columns in a database table using a command of the form:

```
Do SessObj.$makeschema(pSchema,pTableName) Returns #F
```

The parameter pSchema is a reference to an existing schema class that will be overwritten with the definition from the server table pTableName.

For example, to create a schema called "scAuthors" from the server table "authors":

```
Do $schemas.$add('scAuthors') Returns #F
Do SessObj.$makeschema($schemas.scAuthors,'authors') Returns #F
```

Using the $tables() and $makeschema() methods you can obtain a list of tables on the server and build a schema class for each server table.

Defining Lists from Server Tables

As of Studio 8.1.5, the session method; $definelistorrow() can be used to define a list or row variable directly from a named server table, i.e. without the requirement for a schema class. For example:

```
Do SessObj.$definelistorrow(iList1,'logon_names') Returns #F
```

Inside a table instance you can also pass $cinst as the list/row name. If the server table contains a primary key, $definelistorrow sets $excludefromwhere to kTrue for non-primary key columns.

Remote Procedures

Omnis provides methods that enable you to list and execute remote procedures that exist in the database. Support for these methods varies from one database to another, see the Server-Specific Programming chapter for more details.

Listing Remote Procedures

The $rpcprocedures() method generates a results set containing details of remote procedures available to the statement object. Use is of the form:

```
Do StatementObj.$rpcprocedures(pOwner) Returns #F
```

The optional parameter pOwner is used to list only objects belonging to a single named owner and defaults to all owners. For example to create a list of all available remote procedures for all owners

```
Do StatementObj.$rpcprocedures() Returns #F
Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus
```

The results set contains the following columns:

Col	Name	Description
1	DatabaseOrCatalog	Name of database or catalog that contains the object
2	Owner	Name of user that owns the database object
3	ProcedureName	Name of remote procedure
4	DamInfoRow	A row of database specific information about the remote procedure. This may be empty for some session objects

Listing Remote Procedure Parameters

The $rpcparameters() method generates a results set containing details of all parameters required by a particular remote procedure. Use is of the form:

```
Do StatementObj.$rpcparameters(pProcedureName) Returns #F
```

The parameter pProcedureName is the name of the remote procedure. This parameter may take the form Database.Owner.Procedure name depending on whether database and owner qualifiers are supported. The Database and Owner are optional. The use of the qualifier is as follows.

Only Procedure name specified	Return parameter information for all procedures with specified Procedure name in all available databases.
Owner.Procedure name specified	Return parameter information for all procedures with specified Procedure name owned by Owner in all available databases.
Database.Owner.Procedure name specified	Return parameter information for specified procedure owned by Owner in Database

For example, to create a list of all parameters for remote procedure “byroyalty”:

```
Do StatementObj.$rpcparameters('byroyalty') Returns #F
Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus
```

The results set contains the following columns:

Col	Name	Description
1	OmnisData type	Omnis data type (notational)
2	OmnisDataSubType	Omnis data subtype (notational)
3	Length	Column width (for character columns)
4	PassType	How the parameter is used. One of the constants kParameterInput, kParameterOutput, kParameterInputOutput, or kParameterReturnValue.

Col	Name	Description
5	C5	Reserved for future use.
6	C6	Reserved for future use
7	DatabaseOrCatalog	Name of database or catalog that contains the object.
8	Owner	Name of user that owns the database object.
9	ParameterName	Name of the parameter.
10	OmnisData typeText	Omnis data type (description)
11	SQLData type	Equivalent standard SQL data type (CHARACTER, NUMBER, DATETIME, …)
12	Scale	Number of decimal places (for numeric cols), empty for floating numbers
13	DamInfoRow	A row of database specific information about the parameter. This may be empty for some session objects

Note that columns 1, 2 and 3 are the closest Omnis type to the server type specified in the server definition of the procedure parameter. |

Calling a Remote Procedure

Before you can call a remote procedure, it must first be registered with Omnis using the session method $rpcdefine().Use is of the form:

```
Do SessObj.$rpcdefine(pProcedureName,pList) Returns #F
```

The parameter pProcedureName is the case-sensitive name of the remote procedure that must exist on the server. If the procedure has previously been defined, the new definition replaces the old one. The parameter pList defines the parameters and the return value of the remote procedure. The list must have the same layout as that returned by $rpcparameters(pProcedureName), except that only the first 4 columns are required. See the section on Listing Remote Procedure Parameters ($rpcparameters) for details of the list layout.

The easiest way to define a procedure is to first call $rpcparameters(), fetch the result set into a list, and pass the list to $rpcdefine(). For example:

```
Do iResultsList.$define()
Do StatementObj.$rpcparameters('byroyalty') Returns #F
Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus
Do SessObj.$rpcdefine('byroyalty',iResultsList) Returns #F
```

Once a remote procedure has been defined, it can be invoked using the statement method $rpc().Use is of the form:

```
Do StatementObj.$rpc(pProcedureName,pParam1,…pParamN) Returns #F
```

The call to $rpc() will fail if pProcedureName has not been defined using $rpcdefine() or does not exist on the server. The session object will invoke the specified remote procedure, using the procedure definition to determine the parameters it needs. If the optional parameters pParam1…pParamN are included, they are passed to the stored procedure.

If the call is successful, any output parameter values are returned. If the procedure has a return value specified in its definition, it is written to the statement property $rpcreturnvalue, if not, $rpcreturnvalue is Null. If the call to the procedure generates a result set, the statement property $resultspending is set to kTrue and these results may be retrieved using the $fetch() method. Following successful execution of the remote procedure, the statement $state property will be set to kStatementStateExecDirect. If the state prior to this call was not kStatementStateClear, any pending result set or unexecuted SQL statement is cleared.

For example, to invoke the stored procedure “byroyalty” passing the variable lPercentage to the 1st parameter and fetch the results set generated by the stored procedure:

```
Do StatementObj.$rpc('byroyalty',lPercentage) Returns #F
Do StatementObj.$fetch(iResultsList,9999) Returns lFetchStatus
```

Transactions

The $transactionmode session property controls the way that transactions are managed. Depending on the value of this property the session object may automatically manage transactions, or it may be necessary to manage transaction using explicit method calls or SQL statements.

Some servers do not provide support for transactions. You can determine whether a particular server allows transactions using the read-only Boolean session property $allowstransactions which contains kTrue if the server supports transactions or kFalse otherwise. Some session objects will contain a value of kUnknown in this property until the session is logged on. If your server does not allow transactions, the properties and methods described in the section on Transaction Modes below should not be used.

Transaction Modes

You can set the $transactionmode session property using a command of the form:

```
Do SessObj.$transactionmode.$assign(kSessionTranManual) Returns #F
```

The potential values for the transaction mode are

- **kSessionTranAutomatic**
  This is the default and specifies that all transaction management be provided automatically.
- **kSessionTranManual**
  Enables the application to manage transactions manually using the session methods $begin(), $commit() and $rollback().
- **kSessionTranServer**
  Transaction management is provided by the DBMS.

Automatic Mode

After a SQL statement has successfully executed, i.e. $execute() or $execdirect() returns kTrue, the current transaction is automatically committed by the session object. If the command fails, the transaction is rolled back automatically. A new transaction is started automatically if required after a successful commit or rollback.

Note that since each individual SQL command is committed immediately that it is executed, automatic mode does not allow for the creation of transactions that contain a number of SQL commands to be individually prepared and executed prior to a single commit. If this is required, you should use Manual or Server mode.

Manual Mode

In kSessionTranManual mode you manage transactions manually using session methods.

$begin()

Where required by the session object, use the $begin()method to start a new transaction, for example:

```
Do SessObj.$begin() Returns #F
```

The $begin() method should only be executed where the DBMS does not implicitly start a transaction. The read-only session property $autobegintran will contain the value kTrue to indicate for a particular session object that a transaction is automatically started when a connection is established to the database or the previous transaction is committed or rolled back, and in this case the $begin() method should not be used. This method may fail if there is a current transaction or the server does not support nested transactions.

$commit()

```
Do SessObj.$commit() Returns #F
```

The $commit() method will fail if the session $state property is kSessionLoggedOff, if the transaction mode is not kSessionTranManual or if there is no current transaction. With certain types of session objects this will commit and clear all statements, close any cursors used by a statement and clear pending results. This will not destroy the statement objects used by a session. Depending on the value of the session property $autobegintran, the server may begin a new transaction automatically.

$rollback()

```
Do SessObj.$rollback() Returns #F
```

The $rollback() method will fail if the $state property is kSessionLoggedOff, if the transaction mode is not kSessionTranManual or if there is no current transaction. This method cancels the current transaction. With certain types of session objects this will rollback and clear all statements, close any cursors used by a statement and clear pending results. This will not destroy the statement objects used by a session. Depending on the value of the session property $autobegintran, the server may begin a new transaction automatically.

The read-only session properties $commitmode and $rollbackmode describe the effect that the $commit() and $rollback() methods have on statements and their cursors.

kSessionCommitDelete kSessionRollbackDelete	Prepared statements and cursors on all statement objects are deleted. Any pending re lost. The statement object itself is not deleted but will be set to a kStateClear state. To r the same statement it must first be re-prepared.
kSessionCommitClosekSessionRollbackClose	Prepared statements and cursors on all statement objects are closed. Any pending res A statement can be re-executed without first being re-prepared. Any statement object successfully prepared a statement will be in the kStatePrepared state.
kSessionCommitPreservekSessionRollbackPreserve	The state of all statements and cursors remains unchanged.

Note that with some session objects these properties may change once a session is logged on.

Server Mode

Transaction management is provided by the DBMS. The default behavior is determined by the database, which may for example automatically commit each SQL statement unless you override the default.

You may also execute SQL BEGIN, COMMIT and ROLLBACK or other statements depending on the DBMS SQL dialect, to manage transactions manually.

The read-only session property $autobegintran will contain the value kTrue to indicate for a particular session object that a transaction is automatically started when a connection is established to the database, or after a SQL COMMIT or ROLLBACK statement. If $autobegintran is kFalse, an explicit SQL BEGIN statement is required. SQL based transaction commands should not be used other than in kSessionTranServer mode.

As a general rule it is recommended that either automatic or manual mode should be used in preference to server mode.

In kSessionTranManual or kSessionTranServer mode the behavior of the DBMS dictates whether closing a connection commits the current transaction.

The effect of a commit or rollback on existing statement cursors is dependant on the behavior of the DBMS. In most cases, a commit or rollback will close all cursors in the session and clear all results sets. This does not destroy the statement object. It may be possible to re-execute the statement but generally the statement will need to be prepared again.

Care should be taken to note the circumstances in which commits occur as this can have a side effect on the processing of other statement objects associated with the session.

Cursor Results Sets

When a statement is executed that generates results, the results set is preserved until another SQL command is executed. Cursor results sets enable you to process the results of 2 or more different SQL SELECT commands in parallel using a number of statement objects that were created from the same session object using the $newstatement() method.

Note: to use multiple concurrent cursors, the session transaction mode usually needs to be set to kSessionTranManual.

If the database does not implicitly allow for the concurrent processing of multiple results sets, you need to set the statement property $usecursor to a value of kTrue for each statement prior to executing the SELECT. This indicates that a statement should be created via a server-based cursor. It controls the server specific behavior of the $prepare(), $execute(), $execdirect() and $fetch() methods. In some circumstances, the client will automatically generate server-based cursors for SQL SELECT statements and therefore this property is ignored. If the session object manages cursors and the client does not support the manual generation of cursors, the session object will explicitly issue the SQL cursor commands.

Currently, DAMs which support the $usecursor property include the ODBC and Sybase DAMs. The Oracle and OmnisSQL DAMs provide $prepareforupdate() and $posupdate() methods, whilst DB2 provides its own record locking feature.

If the session object has to explicitly issue SQL cursor commands and a statement is prepared when $usecursor is kTrue, the following will be prefixed to the statement:

```
DECLARE <$name> CURSOR FOR <$sqlText>
```

It is important that $sqltext specifies a SQL SELECT statement. Note that the syntax of the DECLARE and associated SELECT command may vary slightly if a particular server does not adhere to the SQL-92 standard.

A subsequent $execute() and $fetch() will issue an

```
OPEN CURSOR <$name> and FETCH CURSOR <$name>
```

Depending on the value of $commitmode and $rollbackmode, any pending results set may be destroyed when a transaction is committed or rolled back.

To ensure that the results sets are not destroyed by an update command you need to set the transaction mode to kSessionTranManual and commit updates manually when ready using the $commit method.

For example, to create a new statement, execute and fetch results from a SELECT command using a cursor:

```
Do SessObj.$newstatement() Returns StatementObj
Do StatementObj.$usecursor.$assign(kTrue)
Do StatementObj.$execdirect('SELECT * FROM authors ORDER BY au_lname,au_fname') Returns #F

Calculate lFetchStatus as StatementObj.$fetch(iResultsList,10)
```

Non-Unicode Compatibility

The DAMs provided with Studio 5.0 are able to function in Unicode or 8-bit compatibility mode. This means that after converting your existing libraries for use with Studio 5.0, it should be possible to continue interacting with non-Unicode databases.

In 8-bit compatibility mode, all DAMs

- Return non-Unicode character data types via the $createnames() and $coltext attributes
- Bind outgoing character variables using the database’s non-Unicode data types
- Convert all data inside outgoing character bind variables to single-byte characters
- Define incoming character columns using the database’s non-Unicode data types
- Convert all data inside incoming character bind variables from bytes into the Omnis character set

Switching to 8-bit compatibility mode

To switch to 8-bit compatibility mode, there is a session property: $unicode- which should be set to kFalse from its default value of kTrue. This implementation allows multiple Unicode and 8-bit session objects to exist side by side if required.

Character Mapping

This section is applicable to session objects operating in 8-bit compatibility mode only.

When reading data from a server database, Omnis expects the character set to be the same as that used in an Omnis data file. The Omnis character set is based on the macOS extended character set, but is standard ASCII up to character code 127. Beyond this value, the data could be in any number of different formats depending on the client software that was used to enter the data.

When assigned, the $maptable session property identifies files containing translation tables for character codes read into and sent out of Omnis. For example, suppose you are working with a database that stores EBCDIC characters. In order to accommodate this database, you should create an ‘.IN’ map file that translates EBCDIC characters to ASCII characters when Omnis in reading server data and a matching ‘.OUT’ file that reverses the process by converting ASCII to EBCDIC characters when Omnis is sending data to the server.

Under Windows and Linux, Omnis uses the same character set as under macOS, so in the general case, mixed platform Omnis applications should have no need for character mapping. However, if the data in a server table was created by another software package, running under Windows for example, the characters past ASCII code 127 would appear incorrect when read using Omnis. In this situation the $maptable property should be used to map the character set.

There are two kinds of character maps: IN and OUT files. IN files are used to translate characters coming from a server database into Omnis. OUT files are used to translate characters that travel from Omnis back to a server database.

The Character Map Editor

The Character map editor is accessed via the Add-On tools menu item and enables you to create character-mapping files. You can change a given character to another character by entering a numeric code for a new character. The column for the Server Character for both .IN and .OUT files may not actually represent what the character is on the server. This column is only provided as a guide. The Numeric value is the true representation in all cases.

To change a character, select a line in the list box and change the numeric code in the Server Code edit box. Once the change has been recorded, press the Update button to update the character map. You can increase/decrease the value in the Server Code edit box by pressing the button with the left and right arrows. Pressing the left arrow decreases the value, pressing the right arrow increases the value.

Make Inverse Map

The File menu lets you create new character map files, save, save as, and so on. The Make Inverse Map option creates the inverse of the current map, that is, it creates an ".IN" file if the current file is an ".OUT" character map, and vice versa. When using the Make Inverse Map option, if any character is defined more than once within the IN.map then only the first value will be translated, and subsequent characters will be set to spaces (dec 32, hex 20). You should check the validity of the results when using the inverse mapping.

Using the Map Files

Establish the character mapping tables by setting the session property $maptable to the path of the two map files. Both files must have the same name but with the extensions .IN and .OUT and be located in the same folder. The $maptable property establishes both .IN and .OUT files at the same time. For example:

```
Do SessObj.$maptable.$assign('C:\Users\My User\Charmaps\pubs') Returns #F
```

In this example, the two map files are called "pubs.in" and "pubs.out".

The session property $charmap controls the mode of character mapping that is to be applied to the data. Set the character mapping mode using a command of the form:

```
Do SessObj.$charmap.$assign(pCharMap) Returns #F
```

The potential values for the character mapping mode parameter pCharMap are

- **kSessionCharMapOmnis**
  Use the internal Omnis character set.
- **kSessionCharMapNative**
  This is the default and specifies that the client machine character set is to be used.
- **kSessionCharMapTable**
  Use the character mapping table specified in the $maptable property. If the $maptable property is not set and the application attempts to assign kSessionCharMapTable this fails.

If you wish to use the character mapping tables defined using the $maptable property, you must set $charmap to kSessionCharMapTable.

Handling Extended Characters

When operating in non-Unicode mode, the session property $codepage determines how 8-bit character codes will be interpreted by the DAM. For example when $codepage is set to kUniTypeAnsiGreek, fetched ANSI extended character codes are interpreted as letters from the Greek alphabet. Conversely, Greek characters inserted from Omnis are mapped to character codes from the Greek code page.
Thus, when operating in non-Unicode mode it is important that the value of $codepage matches with the character set being used by the remote database.

It should also be noted that when inserting data, any Unicode characters which do not correspond with characters in the selected code page will not be mapped correctly and such data is liable to loss or truncation.

Omnis character mapping is applied to fetched character data *before* conversion from the selected codepage, whilst inserted character data has Omnis character mapping applied *after* conversion to the selected code page.

Interpreting 8-bit Data

This section is applicable to the PostgreSQL, MySQL, and Openbase DAMs which interface with their respective client libraries using the UTF-8 encoding .

When operating in Unicode mode, it is possible to receive mixed 8-bit and Unicode data- since UTF-8 character codes 0x00 to 0x7F are identical to ASCII character codes.
Where this data was created using the non-Unicode version of Omnis however, it is possible that the data may contain ASCII extended characters. In this case, the Unicode DAM will encounter decoding errors- mistaking the extended characters as UTF-8 encoded bytes.

This issue was not a concern for the non-Unicode version of Omnis Studio since extended characters were always read and written as bytes- irrespective of the database encoding.

In order to avoid problems when upgrading to the Unicode version of Omnis Studio, it is advisable to convert tables containing ASCII extended characters to UTF-8. This process is simplified where the database character set is already set to UTF-8 (as is often the case with MySQL). All that is required is to read and update each row in the table and repeat this for all tables used by the application. In so doing, Omnis will convert the 8-bit data to Unicode and then write the converted Unicode data back to the database.

In order to facilitate this within the DAM, the session property: $validateutf8 is provided. When set to kTrue (the default), any fetched character data is validated using the rules for UTF-8 encoding. Where a given text buffer fails validation, it is assumed to be non-Unicode data and is interpreted accordingly. When written back to the database, all character data will be converted to UTF-8. Such updates will result in frequently accessed records having their contents refreshed automatically.

By setting $validateutf8 to kFalse, validation is skipped and the DAM reverts to the previous behaviour- in which case extended ASCII characters should be avoided.

Aside from the issue of UTF-8 encoded data, the DAMs provided with Studio 5.0 are able to retrieve non-Unicode data from non-Unicode database columns in either Unicode or 8-bit compatibility mode. The DAM knows the text capabilities of each character data type and assigns encoding values to each result column accordingly.

The difference in behaviour when using 8-bit compatibility is that in compatibility mode, it is also possible to *write* data back to non-Unicode columns.

In Unicode mode, the DAM assumes that it will be writing to Unicode compatible data types and this will cause data insertion/encoding mismatch errors if the clientware tries to insert into non-Unicode database columns.

Character Mapping in Unicode Mode

Character mapping to and from the Omnis character set is also possible where session objects are operating in Unicode mode. This was previously removed from the Unicode DAMs since it provided compatibility between the various 8-bit character sets. Where Unicode DAMs encounter 8-bit data however, the session $codepage property indicates the ANSI code page which should be used to interpret the data.
In the general case, you cannot insert non-Unicode data when the DAM is operating in Unicode mode. To insert such data you should switch to 8-bit compatibility mode (by assigning $unicode to kFalse).

Server Specific Programming

Certain DAMs, namely DAMORA8 and DAMODBC also provide session properties which allow mixing of Unicode and 8-bit data when the DAM is operating in Unicode mode.

The Oracle DAM provides $nationaltonvarchar and $nationaltonclob which allows the Omnis *National* character subtype to be used with Unicode data, whilst the *Character* subtype is reserved for non-Unicode data.

The ODBC DAM provides $nationaltowchar which performs a similar function. These properties are documented further in the Server Specific Programming chapter.

The onus is upon the developer not to put Unicode characters into Character subtypes when using these properties, otherwise data insertion/encoding mismatch errors will occur.

Stripping Spaces

The session object can automatically strip trailing spaces from data returned from the server. This functionality is switched on by setting the statement $sqlstripspaces property to kTrue. The default value for a statement is taken from the session object $sqlstripspaces and the default for the session is read from $clib.$prefs.$sqlstripspaces.

Some data sources may strip trailing spaces prior to sending it to the session object in which case this property has no effect.

Treatment of Date Values

The way the session handles partial, empty and NULL valued date columns can be modified using two properties; sessionObj.$defaultdate and sessionObj.$emptydateisnull.

The $defaultdate property is used to specify default date parts to be used when parts of the date are omitted at bind time. Parts of the date (day, month and/or year) are also substituted where the value being inserted would otherwise constitute an invalid date. This property provides better support for Omnis custom date types, for example DateTime 'H:N:S.s' which may have a corresponding server type which includes the date. Bind variable values override default date values for the constituent date parts which are supplied. The default value for this property is "1 Jan 2000 00:00:00". It is not possible to assign a #NULL value to $defaultdate.

$emptydateisnull can be used to set all outgoing bound dates with an empty value to NULL. This applies to date bind variables used in the WHERE clauses of SELECT, UPDATE and DELETE statements as well as to the input bind variables of INSERT and UPDATE statements. To insert (or test for) empty date values, it is necessary to set $emptydateisnull to kFalse and to assign an empty date to $defaultdate.

The implications of these two properties are summarized below:

To…	Use value…	$defaultdate	$emptydateisnull
INSERT a NULL date into a datetime column orSELECT…WHERE a date value is NULL	#NULLor<empty>	IgnoredIgnored	IgnoredkTrue
INSERT an date value into a datetime column orSELECT…WHERE a date value is	<empty>	<empty>	kFalse
INSERT/test for a date in a datetime column, substituting a default time, (empty datetimes are treated as NULL) or INSERT/test for a Time in a datetime column, substituting a default date, (empty datetimes are treated as NULL).	Datetime value	Defaultdatetime	kTrue
INSERT/test for a Date in a datetime column, substituting a default time, (empty datetimes take on $defaultdate) orINSERT/test for a Time in a datetime column, substituting a default date, (empty datetimes take on $defaultdate).	Datetime value	Default datetime	kFalse

Large Objects

When working with large objects (LOBs) there are some properties that may be used to tune the system so that data is handled and transferred in an efficient way. Large objects include large text, list, binary, picture data and instances of object classes.

Blob Size

The session property $blobsize defines the maximum size of a binary data item. The default value for this property is 10MB, but it may be adjusted to conserve memory. Some session objects may take this value into account when executing a $createnames() method and resulting data types may vary according to the value of $blobsize.

Chunking Large Objects

In order to pass large objects to the server they are broken down into chunks. This is due to potential memory limits that exist with some database vendor client APIs. The session properties $lobchunksize and $lobthreshold represent respectively the size of a single chunk in bytes and the size of object in bytes at which chunking starts. The default value of both of these properties is 32KB.

This only applies to Omnis Character, Picture, Binary, List and Object data types that are greater than 255 bytes.

The value of $lobthreshold may be set to any value between 256 bytes and 2GB. Note that if not set judiciously, this may cause resource problems due to the amount of memory required to cache an object.

The value of $lobchunksize may be set to any value between 256 and the $lobthreshold.

Due to limitations in the database vendor client API, certain session objects may impose their own values for the default and maximum chunk sizes.

Session Pools

Pools of session instances can be created when Omnis starts up and made available to any methods running within Omnis. They are designed to be used by the multi-threaded server and allow client methods to quickly obtain SQL sessions without the burden of constructing their own instances.

There is a notation group $root.$sessionpools that contains the current session pools. Normally the session pools are created by the startup task and exist until Omnis is shut down. Pooled sessions do not appear in the SQL Browser.

When a session is required in order to perform a SQL query, it is obtained from the pool using an object variable and a statement object created in the normal way in order to execute the query and fetch any results. When the session object variable is destroyed the session instance is returned to the pool for later reuse.

Creating a Session Pool

Session pools are created using the $makepool() session object method. The syntax of the method is:

```
$makepool(nam[,count=0,hostname,username,password,initmethod])
# creates a pool of session objects
```

The call to $makepool() will only be successful if used with an external object that is a session object or an object class that has a session object as its subtype, that is, the external server object is its superclass. The method is executed for each session being created before the session is actually logged on.

The $makepool() method returns an item reference to the pool if it is successfully created and the required number of session instances are constructed, otherwise it returns NULL. Once a session pool has been created there is no need to maintain it further during the time that the library is open, however it is possible to change the number of available instances using notation.

Errors encountered when creating session pools are returned via #ERRCODE and #ERRTEXT.

Using $makepool() with an External Object

Create a session pool using an external object with a method call of the form:

```
Do $extobjects.DAMobject.$objects.SessObject.$makepool(pPoolName, pCount,pHostName,pUserName,pPassword) Ret
```

DAMobject is the name of the external component. You can find out what DAM objects are available on your workstation by examining the $root.$extobjects branch of the notation tree using the Notation Inspector. SessObject is the name of the session object. You can find out what session object is available by expanding the $objects group for a particular DAM object using the Notation Inspector. The pPoolName parameter is the name of the pool and must be unique amongst session pools and the pCount parameter is the number of object instances to be initially contained in the pool. The other parameters are optional and if specified are passed to the $logon() method for the instance. If they are not specified, the instance is constructed but not logged on.

For example, to create a session pool called “poolone” containing 5 sessions all logged on to SQL Server

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

Using $makepool() with an Object Class

Alternatively $makepool() may be used with an object class with a method call of the form

```
Do $objects.ObjectClass.$makepool( pName,pCount,pHostname,pUserName,pPassword) Returns lPoolRef
```

ObjectClass is the name of an object class that must have a session object as its superclass. You can achieve this by selecting the object class in the browser and clicking on the $superclass in the Property Manager, click the arrow and select External Objects and double-click on the required session object.

For example:

```
Do $objects.odbcobj.$makepool('pooltwo',10,lHostName,lUserName,lPassword) Returns lPoolRef
```

Initialising session objects

The makepool() method has a sixth parameter that allows you to pass the name of an initialisation method to the session object in the form 'class.method'. The initialisation method needs to have a parameter at position 1 of Object Reference type. This method is called for each session added to the pool. The Object Reference Parameter will contain a reference to the newly created session which can be used to initialise the session object. Example:

```
Do $extobjects.MYSQLDAM.$objects.MYSQLSESS.$makepool('pool1', 5,'192.168.1.25', 'user1', 'mypass', 'NewWind
```

Obtaining a Session Instance From a Pool

The $new() method is used to assign a session instance from a pool. For example

```
Calculate SessObj as $sessionpools.poolone.$new()
```

If $new() is successful the session instance assigned from the pool belongs to SessObj and is normally returned to the pool when SessObj is destroyed (for example, if SessObj is a local variable the session is returned to the pool when the method containing SessObj terminates). Alternatively the session can be manually returned to the pool by assigning some other object or zero to SessObj. The $new method returns NULL if all instances contained in the pool have already been assigned out.

Now you can use the $newstatement() method with the session object to create a statement to execute SQL in the normal way.

Session Pool Notation

The group $sessionpools supports the usual $findname(), $first(), $next() and $makelist() notation. The $remove() method is also implemented but not $add() since $makepool() is used to create a session pool.

A pool has the $name, $poolsize and $inuse properties. The $poolsize property is the number of instances stored in the pool and $inuse is the number of these which are currently assigned out. If you increase the value of $poolsize the new session instances are immediately constructed and if the hostname, username and password parameters were specified at $makepool, they are also logged on.

You can use poolRef.$poolsize.$assign(poolRef.$poolsize-1) to reduce the pool size, that is, to destroy a session instance in the pool, providing enough sessions are available to be destroyed. If not, then the pool size is reduced as and when sessions are returned to the pool. Note that the $poolsize property reflects the actual number of active sessions and not necessarily the number of sessions required by the user.

An alternative form of the $new() method is poolone.$new(pWaitSeconds) which is designed to be used in client methods running on the multi-threaded server. If there are no sessions currently available in the pool this waits for the specified number of seconds. Whilst it is waiting other threads are allowed to run and if a session is returned to the pool it will be used by the waiting method. At the end of the time period NULL is returned if no session has become available. Note that this waiting time period should be treated as approximate.

Destroying a Session Pool

A session pool normally exists for the lifetime of the Omnis process. You can forcibly destroy a session pool using notation, passing an item reference to the $sessionpools.$remove() method. For example:

```
Set reference ref to $sessionpools.pool1.$ref()
Do $sessionpools.$remove(ref) Returns #F
```

Diagnosing Problems

As of Omnis Studio 5.0, the session object provides the $debugfile and $debuglevel properties. These are useful in the event of program faults, if you are looking to optimise network traffic or if you are simply curious about how the DAM is executing commands. It should be noted that when debugging is enabled, there is a noticeable impact on performance. Hence, debugging should be reserved for application development and technical support issues- in which case use of $debuglevel 4 is recommended.

When $debugfile is set to a valid filename, the DAM starts writing debug information to this file- which is cleared before use. The file is created if it does not already exist. Debugging continues either until the session object is destructed or until $debugfile is set to an empty string. For macOS and Linux, a POSIX style path is expected and the user must have permission to write to the file at the specified location.

It is also possible to assign the value "stderr" to $debugfile. This is useful for macOS and Linux platforms where Omnis is run from the command prompt. Debug information will be written to the terminal window.

The $debuglevel property determines the level of debugging information that is written to the debug file and supports the following values:

Debug level	Description
0	No debugging. The debug file remains open but debugging output is suspended until $debuglevel is set to a higher level.
1	Base level debugging. At this level, the DAM base class writes high level descriptions about the operation of the DAM; statement prepares, executes, describing of result sets, fetching, etc. This is the default level of debugging.
2	Detail refinement 1. At this level metadata, property and method call information are also written including the SQL text associated with $prepare()s and $execdirect()s.
3	Detail refinement 2. At this level, buffer allocation, parameter and bind variable values are also written where possible.
4	Detail refinement 3. At this level, details of parameters passed to implementation API calls are also written, provided that the DAM implements this level of debugging.
5	Causes a time stamp to be prepended on to each debug entry. The time stamp is accurate to 1/60th second and reflects the time since the session object logged-on. Debug lines written before $logon() reflect the system up-time.

Session and Statement Properties and Methods

Below is a summary of the base methods and properties common to all object DAMs.
For details of DAM specific methods and properties, refer to the chapter on *Server Specific Programming.*

Session Methods

Method	Description
$begin()	$begin() explicitly begins a new database transaction.
$clear()	$clear() clears all statement objects based in the session and resets the session to its default state.
$coltext	$coltext(vVarRef) returns the DBMS specific SQL text corresponding to the data-type of the supplied

Method	Description
$commit()	$commit() explicitly commits the current database transaction.
$createnames()	$createnames(vTableDef,[bNullInfo,bPrimaryKeyInfo]) returns a DBMS specific text string consisting o and types based on the column types of the supplied list or row variable. The optional parameters bN can be used to request additional information about columns where the list has been defined from a
$definelistorrow()	$definelistorrow(&vListOrRow,cTableName) defines a list or row from the specified server table. (Studi
$insertnames()	$insertnames(wRowRef) returns a text string consisting the SQL column names and bind variable na supplied list or row variable.
$logoff()	$logoff() disconnects the session from the database.
$logon()	$logon(cHostname,cUsername,cPassword[,cSessionName]) connects the session to the DBMS using The optional cSessionName parameter registers the session with $root.$sessions and the SQL Brows
$makeschema()	$makeschema(pSchema,cTableName) makes a schema class based on the specified server table nar
$newstatement()	$newstatement([cStatementname]) returns an instance to a new statement object. cStatementnam
$newstatementref()	$newstatementref([cStatementname]) returns a reference to a new statement object. cStatementna
$nextnativeerror()	$nextnativeerror() retrieves a pending DBMS error code and error message, placing them in $nativee $nativeerrortext.
$rollback()	$rollback() explicitly rolls-back the current database transaction.
$rpcdefine()	$rpcdefine(cProcedureName,lParamList) defines the parameter structure for a subsequent call to sta and contents of the lParamList parameter are discussed in *Calling a Remote Procedure*
$selectnames()	$selectnames(vTableDef[,cTableName]) returns a text string consisting of comma delimited column r vTableDef. The optional cTableName is pre-pended to the column names if supplied.
$updatenames()	$updatenames(wRowRef) returns the text string for a SQL UPDATE clause based on the contents of v
$wherenames()	$wherenames(wRowRef[,cTableName,cComparison,cOperator]) returns the text string for a SQL WHE supplied list or row variable. The optional cTableName is pre-pended to the column names if supplie cOperator parameters can be used to modify the corresponding parts of the WHERE clause.

Session Properties

Property	Description
$allowstransactions	kTrue if the session is capable of manual transactions. (Read-only)
$apiversion	Version of the database client API that the DAM was built with. (Read-only)
$autobegintran	kTrue if the session automatically begins transactions, e.g. each time a transaction is committe (Read-only)
$batchsize	The desired batch size used when fetching multiple rows in a single network transaction. State objects. Default value: 1. (Studio 10.0.1)
$blobsize	The maximum size of a binary data item. Default value 10MB. Also used by $createnames() for
$charmap	Determines how character data received from and sent to the DBMS is mapped. For Non-Unic kSessionCharMapOmnis (the default), kSessionCharMapNative or kSessionCharMapTable.For U assumed for fetched 8-bit data; either kSessionCharMapLatin or kSessionCharMapRoman.
$commitmode	Indicates how SQL cursors behave when a session is committed. Either kSessionCommitClose,
$damname	The DAM name as shown in $root.$components. (Read-only)
$debugfile	When set to a valid filename, the DAM starts writing debug information to this file. Debugging $debugfile is set to an empty string. For macOS and Linux, a POSIX style path is expected.
$debuglevel	Determines the level of debugging information that is written to the $debugfile. 0 specifies no
$defaultdate	Used to specify the default date parts to be used when parts of the date are omitted at bind tir
$emptydateisnull	If set to kTrue, all out-going bound dates with an empty value will be set to NULL.
$encoding	Indicates the Unicode encoding used by the client library for bind data and fetched rows. The with Omnis. (Read-only)
$codepage	Set to one of the kUniType... constants found in the Catalog under *Unicode types*. Default value page used to interpret non-Unicode data. $codepage should match the character set of the da written correctly.
$errorcode	Returns the Internal Omnis error code generated by last executed session method. (Read-only
$errortext	Returns the error message generated by the last executed session method. (Read-only)
$fetch64bitints	If kTrue (default), 64-bit integers are fetched into 64-bit Integer fields. If kFalse, they are fetched provides backward compatibility with the old-style web client plug-in which does not support
$hostname	The hostname currently in use by the connection. (read-only)
$lobchunksize	The size (in bytes) of a single chunk when inserting large binary data. Default value is 32KB
$lobthreshold	The size (in bytes) of a binary data item at or above which chunking will occur. Default value is
$maptable	The path to the character map files to use when $charmap is assigned kSessionCharMapTable.
$nativeerrorcode	The error code generated by the DBMS or DBMS clientware in response to the last executed se
$nativeerrorpending	kTrue indicates that a further native error code and error text are available. See session.$nextna
$nativeerrortext	The error message generated by the DBMS or DBMS clientware in response to the last execute

Property	Description
$password	The password currently in use by the connection. (Read-only)
$quotedidentifier	If kTrue, table and column name identifiers returned from the $createnames(), $insertnames(), will be quoted “thus”, facilitating case-sensitive names and names containing spaces. Affects t 5.2 and later)
$rollbackmode	Indicates how SQL cursors behave when a session is rolled-back. Either kSessionRollbackClose (Read-only)
$sqldecimalseperator	The character that the DBMS uses to represent the decimal separator when storing numeric va
$sqlstripspaces	If kTrue, trailing blank characters are stripped from character data returned from the DBMS.
$sqlthousandsseperator	The character that the DBMS uses to represent the thousands separator when storing numeric
$state	Indicates the current state of the session object’s connection. Either kSessionStateLoggedOff c
$transactionmode	Used to set the transaction mode. Can be one of kSessionTranAutomatic (the default), kSession
$unicode	Used to enable/disable 8-bit compatibility mode.If kTrue (the default), all character data is exch non-Unicode DAM is adopted.
$username	The username currently in use by the connection. (Read-only)
$validateutf8	If kTrue and the $encoding is kSessionEncodingUtf8 and $unicode is kTrue, fetched UTF-8 data $charmap. Not implemented by all DAMs.
$version	Once a session has been established this is the version of the database the object is connectec

Statement Methods

Method	Description
$clear()	$clear() clears pending results and resets the statement.
$columns()	$columns(cTableName[,iFlags]) generates a result set describing the columns of the specified table. A parameter can be specified to generate column lengths for Number, Integer and Date columns. Valu together can be found in the catalog under Statement Flags.
$execdirect()	$execdirect([cSqlText]) directly executes the specified SQL text or executes the contents of the Staten cSqlText is omitted.
$execute()	$execute() executes previously prepared SQL text.
$fetch()	$fetch([lListOrRow,iRowCount,bAppend]) fetches the specified number of rows from the result set int variable. If iRowCount is omitted, a single row is fetched. If bAppend is kFalse or omitted, the list or ro cleared prior to the fetch.
$fetchinto()	$fetchinto(vParam1…vParamN) fetches one row of the result set and stores each column in the suppli variable for each column.
$fetchtofile()	$fetchtofile(cFileName[,iRowCount,bAppend,bColumnNames]) fetches the specified number of rows set and stores them in the specified file. iRowCount, bAppend and bColumnNames parameters are o
$indexes()	$indexes(cTableName[,iIndexType]) generates a result set providing information on the indexes of the The optional iIndexType can be one of kStatementIndexUnique (default), kStatementIndexNonUniqu kStatementIndexAll.
$nextnativeerror()	$nextnativeerror() retrieves a pending DBMS error code and error message for the statement object, $nativeerrorcode and $nativeerrortext.
$prepare()	$prepare([cSqlText]) prepares the supplied SQL text ready for subsequent execution. If omitted, the c statement buffer are prepared instead.
$results()	$results(lListOrRow) populates the supplied list variable with a description of the columns contained result set.
$rpc()	$rpc(cRpcName,vParam1…vParamN) calls the specified remote procedure. Any supplied parameters a procedure call.
$rpcparameters()	$rpcparameters(cRpcName) generates a result set describing the parameters used by the specified r
$rpcprocedures()	$rpcprocedures([cOwnerName]) generates a result set containing the names of remote procedures c specified user. If omitted, all procedure names are returned.
$tables()	$tables([iTableType,cTableOwner]) generates a result set containing the names of tables accessible by If cTableOwner is omitted all tables are returned. iTableType can be one of kStatementServerTable, kStatementServerView or kStatementServerAll (the default).

Statement Properties

Property	Description
$batchsize	The number of simultaneous rows to be retrieved for a single network transaction. Defaults to 1* but value. Not implemented by all DBMSs, in which case this property will be read-only.(*In Studio 10.0.1, from sessionObject.$batchsize).

Property	Description
$effectivebatchsize	Reflects the maximum attainable batchsize for the current statement. Will be less than $batchsize i batch size cannot be accommodated. (equivalent to $maxbuffersize / largest-column-size) (Read-or
$columncount	The number of columns contained in the current result set. (Read-only)
$columndelimiter	The character used to delimit column values when fetching data using $fetchtofile(). Defaults to kTa character value. Non-printable characters should be assigned using the chr() function.
$errorcode	The Omnis internal error code generated by the last statement method. (Read-only)
$errortext	The error message generated by the last statement method. (Read-only)
$maxbuffersize	Used in conjunction with $batchsize. Sets the maximum buffer size used to store an array of columr fetching. Defaults to 32KB but accepts any value larger than 255. Not implemented by all DBMSs, in will be read-only.
$maxresultsetsize	Designed to prevent Omnis from running-out of memory during $fetch() operations, this property li to the specified value. The default value is 100MB
$nativeerrorcode	Error code generated by the last statement method. (Read-only)
$nativeerrorpending	Indicates that a further error message is available. Use $nextnativeerror() to retrieve. (Read-only)
$nativeerrortext	Error message generated by the last statement method. (Read-only)
$resultspending	Indicates that the last statement method generated some results or that there is another result set
$rowcount	The number of rows in the current result set. If a particular session object cannot determine this val (Read-only)
$rowdelimiter	The character used to delimit row values when fetching data using $fetchtofile(). Defaults to kCr but character value. Non-printable characters should be assigned using the chr() function.
$rowsaffected	The number of rows affected by the last executed statement; usually an INSERT, UPDATE or DELETE session object cannot determine this value, this property returns –1. (Read-only)
$rowsfetched	The number of rows retrieved by the last fetch method to be executed. (Read-only)
$rpcreturnvalue	The return value of the most recently executed call to $rpc().(Read-only)
$state	A constant value indicating the current state of the statement object. (Read-only)
$statementname	The name which was assigned to the statement object during creation. (Read-only)
$sqlstripspaces	Denotes that fetched character data should have trailing spaces stripped.
$sqltext	The DBMS representation of the last SQL statement submitted to $prepare() or $execdirect(). (Read
$usecursor	Denotes that this statement object should be associated with a SQL cursor.

SQL Multi-tasking and SQL Workers

You can execute long-running tasks such as a SELECT statement on a separate background thread that reports back to the main thread as each task completes. To enable this functionality, the Omnis DAMs allow the creation of "SQL Workers" which are instantiated from a ***SQL Worker Object*** variable type available in the PostgreSQL, SQLite, DB2, Sybase, Oracle, MySQL, and ODBC DAMs (subject to your version of Omnis Studio).

SQL Worker object completion methods allow list fields and other form data to be populated asynchronously, making applications more responsive and potentially faster where multiple SQL Workers are used simultaneously. There is an example library in the **Hub** (available when you start Omnis Studio) showing how you can use the SQL Worker Objects: the example connects to a SQLite database and runs multiple queries, some with bind variables, running at the same time.

Overview

The SQL Worker Objects support three primary methods:

- **$init()**
  Initialises or resets a worker object ready to perform its task
- **$start()**
  Starts the worker task on a background thread (non-blocking); can be called multiple times to run different threads simultaneously
- **$cancel()**
  Aborts a worker task running on a background thread

There are additional properties to allow a running task to be discarded in place of a new task and to cancel such tasks as they become "orphaned". There is also a property to report the state of a worker object's running background thread.

Worker objects are created by sub-classing an Omnis Object class with the appropriate SQL Worker Object type. You initialise the object by supplying a SQL statement along with any bind variables that the SQL statement may need. Logon details or optionally the name of a session pool are also passed during initialisation.

A SQL Worker thread is dispatched by calling $start(). Upon completion, the worker thread calls back into the worker object's $completed() method, or $cancelled(), with the result set or error information.

SQL Worker Object Methods

Method	Description
$init()	$init(ParamRow). Initialises or resets a worker object ready to perform a unit of work.*
$start()	Starts the worker task running on a background thread (non-blocking).*
$run()	Starts the worker task running on the caller thread (blocks until complete). Intended for testing purposes.*
$cancel()	Aborts a worker task running on a background thread.*
$sessionref()	$sessionref(ObjectRef). Returns a reference to the session object being used by the underlying background t
$completed()	Called by the background thread upon completion of its work.
$cancelled()	Called by the background thread if the running background task was cancelled.

*Returns kTrue on successful execution, kFalse otherwise.

SQL Worker Object Properties

Property	Description
$cancelifrunning	If kFalse (the default), orphaned background tasks run to completion. If kTrue, they are instructed to cancel before being detached.
$waitforcomplete	If kTrue (the default), the Interface Object waits for completion of a running background task before the object can be used again. If kFalse, the running task is detached and a new background thread takes its place.
$state	Returns the current state of the underlying background task; either kWorkerStateCancelled, kWorkerStateClear, kWorkerStateComplete, kWorkerStateInit or kWorkerStateRunning.
$errorcode	On failure of a command function, contains the error code.
$errortext	On failure of a command function, contains the error message.
$threadcount	Reports the number of active threads spawned by the worker object, including detached threads.

Creating SQL Worker Objects

Worker objects are created by sub-classing an Omnis object class as a Worker Object. For example, using the Select Object dialog, assigning a $superclass for use with an Oracle connection results in: ***.ORACLEDAM.Worker Objects\OracleWorker***.

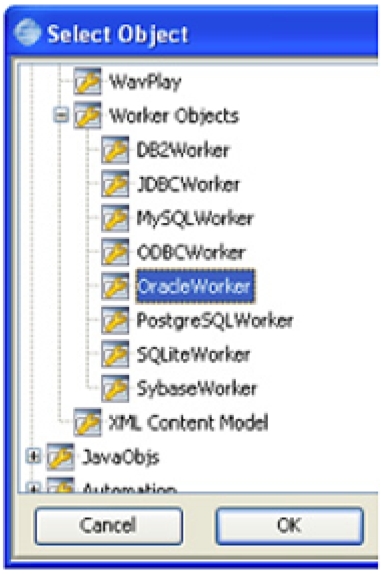


Figure 129:

Note that some objects may not appear in the Select Object dialog depending on your version of Omnis Studio.

To access worker functionality from your code, you then create one or more object instance variables of subtype <your-object >.

Worker Object Initialization

A worker object must be initialised on the caller thread before it can run. You initialise the object by supplying a SQL statement along with any bind variables that the SQL statement may require. Logon details, or optionally the name of a session pool, are also passed during initialisation.

The initialisation parameters are supplied to the $init() method via a row containing attribute values. Attribute names appear in the column headings. The attribute names recognised by the Worker Object are as follows (case-insensitive):

Attribute name	Attribute value
session	A session object or object reference. The session must be logged-on and in a useable state.
poolname	The name of an existing session pool. The worker will take a session object from this pool, returning it upon completion.
hostname	The hostname/IP address of the database server.
database	The database name to use for a logon.
username	The username to use for a logon.
password	The password to use for a logon.
query	The SQL statement to be executed by the worker object.
bindvars	A list containing bind variable values. Bind variables are matched by name. If the list contains multiple rows, the query is re-executed for each row.
work	A list containing multiple SQL queries and associated bind variables. If specified in place of *query* and *bindvars*, allows the worker object to execute multiple SQL statements and return any result sets associated with each query.

If the *session* attribute is supplied, the other logon attributes, i.e. hostname, database, username & password are ignored, since it is assumed that the session object is already in a useable state.

Please Note: In this mode, the session should be considered reserved for use exclusively by the worker. If the main application attempts to share a session object being used by a worker running on another thread, the results are undefined.

The logon parameters are also ignored if the poolname attribute is supplied. In this mode, the worker attempts to obtain a session from the named session pool, releasing it when the worker task completes. (If both session and poolname are supplied, poolname is ignored.)

Where neither, session or poolname are supplied, an internal session object is created dynamically. Valid logon credentials should be supplied via hostname, username and password. Although read during the call to $init(), the worker will not attempt to logon until the $run() or $start() method is called. In this mode, the session is automatically logged-off when the worker task completes (or is cancelled). Should you need to modify one or more session attributes before calling $run() or $start(), it is possible to obtain a reference to the session object by calling the worker object's $sessionref() method, for example:

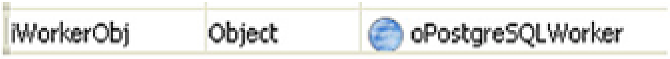
```
Do iWorkerObj.$sessionref(lObjRef) Returns #F
Do lObjRef.$port.$assign(5435)
```

The SQL text supplied via the query attribute may contain any SQL statement but ideally, should be a statement that normally takes an appreciable amount of time to execute, for example; a SELECT, UPDATE or DELETE statement. The query text may also contain one or more bind variables, specified using standard @[...] notation.

Bind variable values are supplied via a separate *bindvars* parameter. The supplied list is stored during $init() and read when the worker task starts. Where the list contains multiple rows, the worker re-executes the supplied SQL statement for each row of bind variables. Bind variable place holders in the query must reference columns by name within the bindvars list.

Running the Worker Object

The $start() method causes the worker task to run on a background thread. Thus, return from $start() is immediate and the main application is free to continue processing. For example, the iWorkerObj var has been created from the oPostgreSQLWorker class:



iWorkerObj	Object	oPostgreSQLWorker

Figure 130:

```
# iWorkerObj is an instance of an Object class
Calculate Params as row(iSQLText,'192.168. 0.10',iUser,iPassword,iDBName)

Do Params.$redefine(query,hostname,username,password,dat)
Do iWorkerObj.$init(Params)

Do iWorkerObj.$start() Returns #F
```

The $run() method is analogous to $start() but provided for debugging and testing purposes only. In this mode, the benefit of the worker object is negated owing to the fact that the worker will run on the same thread as the caller, thus blocking the caller thread until the worker task is complete.

Once initialised, a worker object may be run repeatedly if desired provided that the supplied session object remains useable, the session pool has one or more free sessions or that the logon credentials remain valid. Any bind variables supplied will be re-used each time the worker is run.

If an error occurs during $init(), $start() or $run(), an error message is returned via the object's $errorcode and $errortext properties.

Processing Worker Results

When complete, the worker task causes the main thread to jump into one of the worker object's *callback* methods:

- $completed()
  This method is called with a row parameter defined with two columns: Results: a three-column list containing zero or more SQL result sets (lists) together with their associated QueryNum and BindRow where appropriate. Errors: a four-column list containing ErrorCode, ErrorMsg, NativeErrorCode and NativeErrorMsg values.
- $cancelled()
  This method is called (without parameters) if the user calls $cancel() on the worker object whilst it is running. When cancelled, any pending results are discarded.

A library may contain multiple worker objects of a given type. Each may be assigned a separate unit of work (SQL query) and each may be started asynchronously. It is the responsibility of the completion method in each worker object to process its result information and make this available to the main application as/when it becomes available. For example, here is a $completed() method:

```
Calculate List as pRow.Results.1.1 ##extract the first result set
Calculate Errors as pRow.Errors ##extract list containing error info
If Errors.$linecount()>0
  Do method reportError(Errors)
Else
  Do method updateWindow(List) ## see comment below
End If

Calculate iThreadCount as $cinst.$threadCount ##shows how many threads are running
```

If the results returned by $completed() are to be displayed in a window or a remote form in the JavaScript Client, you will have to explicitly redraw the window instance, or in the case of a web form the remote client must contact the server to get updated automatically.

Executing Multiple SQL Statements

If the worker object is initialised with a *work* parameter in place of the *query* and *bindvars* parameters then a list of SQL statements will be executed and the worker's $completed() method will be called once all statements have been executed and any result sets have been generated. The work parameter is a list defined with two columns;

- query – a character column containing the SQL statement plus any required bind variable place holders
- bindvars – a list containing one or more rows. Bind variable place holders in the query must reference columns by name within the bindvars list

Example

```
Do work.$definefromsqlclass('scWork') ## scWork is defined as query(char) and bindvars(list)
Do work.$add('select * from table1 where col1 < @[binds1.col1]',binds1)
Do work.$add('select count(*) from table1',)
Do work.$add('select * from table2 where idCol >= @[binds2.idCol]',binds2)

Do work.$add('select oneCol from table3',)
Do initRow.$definefromsqlclass('scWorkInit') ## scWorkInit is defined as hostname(char), username(char), pa

Do initRow.$add(lHostname,lUsername,lPassword,work)
Do iWorker.$init(initRow) Returns #F

Do iWorker.$start() Returns #F
```

Note that when the worker object's $completed() method is called, the user is responsible for associating each result set with its originating SQL query, e.g.

```
Calculate ResultInfo as pResults.Results.1
Calculate Result1 as ResultInfo.Result
```

Manual Transaction Mode

If the session object passed to the worker is placed in manual transaction mode (kSessionTranManual), then all SQL statements executed by the worker object are executed as a single transaction which will either be committed on completion of the last statement, or rolled-back if one of the statements returns an error.
To use manual transaction mode, the worker object should be supplied with a pre-initialised session object, using the *poolname* or *session* parameters.
Manual transaction mode has no effect for SELECT statements, although if specified as part of a *work* list, the SELECT will not be executed if an error occurs in one of the preceding statements.

Worker State

The current state of a SQL Worker object may be interrogated by inspecting the object's $state property. This will return either:

- kWorkerStateCancelled – The worker has been cancelled.
- kWorkerStateClear – The worker is in a pre-initialised state.
- kWorkerStateComplete – The worker has completed.
- kWorkerStateInit – The worker has been initialised.
- kWorkerStateRunning – The worker is currently running.

For example:

```
If iWorkerObj.$state=kWorkerStateRunning & iWorkerObj.$waitforcomplete=kTrue
  Calculate iMesg as 'Still running (waiting for completion)'
  Quit method
End If
```

How SQL Worker Objects work

A worker object may be thought of as two sub-objects:

- **Interface Object**
  This takes the form of a standard Omnis non-visual object and provides the methods and properties described above
- **Background Object**
  Normally created/executed on a separate thread, the Background Object performs the actual work of the worker object, calling back to the Interface Object upon completion.

Behind each Worker Object, there is hidden Background Object.

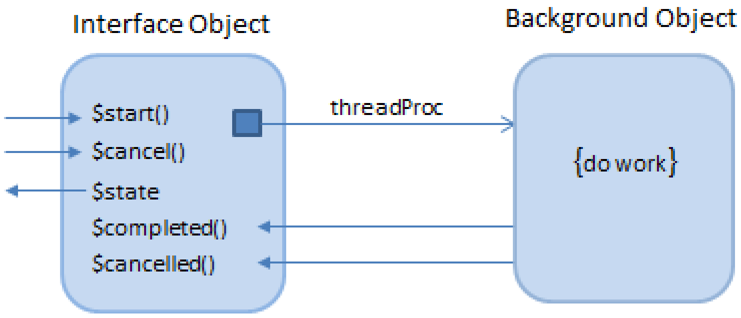
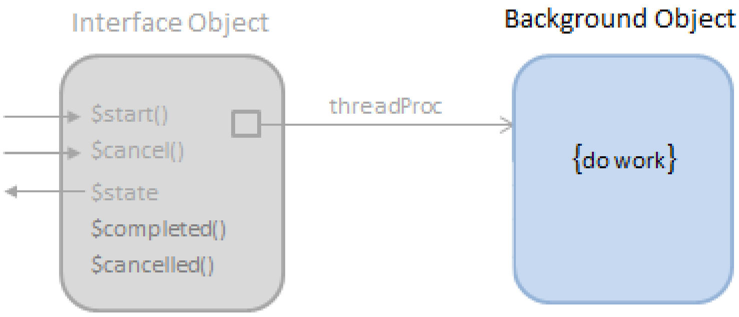


Figure 131:

Detaching Worker Processes

When instructed to $start(), the Background Object completes its work before calling back to the Interface Object's $completed() or $cancelled() method. For $run(), $completed() is *always* called since the worker object blocks- preventing cancellation.



A detached Background Object. The Interface Object may have gone out-of-scope
or may now point to a new Background Object.

For $start() however, the Interface Object may legitimately go out-of-scope or otherwise get destructed before the background thread completes. In this situation, the background thread has no object and hence no $completed()/$cancelled() method to call back to. Any results or error information will therefore be discarded.

Discarding Running Processes

In the case where the Interface Object remains in scope, it is possible to call $init() and $start() whilst the worker object is still running a previous task. In this case, the $waitforcomplete property determines whether the running process should be allowed to run to completion and call back to the Interface Object to signal completion.

If $waitforcomplete is kFalse, the running process is detached from the Interface Object as if the Interface Object were about to go out-of-scope. In this case however, a new Background Object is created which is then initialised and used to execute the new worker process, and potentially call back to the Interface Object when complete.

If $waitforcomplete is kTrue, Worker Main returns an error to the Interface Object if an attempt is made to re-use the worker object while the Background Object is still running. In this case, the worker object cannot be re-used until $completed()/$cancelled() has been called and the $state changes to indicate completion.

A worker object with its $waitforcomplete property set to kFalse, effectively becomes a "fire and forget" worker object, for example allowing a succession of INSERT, UPDATE or DELETE statements to be continuously dispatched to separate threads using the same worker object.

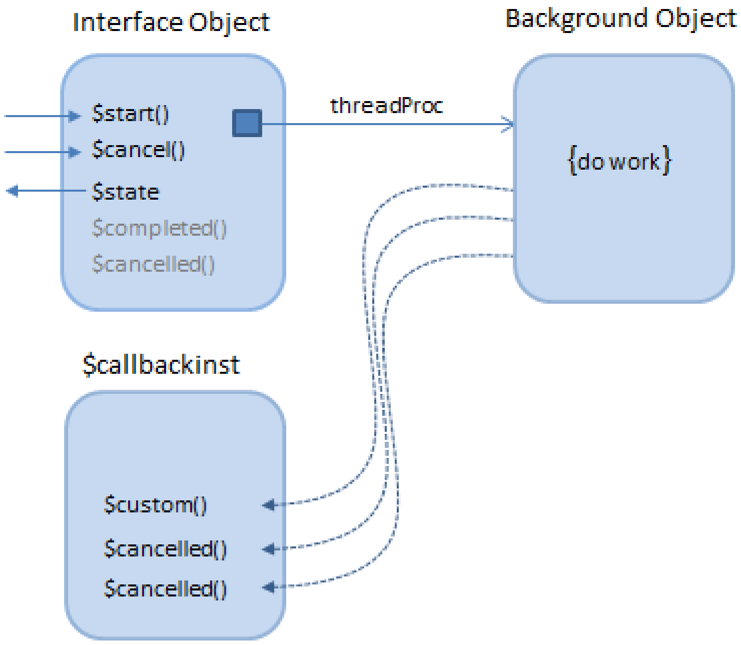
Cancelling Detached Processes

By default, orphaned background threads are allowed to run to completion. When a worker process becomes orphaned it may be preferable to issue a cancel request to the worker, especially where it may be processing a SELECT statement- for which the results will not be retrievable once detached from the Interface Object. This is achieved by setting $cancelifrunning to kTrue before the worker object gets re-used or destructed.

If $cancelifrunning is set to kFalse (the default), orphaned worker threads run to completion before discarding their results and exiting.

Alternative Completion Model

From Studio 8.0.2 Worker Objects support an alternative completion model. The $completed and $cancelled methods can optionally be sent directly to another instance. This means you do not need to sub-class the worker object, in order to receive its results. We would recommend that you use object references rather than objects for this technique.



Use of $callbackinst allows callbacks to be sent to another class instance.

In order to use this new functionality, there is a new property of worker object instances, called $callbackinst. If you do not use this new property, behavior is unchanged from Studio 8.0.1 and earlier.

For example, if iWorker is a SQL Worker Object (an instance variable in a window class), then within the window instance you can execute:

```
iWorker.$callbackinst.$assign($cinst)
```

You need to implement $completed and $cancelled in the window class methods. The parameters are as follows:

- $completed(*row,object*)
  where *row* is a parameter of type Row, same definition as that passed to $completed in the sub-classed object when not using $callbackinst.
  object is a parameter of type Object reference (when the worker object is an object reference) or Item reference (when the worker object is an object). object is the worker object for which $completed is being called.
- $cancelled(*object*)
  where *object* is the same as for $completed

Additional Notifications

The SQL Worker Objects supplied with Studio 8.1 and later support an interim '$progress' method to be called whilst the worker is running. If implemented in the $callbackinst;

- $progress(*row*)
  will be called with a row parameter containing a single column; 'Progress' calculated as a percentage of the total number of SQL queries that will be executed.

Where a work-list/query and bindvar combination is supplied, the total number of queries is calculated by adding the number of times each query will be executed. The received parameter value is suitable for direct assignment to a progress bar component, for example:

```
On evClick
  …
  Do iWorker.$callbackinst.$assign($cinst)
  Do iWorker.$init(lParams) Returns #F

  Do iWorker.$start() Returns #F
```

This code appears in the window instance's $progress method

```
Do $cwind.$objs.progress.$val.$assign(pRow.Progress)
```

The 'worker' sample component supplied with the External Component SDK Component SDK also demonstrates this functionality.

SQL Worker Lists

You can define a list or row variable from a SQL class (query, schema or table class), and associate a *SQL Session Object* with the variable in order to perform various SQL operations on the list, e.g. populate the list from the database, insert a row into the database.

Alternatively, you can specify that the SQL list or row will use a *SQL Worker Object* of the same DAM type as the SQL Session Object to perform SQL operations asynchronously (or synchronously, if preferred). Because the SQL Worker can run asynchronously, there are some differences in the way that you can use a table class from which the list or row is defined, compared to the way you use the table class with a SQL session object. To be specific, there is less scope to override SQL methods using the table class because of the need to execute the worker in a separate self-contained thread.

Using a Worker in a SQL List or Row

$useworker and $synchronous

If you want to use a worker object with your SQL list or row, you need to assign the property, $useworker to kTrue. $useworker must be assigned after assigning $sessionobject, and once you have assigned $useworker, you can no longer assign $sessionobject, or access $statementobject (the latter is destroyed if present when $useworker is assigned). $useworker cannot be assigned to kFalse.

In addition, there is the property $synchronous: if true, and $useworker is true, the worker object for the schema or table instance executes synchronously in the current thread rather than asynchronously in a separate thread. $synchronous defaults to false (meaning use another thread).

In addition, Omnis does not expose the worker properties $waitforcomplete and $cancelifrunning.

$waitforcomplete will always be kTrue, to make sure the application is notified of the success or failure of an operation, and $cancelifrunning is not relevant - the table will not invoke a new request until the previous request has completed - requests are queued by the table instance while the worker is busy processing a request.

Selecting & Fetching Data

Non-worker SQL lists and rows can operate in a synchronous manner. So $select() can be used to generate a result set, and $fetch() can be called multiple times to retrieve the result set.

SQL Worker based lists and rows cannot run in this simple synchronous manner, because the result set is generated by the worker in a separate thread. Therefore, worker SQL lists and rows have a new method, $selectfetch that performs both the select and the fetch of the data. It has the following definition:

- **$selectfetch()**
  $selectfetch([bDistinct=kFalse, iMaxRows=1, bAppend=kTrue, cText,...])

Note that $selectfetch() cannot be used with a row variable defined from a SQL class, so if you want to fetch data using a worker you must define a list from the SQL class.

Note also that you cannot override $selectfetch() in a table class. The parameters are as follows:

- **bDistinct**
  Pass this as kTrue to make the worker use a SELECT DISTINCT query rather than SELECT.
- **iMaxRows**
  The maximum number of rows to fetch. Must be between 1 and 10000000 inclusive.
- **bAppend**
  Pass this as kTrue to append the fetched data to the list, kFalse to replace the list contents with the fetched data.
- **cText,...**
  Any further parameters are treated as SQL text and appended to the generated SELECT or SELECT DISTINCT query.

Any errors that are detected before invoking the worker object, result in a call to $sqlerror in the table instance.

After fetching the data, the worker generates a notification to $completed in the table instance.

Inserts, Updates and Deletes

When using a worker, you cannot override $insert, $update or $delete in a table class.

When you execute these methods via a worker, the table instance copies the current values of the affected row (rows for $update) into the parameter list for the worker, and then starts the worker.

Any errors that are detected before invoking the worker object, result in a call to $sqlerror in the table instance.

On completion, the worker generates a notification to $completed in the table instance.

Smart List Methods

When using a worker, you cannot override $dowork, $doinserts, $doupdates, $dodeletes, $doinsert, $doupdate or $dodelete. Also, you cannot call $doinsert, $doupdate or $dodelete.

When you call $dowork, $doinserts, $doupdates or $dodeletes, the table instance generates a single query for each of the relevant operations insert, update and delete. The instance then copies bind variable values into a list, for each set of rows to be inserted, updated or deleted. Finally, the table instance starts the worker with the copied data as its parameters. When the worker completes, the worker generates a notification to $completed, that identifies any rows for which an error occurred, with information about the error.

Note that as soon as you call $dowork, $doinserts, $doupdates or $dodeletes, the smart list updates just before starting the worker

Any errors that are detected before invoking the worker object, result in a call to $sqlerror in the table instance.

Completion Row

The table instance properties $rowsaffected and $rowsfetched are not relevant when using a worker.

$completed in the table instance is passed a row variable parameter with columns as follows:

- **errorcode**
  An error code. Zero means the worker was successfully passed the query and bind variables. Note that the query or queries may still have failed - see the errors column.
- **errortext**
  Error text describing the errorcode.
- **work**
  The list of queries and bind variables that were passed to the worker. This has the usual structure for SQL workers - two columns, query and bindvars.
- **errors**
  If errorcode is zero, this is a list of queries that generated a SQL error of some sort. This has the same structure as the Errors column generated by a SQL worker in the worker completion row.
- **rowsFetched**
  If a call to $selectfetch successfully fetched some rows, this is the number of rows fetched.

Chapter 8—SQL Classes and Notation

Omnis has three *SQL classes* that provide the interface to your server database: they are the **Schema class, Query class,** 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; or when you declare a list or row variable, you can set its subtype to a schema or query class. 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.

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, or a schema can be used as the subtype of a list variable. 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

- **$objs**
  the group of columns in the schema class
- **$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

-. **Name**
the name of the column

- **Type**
  constant representing the Omnis data type of the column
- **Subtype**
  constant representing the data subtype of the column
- **Description** (optional)
  a text string describing the column
- **Primary-key** (optional)
  a boolean set to kTrue if this column is a primary key. If omitted it defaults to kFalse
- **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.
- **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)

- **$name**
  the name of the column
- **$coltype**
  the Omnis data type of the column
- **$colsubtype**
  the Omnis subtype for the data type of the column
- **$colsublen**
  the maximum length for Character and National columns
- **$desc**
  a text string describing the column
- **$primarykey**
  if kTrue the column is a primary key

- **$nonull**
  if kTrue the column does not allow null values
- **$order**
  the position of the column in the list of columns in the schema class
- **$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.

Making a Schema from a Server Table

You can make a schema class that matches the columns in a database table automatically using the $makeschema() session method:

```
Do SessObj.$makeschema(pSchema,pTableName) Returns #F
```

The parameter pSchema is a reference to an existing schema class that will be overwritten with the definition from the server table pTableName 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, or a query class can be used as the subtype of a list variable.

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:

- A schema name, which determines the table to be used in the SQL statement.

- 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.
- 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:

- **Schema name**
  the name of the schema, which can be qualified by a library name
- **Column name** (optional)
  the name of the column in the schema
- **Alias** (optional)
  the alias used to eliminate duplicate list column names
- **$schema**
  the name of the schema, which can be qualified by a library name
- **$colname**
  the name of the column in the schema; if empty, all columns from the schema class specified in the $schema property are included
- **$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 $schema; otherwise, Omnis uses a non-empty alias in the place of the column name
- **$order**
  the position of the object in the class
- **$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

- Create the schema or query class in the Studio Browser
- Define the columns in the schema or query class
- Use the SQL Browser to open the SQL session for your database
- 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,...] [,,con-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.

- **$select()**
  generates a Select statement and issues it to the server
- **$selectdistinct()**
  generates a Select DISTINCT statement and issues it to the server
- **$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.

- **$insert()**
  inserts a row into the server database
- **$update()**
  updates a row (or rows if the where clause applies to several rows) in the server database
- **$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.

- **$doinserts()**
  inserts all rows in the list with the row status kRowInserted
- **$dodeletes()**
  deletes all rows in the list with the row status kRowDeleted
- **$doupdates()**
  updates all rows in the list with the row status kRowUpdated
- **$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.

- **$doinsert()**
  inserts a single row with row status kRowInserted
- **$dodelete()**
  deletes a single row with row status kRowDeleted
- **$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.

- **$undoinserts()**
  removes any inserted rows from the list
- **$undodeletes()**
  restores any deleted rows to the list, and resets their status to kRowUnchanged
- **$undoupdates()**
  restores any updated rows to their original value, and resets their status to kRowUnchanged
- **$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.

- **$selectnames()**
  returns a comma-separated list of column names in the list or row variable, suitable for inclusion in a SELECT statement
- **$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
- **$updatenames()**
  returns a text string containing a SET clause, suitable for inclusion in an UPDATE statement
- **$insertnames()**
  returns a text string containing a list of columns and values for a row variable, suitable for inclusion in an INSERT statement
- **$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.

- **$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.

- **$sqlclassname**
  the name of the associated schema or query class used to define the columns of the list; this property is NOT assignable
- **$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 $useprimarykeys to kTrue when defining the list, if and only if there is at least one primary key column in the list
- **$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

- **$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

- **$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

- **$colsinset**
  the number of columns in the current result set for the session used by the table instance; this property is NOT assignable

- **$rowsaffected**
  the number of rows affected by the last call to $insert(), $update(), $delete(), $doinserts(), $doupdates(), or $dodeletes()

- **$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

- **$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()
```

$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

Constant	Description
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.$fetch() Returns lv_Status
  If lv_Status=kFetchFinished=kFetchError
    Do iv_SQLData.$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:

Parameter	Description
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
```

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.c
```

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]) 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],...,
```

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]) 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.c
```

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]) 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
```

Chapter 9—Server-Specific Programming

This chapter contains server-specific information for each of the proprietary databases and middleware configurations that can be accessed in Omnis Studio.

The type of database you can access in Omnis Studio will depend on the edition of Omnis Studio you have. All editions allow you to access:

- PostgreSQL
- SQLite

In addition to those above, other editions, including the Professional Edition, may provide access to:

- Oracle
- Sybase
- DB2
- MySQL
- ODBC
- Amazon SimpleDB

The following are for legacy or existing Omnis applications only and should not be used for new applications:

- OmnisSQL DAM
  provided for backwards compatibility for legacy apps only, should not be used for new apps
- JDBC
  support for JDBC has been removed in Studio 10 or above, but the supporting files can be obtained by contacting Omnis Support

Every DBMS has its own specific, extra features that are not part of the SQL standard. Some of these features are supported by sending a specific database command to the server using the standard $prepare(), $execute(), and $execdirect() methods. Others are implemented as special session and statement object properties and methods.

In addition to the DAMs provided with Omnis Studio, FrontBase Inc also produce and maintain DAMs for Omnis Studio, see: http://www.frontbase.com

Server and Clientware Compatibility

Some aspects of functionality and compatibility are subject to frequent change with new versions of server software or clientware. Check the Omnis Developer website for details of software versions supported and middleware configurations at: www.omnis.net/developers/resources/dams/

There you can view the latest information about the Clientware supported by the different server databases supported in the current version Omnis Studio.

64-bit DAMs

The DAMs provided with the 64-bit version of Omnis Studio 8.0 or higher use 64-bit architecture. This means that you will need to install separate 64-bit clientware where appropriate. The 64-bit DAMs are not interoperable with 32-bit client libraries and vice-versa. For single-tier and embedded DAMs, including DAMSQLITE, DAMOMSQL, DAMMYSQL, DAMPGSQL and DAMAZON, all necessary changes have been made. The 64-bit ODBC DAM requires the 64-bit ODBC Administrator library and should be used with 64-bit ODBC Drivers to ensure compatibility.

PostgreSQL

This section contains the additional information you need to access a PostgreSQL database, including server-specific programming, data type mapping to and from PostgreSQL, as well as troubleshooting. For general information about logging on and managing your database using the Omnis SQL Browser, refer to the earlier parts of this manual.

For additional information on changes to the PostgreSQL DAM, refer to the readme file which accompanies your Omnis download.

PostgreSQL Client Libraries

This section discusses the PostgreSQL client library, which must be present on the library search path before the PostgreSQL DAM can be used.

Win32 platforms

For Win32 platforms, the library search path includes the Windows\System32 folder or any location in the PATH environment variable, including the folder containing omnis.exe. The Win32 client library is named **libpq.dll**.

Linux and macOS platforms

The Linux and macOS ports of the PostgreSQL DAM look for **libpq.so** or **liqpq.dylib** respectively.

In most cases the library present on your system will be labelled according to the version you have installed. For example on Linux, libpq.so might be a symbolic link to the target library libpq.so.5.0. A detailed directory listing shows this relationship, e.g.

```
12 2007-01-15 10:20 libpq.so -> libpq.so.5.0
117338 2007-01-15 10:20 libpq.so.5.0
```

Under Linux and macOS therefore, it is essential that the target library and symbolic link to it *both* exist either in the library search path or in the same folder as the Omnis executable. For macOS, the library search path also includes the Omnis.app/Contents/Frameworks folder.

Properties and Methods

In addition to the "base" properties and methods documented in the *SQL Programming* chapter, the PostgreSQL DAM provides the following additional features.

Session Properties

Property	Description
$maxvarchar	Defines the maximum size above which- Omnis Character fields will be mapped to TEXT type instea VARCHAR. The default value for this property is 2000.
$database	Used to set the additional dbname logon parameter. If not specified, defaults to be the same as the
$service	Service name to use for additional parameters. It specifies a service name in pg_service.conf that ho additional connection parameters. This allows applications to specify only a service name so connec parameters can be centrally maintained.
$protocolversion	(Read-only)This property reports the communication protocol version supported by the client library DAMPGSQL requires version 3.0 or higher in order to work correctly.
$backendpid	(Read-only) Following logon, this property holds the process ID of the backend server process handl connection. This may be useful for debugging purposes since the PID is reported in NOTIFY messag
$port	Used to set the additional port logon parameter. This property has a default value of 5432.
$socket	(Read-only) Following logon, this property holds the file descriptor number of the connection socke server. A valid descriptor will be greater than or equal to 0; a result of -1 indicates that no server conn currently open.
$options	Used to specify additional text to be appended to the logon connection string. One or more parame settings can be added, separated by spaces. The options string is limited to 255 characters. Discussio advanced connection options is beyond the scope of this text but an example string might be: Do sess.$options.$assign("options='-c geqo=off' sslmode=require")
$logontimeout	Maximum wait for a connection, in seconds. Zero implies wait indefinitely. The default timeout is se seconds. A timeout of less than 2 seconds is not recommended.
$timezone	Character string representing the time zone to be appended on to bind variables being inserted int and TIMESTAMPTZ columns. The default time zone is "+00" but $timezone will accept any character characters max).
$usetimezone	If set to kTrue, the value contained in $timezone is appended to outgoing Time and Datetime bind v This property also affects the text returned by $createnames() for Time and DateTime columns. $tim be ignored during insert/update of TIMESTAMP & TIME columns
$serializable	If set to kTrue, manual transactions will be created using the Serializable isolation level. When set to (the default), manual transactions will be created using the Read Committed isolation level.

Property	Description
$readonly	If set to kTrue, manual transactions will be created using read-only access mode. When set to kFalse default), transactions will have read/write access.
$schema	The optional schema name to be prepended to table names. Used by the SQL Browser when perfor SELECTs. The default schema name is an empty string.
$numericprecision	Defines the precision used by $createnames() when mapping Omnis number (dp) columns to the N type. Cannot be set lower than the default value: 15.
$sequencetoint	If set to kTrue, the Omnis Sequence type is mapped to INTEGER. If set to kFalse (the default), the Se type is mapped to SERIAL. Affects $createnames() and outward bind variables.
$char38touuid	If set to kTrue, Omnis character types of field length 38 are mapped to the PostgreSQL 8.3 Universal Identifier type (UUID).
$char39tooid	If set to kTrue, Omnis character types of field length 39 are mapped to the PostgreSQL Object Ident (OID).
$defaultdateisempty	If kTrue, fetched datetimes matching $defaultdate are treated as empty values.
$programname	If specified, registers an application name during $logon() which will be stored in the pg_stat_activit
asof 35971 $listenername	If specified, registers a name for the $listen session which will be stored in the pg_stat_activity table
$infinitydates	If kTrue, date value 31 Dec 9999 and datetime value 31 Dec 9999 23:59:59 maps to the special value; ‘i
asof 35977 $cannotify	While kTrue, the $notify() method is enabled. While kFalse, notifications are queued. This property c used to temporarily disable notifications for example; while a thread-critial method is running.

Session Methods

Method	Description
$connectstatus()	Returns a PGSQLDAM Connection Status constant representing the current state of the connection to the database server, or empty if not connected.
$escapebinary()	Returns a text-escaped representation of the supplied binary variable, suitable for use in an SQL statement as a quoted string literal. The returned string does not include the quotes.
$getssl()	Returns qtrue if the connection is using SSL, qfalse otherwise. An optional list parameter can also be passed to return additional information. Currently, the SSL type and version are returned.
$listen()	Listens for the specified notification channel name and calls obj.$notify() when triggered. Call $unlisten to remove the listener.
$notify()	Create this method inside an object class of subtype PGSQLDAM.PGSQLSESS. obj.$notify() will be callec with a single parameter of type *row* when a client issues a NOTIFY command with a channel name previously registered using the $listen() method. The row parameter will be defined as: Channel: the notification channel name PID: the ID of the calling client process Message: character variable containing the message ‘payload’
$transactionstatus()	Returns the current in-transaction status of the server. The status can be kPgSqlTranIdle (currently idle) kPgSqlTranActive (a command is in progress), kPgSqlTranInTrans (idle, in a valid transaction block), or kPgSqlTranINError (idle, in a failed transaction block). kPgSqlTranUnknown is reported if the connectior is bad. kPgSqlTranActive is reported only when a query has been sent to the server and not yet completed.
$parameterstatus()	Looks up a current parameter setting of the server. Supported (string) parameters include server_versio server_encoding, client_encoding, is_superuser, session_authorization, DateStyle, TimeZone, integer_datetimes, and standard_conforming_strings. For a full list, refer to the API documentation for the PQparameterStatus function.
$reset()	Resets the communication channel to the server. This function will close the connection to the server and attempt to re-establish a new connection to the same server, using all the same parameters previously used. This may be useful for error recovery if a working connection is lost.
$cancel()	Requests that the server abandon processing of any transactions pending on the session. Successful execution is no guarantee that the request will have any effect, however. If the cancellation is effective, the current command(s) will terminate early and return an error result.
$addcustomtype()	$addcustomtype(iFieldlength,cDatatype) Creates a custom data type mapping for specified Omnis character subtypes. Intended to allow creation and insertion into PostgreSQL 8.3 enum, xml and json columns.
$clearcustomtypes()	$clearcustomtypes() Removes all previously created custom data type mappings.
$lobimport()	$lobimport(cFilename[, iOid]) Imports the contents of the specified operating system file into the database and returns the new OID on success, zero otherwise. If a specific OID value is desired, it may b passed in via parameter 2. Must be called within a manual transaction block.
$lobexport()	$lobexport(cFilename, iOid) Exports the object specified by iOid into the specified operating system file Must be called within a manual transaction block. Returns kTrue on success.

Method	Description
$lobcreate()	$lobcreate([iOid]) Creates a new large object and returns the new OID value on success, zero otherwise. a specific OID value is desired, it may be passed in via parameter 1. Must be called within a manual transaction block.
$lobunlink()	$lobunlink(iOid) Removes the specified object from the database and unlinks the Object Identifier, effectively deleting the object. Must be called within a manual transaction block. Returns kTrue if the object was successfully unlinked.
$lobopen()	$lobopen(iOid[, bReadOnly]) Opens the specified large object for reading/writing and returns the large-object descriptor which is only valid for the duration of the current transaction. If bReadOnly is specified (kTrue), a read-only snap shot of the object is taken as it was at the start of the transaction.
$lobwrite()	$lobwrite(iDesc, xBinary[, iSize]) Writes the supplied binary data to the specified large-object descriptor, returning the number of bytes that were written on success, or -1 on failure. By default, the entire binary field is written unless iSize is specified.
$lobread()	$lobread(iDesc, xBinary[, iSize]) Reads the large object specified by iDesc into xBinary and returns the number of bytes read on success, or -1 on failure. If specified, iSize bytes are allocated and read from the large object. If omitted, *$blobsize* bytes are allocated/requested.
$lobseek()	$lobseek(iDesc, iOffset, iWhence) Moves the read/write pointer within an open large object by *iOffset* bytes. iWhence governs how the offset is interpreted; kPgSqlSeekSet specifies an absolute offset from the start of the object, kPgSqlSeekCur specifies an offset from the current position, kPgSqlSeekEnd specifies an offset from the end of the object. Returns the new location on success, or -1 on failure.
$lobtell()	$lobtell(iDesc) Returns the current position of the read/write pointer within the large object specified by iDesc, or -1 on failure.
$lobtruncate()	$lobtruncate(iDesc, iSize) Resizes the specified large object to *iSize* bytes. If iSize is larger than the curren size, the large object is padded with null bytes. Returns kTrue on success, kFalse otherwise.
$lobclose()	$lobclose(iDesc) Explicitly closes the specified large-object descriptor. Any large-object descriptors that remain open at the end of a transaction will be closed automatically. Returns kTrue on success, kFalse otherwise.
$unlisten()	Removes the specified notification channel from the listener queue.

Statement Properties

Property	Description
$sqlstate	(Read only) On error, this property contains the five-character SQLSTATE associated with the $nativeerrortext. Refer to the PostgreSQL reference manual for a full list of SQLSTATEs.

Logging on to PostgreSQL

In addition to the hostname, username and password parameters provided by the $logon() method, the PostgreSQL DAM provides several session properties which enable additional logon parameters to be set. These should be set before calling $logon().

- $database is used to specify the dbname connection parameter.
- $port is used to specify the port connection parameter.
- $logontimeout is used to specify the connect_timeout parameter.
- $options is used to specify further optional connection parameters.
- $service is used to specify a service (filename) to use for additional parameters.

The $logon() *hostname* parameter can either be specified as an IPv4 (e.g. 192.168.1.100) or an IPv6 IP address or as a machine name. If prefixed with a slash, *name* refers to a Unix domain name.

Metadata Functions

- $indexes()

The DamInfoRow for $indexes() is defined with a single column containing the SQL text used to define the index.

- $tables()

The PostgreSQL DAM implements $tables() slightly differently. In particular, only the kStatementServerTable and kStatementServerView parameters are supported. This is because the processes for querying tables are incompatible with those for querying views. (kStatementServerAll defaults to kStatementServerTable).

The DamInfoRow for $tables() is defined with three Boolean columns with additional information on the table or view: HasIndexes, HasRules & HasTriggers.

Transactions

PostgreSQL supports two transaction isolation levels: Read Committed (the default) and Serializable. Using Read Committed mode, a statement can only see rows that were committed before the current transaction began. Using Serializable mode, all statements in the current transaction can only see rows that were committed before the first query or data-modification statement was executed in this transaction.

Transactions can also be instantiated as read-only if required. This enables significant performance improvements for read operations. When a transaction is read-only, the following SQL commands are disallowed: INSERT,UPDATE,DELETE and COPY FROM if the table they would write to is not a temporary table; all CREATE,ALTER and DROP commands; COMMENT,GRANT,REVOKE,TRUNCATE,EXPLAIN ANALYZE and EXECUTE if the command they would execute is among those listed. Please refer to the PostgreSQL documentation on transactions for further details.

When using manual transaction mode (kSessionTranManual), the transaction isolation level can be switched between Read Committed and Serializable using the $serializable session property.

The access mode can be changed using the $readonly session property.

The PostgreSQL DAM treats the kSessionTranAutomatic and kSessionTranServer transaction modes identically. In either of these modes the server automatically begins and commits read/write transactions.

Remote Procedure Calls

PostgreSQL does not support the concept of stored procedures but supports functions instead. This has a few implications as described below.

- $rpcprocedures()
  The DamInfoRow returned by $rpcprocedures is defined with the following columns:

Column	Description
Language	The implementation language or call interface for this function.
IsAgg	kTrue if this is an aggregate function.
SecDef	kTrue if this function is a security definer (i.e. a “setuid” function).
IsStrict	kTrue if this is a “strict” function. Strict functions must be prepared to handle null inputs.
RetSet	kTrue if the function returns a result set (i.e. multiple values of the specified data type).
Volatile	Indicates whether the function result depends only on its input arguments, or is affected by outside factors. It is i for “immutable” functions, which always deliver the same result for the same inputs. It is s for “stable” functions, whose results (for fixed inputs) do not change within a scan. It is v for “volatile” functions, whose results may change at any time, that have side-effects for other functions or tables or functions which cannot otherwise be optimised.
Source	Indicates how the function should be invoked. It might be the actual source code of the function for interpreted languages, a link symbol, a file name, or just about anything else, depending on the implementation language/call convention.

- $rpc()
  Calling $rpc() is similar to executing a SQL SELECT statement of the form:

```
SELECT * from proc_name (param1, param2, … )
```

with the exception that $rpc() will also set any InputOutput or Output parameters.

Any return value generated by the function will be available via $rpcreturnvalue although in the case where the function generates a result set, it may be preferable to retrieve the entire set by calling $fetch(). The value returned by $rpcreturnvalue is also returned as the first row of this result set.

Notification Channels

PostgreSQL supports asynchronous notification channels via its LISTEN, UNLISTEN and NOTIFY SQL commands. You can register the session object as a listener for a notification channel using the $listen() method, specifying the channel name to listen for. $unlisten() removes the listener.

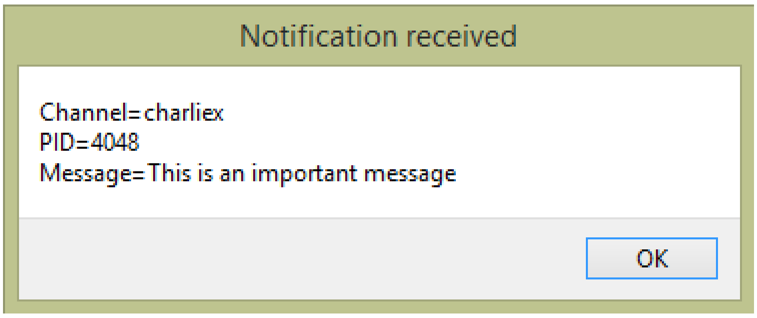
Once registered, if any client executes a NOTIFY for that channel name, the listener calls the session object's $notify() method with parameters that indicate the channel name, the notifier's process ID and an optional 'payload'/ text string.

In the following example, oPgSQL is an object class with $superclass .PGSQLDAM.PGSQLSESS:

```
Do oPgSQL.$logon('192.168.0.96','postgres','postgres','session1') Returns #F
Do oPgSQL.$listen('charliex') Returns #F
Do oPgSQL.$newstatement() Returns statObj
Do statObj.$execdirect("notify charliex,'This is an important message'") Returns #F
```

Create method oPgSQL.$notify() with a single parameter of type Row to be called whenever a notification is received. For example:

```
OK message Notification received {Channel=[pRow.Channel]//PID=[pRow.PID]//Message=[pRow.Message]}
```



Notification received

Channel=charliex
PID=4048
Message=This is an important message

Figure 132:

As of Studio 10.2, $listen() automatically encloses the channel name in double quotes when $quotedidentifier is kTrue. Unquoted channel names containing illegal characters now cause $listen() to return kFalse with $nativeerrortext; "Malformed unquoted identifier".

Valid unquoted identifiers are case-insensitive, commence with a-z or _ and can contain a-z, 0-9 as well as _ and $. Quoted identifiers are case-sensitive and can contain any characters. PostgreSQL identifiers have a maximum length of 63 characters. As of Studio 10.2, the message size can be up to 8000 ANSI characters.

Similar to the $programname session property, you can assign or change the name for the listener session using the $listenername property. This name will subsequently appear in the pg_stat_activity system table.

To prevent incoming notifications from interrupting the currently executing method, you can use the $cannotify property. Setting this to kFalse, disables the $notify() method and causes incoming notifications to be queued. Set $cannotify to kTrue again in order to receive any queued notifications.

Handling Dates

When kTrue, the $defaultdateisempty tells the DAM to convert retrieved datetimes to empty when they match $defaultdate.

UUID, ENUM and XML Column Types

Support for the following data types is available in Omnis Studio 4.3.1 and above.

UUID

The PostgreSQL DAM is able to read and write Universally Unique Identifiers. An example of a UUID in standard form might be:

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11 (36 characters)
```

but the DAM also accepts UUIDs formatted without hyphens and/or encapsulated using curly braces.

Output from UUID columns is always in the standard form.

To allow input binding of UUIDs and to make $createnames() return UUID types, it is necessary to set $char38touuid to kTrue. Once set, the Omnis Character 38 data subtype maps to UUID.

Note: there is no facility either in the PostgreSQL client library or in the DAM to create UUID values. This must be implemented by the Omnis application.

ENUM

Enumerated types are created by executing CREATE TYPE statements, for example:

```
Do statObj.$execdirect("CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy')") Returns #F ##creates the enumerat
```

To make Omnis map certain character sub types to ENUMs, the $addcustomtype() method is provided.

The following example maps the Omnis Character 2001 data subtype to the “mood” enumerated type:

```
Do sessObj.$addcustomtype(2001,'mood') Returns #F
```

Once set, this mapping affects the text generated by $createnames() as well as input binding.

To clear previously defined enumerated type mappings, the $clearcustomtypes() method is provided.

XML

The $addcustomtype() method can also be used to force an Omnis Character subtype to map to the XML data type, for example:

```
Do sessObj.$addcustomtype(10001,'xml') Returns #F
```

As above, this mapping affects the text generated by $createnames() as well as input binding and remains in effect until $clearcustomtypes() is called.

Large Object Support

As of Omnis Studio 5.1.1, the PostgreSQL DAM supports additional session object methods for manipulating large objects stored in the database.

The methods; $lobimport(), $lobexport() and $lobunlink() complement their SQL equivalents (lo_import(), lo_export() & lo_unlink()) with the exception that the client-side methods operate on files in the client machine’s file system. The SQL functions operate on files in the database server’s file system. In other respects, the operation of these methods is comparable:

- $lobimport() – creates a large object and imports data into it from a local file.
- $lobexport() – retrieves data from a large object and writes it to a local file.
- $lobunlink() – removes a large object from the database.

There is an additional client-side method for creating a large object:

- $lobcreate() – creates a new (empty) large object and returns the OID value.

The large objects are identified by their OID values, which can subsequently be stored and retrieved in database *Oid* columns in a similar fashion to standard integers.
Once created, the following methods can be used to manipulate data inside large objects:

- $lobopen() – opens a large object (OID) and returns a large object descriptor.
- $lobclose() – closes a large object descriptor.
- $lobread() – reads zero or more bytes from a descriptor into a Binary variable.
- $lobwrite() – writes zero or more bytes to a descriptor from a Binary variable.
- $lobseek() – repositions the read/write pointer within a large object.

- $lobtell() – reports the current pointer position within a large object.
- $lobtruncate() – resizes a large object to the desired size (in bytes).

To use these methods, it is important to note that large object operations must be performed within a **single transaction**, i.e. in manual transaction mode. Any open large object descriptors are automatically closed upon $commit(). For example:

```
Do cSess.$transactionmode.$assign(kSessionTranManual)
Do cSess.$begin()
Do cSess.$lobcreate() Returns lOid ## create new oid

Do cSess.$commit()
Do cSess.$begin()
Do cSess.$lobopen(lOid) Returns fileDesc
Calculate lBinary as 'Some Unicode character data'
Do cSess.$lobwrite(fileDesc,lBinary) Returns lNumBytes ## write data into the large object
Do cSess.$lobseek(fileDesc,8,kPgSqlSeekSet) Returns lFilePos ## move the read/write pointer to byte 8/chara
Do cSess.$lobread(fileDesc,lCharValue) Returns lNumBytes ;;=> 'me Unicode character data'

Do cSess.$commit() ## commit closes the descriptor
```

For further information on the behavior of these methods and the parameter values that may be applied, please refer to the Session Methods section above.

JSON Column Types

As of Studio 8.0.3 you can select and insert JSON strings into PostgreSQL JSON and JSONB columns. In earlier versions of Studio, there was partial support for insertion of JSON strings and it was possible to select from JSON columns using the CAST(… as VARCHAR(n)) operator.

The client library will parse and validate text before insertion into JSON/JSONB columns. You can optionally use this feature to validate JSON strings, for example:

```
Do cStat.$execdirect("select '5'::json as myCol") Returns #F

Do cStat.$fetchinto(lResult) ## Returns 5
Do cStat.$execdirect("select '{""col1"":1,""col2"":""mostly cloudy"",""col3"":true}'::json as myJSON") Retu

Do cStat.$fetchinto(lResult) ## Returns {"col1":1,"col2":"mostly cloudy","col3":true}
Do cStat.$execdirect("select '{""col1"":1,""col2"":""600 meters"",""col3"":[[""one"",""two"",""three""]}'::

Do cStat.$fetchinto(lResult) ## Returns {"col1":1,"col2":"600 meters","col3":["one","two","three"]}
```

Any of the resulting strings can then be inserted into the database:

```
Do cStat.$execdirect('create table jsontest(col1 int, col2 jsonb)') Returns #F
Do cStat.$execdirect('insert into jsontest values(1,@[lResult])') Returns #F
```

Note that JSON string literals must be suitably *escaped* in respect of quotes and square brackets, as shown above.

To insert JSON strings using bind variables, the $addcustomdatatype() method should be used. This tells the DAM to map Omnis character fields of a specific fieldlength to JSON and/or JSONB columns, and also allows $createnames() to generate JSON and/or JSONB columns. For example:

```
Do cSess.$addcustomtype(1000,'JSON') Returns #F
Do cSess.$addcustomtype(1001,'JSONB') Returns #F
Do lList.$definefromsqlclass('scTest')
Calculate lSql as cSess.$createnames(lList)
```

Where scTest defines Character columns of length 1000 or 1001, the 'JSON' and 'JSONB' column type will be returned accordingly.

PostgreSQL Data Type Mapping

Omnis to PostgreSQL

Omnis Data Type	PostgreSQL Data Type
CHARACTER	
Character/National n (n<=$maxvarchar)	VARCHAR(n) [4]
Character/National n (n>$maxvarchar)	TEXT
Character(38)	UUID [3]
NUMBER	
Integer 64 bit	BIGINT
Integer 32 bit	INTEGER
Short integer	SMALLINT
Number 0..14dp	NUMERIC(15[1] ,0..14)
Short number 0/2dp	NUMERIC(15[1],0/2)
Number floating dp	DOUBLE PRECISION
DATE/TIME	
Short date (all subtypes)	DATE
Short time	TIME /TIMETZ*
Datetime (all subtypes)	TIMESTAMP /TIMESTAMPTZ*
OTHER	
Boolean	BOOLEAN
Sequence	SERIAL/INTEGER [2]
Picture	BYTEA
List	BYTEA
Row	BYTEA
Object	BYTEA
Binary	BYTEA
Item reference	BYTEA

[1] Numeric precision for Number (dp) columns uses the value of $numericprecision.
[2] The mapping used for the Omnis Sequence type depends on the value of $sequencetoint.
[3] This mapping occurs only if $char38touuid is set to kTrue
[4] Use the $addcustomtype() method to add additional mappings, e.g. for XML and JSON
*Time zone data types are used when session.$usetimezone is set to kTrue

PostgreSQL to Omnis

PostgreSQL Data Type	Description	Om
NUMBER		
INT2/SMALLINT	-32768 to +32767	Int
INT/INT4/INTEGER	-2147483648 to +2147483647	Int
INT8/BIGINT	-2^63 to +2^63-1	Int
SERIAL	1 to 4294967296	Int
SERIAL8/BIGSERIAL	1 to 2^64	Int
FLOAT4/FLOAT/REAL	1E-37 to 1E+37	Nu
DOUBLE/FLOAT8	1E-307 to 1E+308	Nu
NUMERIC	Numbers with max precision 1000	Nu
MONEY	-21474836.48 to +21474836.47	Nu
DATE/TIME		
DATE	Dates only	Sho
TIMESTAMP/TIME	Timestamp/time without time zone	Dat
TIMESTAMPTZ/TIMETZ	Timestamp/time with time zone	Cha
INTERVAL	Flexible format time interval	Cha
CHARACTER		
CHAR	Blank-padded characters with size limit	Cha
VARCHAR	Variable length characters with size limit	Cha
TEXT	Variable length characters with no size limit	Cha
JSON/JSONB	JavaScript Object Notation	Cha
BOOLEAN/BOOL	{'f', 'false', 'n', 'no', '0', 't','true','y','yes','1'}	Boo
CIDR, INET, MACADDR	Strings containing address information	Cha

PostgreSQL Data Type	Description	Om
UUID	Universally Unique Identifier	Ch
ENUM	Custom enumerated types	Ch
XML	Extensible Markup Language content	Ch
OTHERS (including, but not limited to)		
BYTEA, BIT, VARBIT, BOX, CIRCLE, POINT, LINE, PATH, POLYGON, LSEG		Bi

[1] DAM will map decimal values to the Omnis Number dp data type where column scale is <=14
[2] Supported in Studio 8.0.3 and later

PostgreSQL Troubleshooting

The following points may help in resolving programming issues encountered using PostgreSQL session and statement objects.

- $rpcparameters()
  When calling $rpcparameters(), the DAM uses defaults for the column precision and/or scale since this information is not provided by the pg_proc system table.
  For this reason, the API may report parameter-matching problems when calling certain functions and the list (passed to $rpcdefine()) may need to be manually coerced.

- Error Messages
  The following additional error messages may be returned via the session or statement $errortext property:

  - "Native error text could not be retrieved". No connection currently exists to the server or there is no message corresponding to the current error code.
  - "Unsupported client protocol version". The protocol version reported by the client API is too low. The DAM cannot use this version and you should upgrade to a newer version of the client library. Use the PostgreSQL access library supplied with Omnis Studio.
  - "Client or interface function not available". The most likely cause of this error is that the client library (or one of its dependencies) was not found and has not been loaded. Can also occur if the client library being used does not provide a required interface function.
  - "server closed the connection unexpectedly. This probably means the server terminated abnormally before or while processing the request." This error can occur when logging on with a username other than "postgres". The client library uses the username for the database name unless the database name is specified. Set $database to the required database name (e.g. "postgres") and try again.
  - "SCRAM authentication requires libpq version 10 or above" or "authentication method 10 not supported".
    The PostgreSQL client library shipped with Omnis Studio does not support the requested authentication method (scram-sha-256) which was introduced in PostgreSQL v10. For connection to PostgreSQL version 10 and later, we have a technote that discusses the replacement of the PostgreSQL Client library: TNSQ0031. The relevant client library and any OpenSSL dependencies can be copied from the PostgreSQL server installation.

- Linux Terminal Messages. On Linux, NOTICE and/or WARNING messages are sent to stderr, (normally the terminal window behind Omnis). To avoid these, refer to the server configuration parameter: client_min_messages and set this to a higher level.

- Chunking and Batch Fetching. Chunking of large character/binary data is not handled by DAMPGSQL but is handled automatically by the API. Such data is effectively returned to the DAM as single chunks. $lobthreshold , $lobchunksize and $blobsize therefore have no effect.

Batch fetching of data is also not handled by DAMPGSQL. The API automatically manages transfer of the data and presents the DAM with the entire result set. Hence setting $batchsize has no effect.

Further troubleshooting notes, "how-tos" and tips can be found on the Omnis website at: https://www.omnis.net/developers/resources/technotes/

SQLite

This section contains the additional information you need to access a SQLite database, a very popular database which is embedded into a whole range of applications on desktop and mobile devices. The code for SQLite is in the public domain and is thus free for use for any purpose, commercial or private.

SQLite implements a self-contained, server-less, zero-configuration, transactional SQL database engine. Unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads and writes directly to a disk file which can contain multiple tables, indices, triggers, and views. For more information about SQLite and to download it, please go to the website: www.sqlite.org (portions of this text are taken from the SQLite website).

This section contains the additional information you need to access a SQLite database, including server-specific programming, trouble-shooting and data type mapping to and from the database. For additional information on changes to the SQLite DAM, refer to the readme file which accompanies your Omnis download.

Server-specific Programming

Logging on to SQLite

To connect using the SQLite DAM, create an object variable of subtype "SQLITESESS".

You connect to a SQLite data file using the $logon() method. The hostname parameter should be the full path to the data file.

SQLite does not require a username or password, but you can specify a session name that will appear in the SQL Browser and in the Notation Inspector under *$sessions*.

SQLite expects a DOS-style pathname under Windows and an absolute POSIX-style path under macOS and Linux. For example:

```
Do mySession.$logon('C:\mydata\mydatafile.db','','','session1') Returns #F ## on Windows

Do mySession.$logon('/Users/MyUser/mydatafile.db','','','session1') Returns #F ## on macOS / Linux
```

Additionally, you can force SQLite to create the specified data file if it does not exist. To do this, set the $opencreate session property to kTrue before logging on.

To open a read-only connection, set the $readonly session property to kTrue before logging on. (It is not possible to create a data file if the connection is read-only).

If the hostname is ":memory:", then a private, temporary in-memory database is created for the connection. This in-memory database will be deleted when the database connection is closed. Filenames beginning with ":" should be considered reserved for future SQLite extensions and avoided to remove ambiguity. For in-memory databases, $version will be set to ":memory:" following $logon().

If the hostname is an empty string, then a private, temporary on-disk database will be created. This private database will be automatically deleted as soon as the database connection is closed. For temporary databases, $version will be set to ":temporary:" following $logon().

For standard data file connections, $version is read directly from the file header information and reflects the file format version that the data file supports.

Transaction Support

SQLite supports both automatic and manual SQL transactions.

To invoke manual transaction mode, the $transactionmode session property should be set to kSessionTranManual.

In this mode you must commence each transaction by calling the $begin() session method and terminating each transaction either by calling $commit(), $rollback(), by switching back to kSessionTranAutomatic or by logging off.

The SQL text that is submitted each time $begin() is called may be augmented using the $transactiontype session property as shown below. The different transaction types affect the way in which SQLite acquires row locks on tables:

$transactiontype	Resulting SQL text	Meaning
kSQLiteTranDeferred	BEGIN	No locks are acquired on the database until the database is
kSQLiteTranExclusive	BEGIN EXCLUSIVE	EXCLUSIVE locks are acquired on all databases as soon as th
kSQLiteTranImmediate	BEGIN IMMEDIATE	RESERVED locks are acquired on all databases as soon as th

The $commit() and $rollback() methods, invoke the COMMIT and ROLLBACK commands respectively.

$commitmode and $rollbackmode are set to kSessionCommitClose and kSessionRollbackClose respectively for the SQLite DAM. Statement objects are closed upon $commit() / $rollback(). Any pending result set is discarded and the statement is returned to its prepared state ready for re-execution if desired.

Incremental BLOB I/O

SQLite supports incremental Input/Output to BLOB columns in database tables. This means that you effectively bind a placeholder for the BLOB at bind time, then write the data to it later. Similarly, you can open a handle to a BLOB which already exists in the database and read/modify its contents without the need to perform a SELECT statement.

To create a placeholder for a BLOB, you should bind the binary variable inside the SQL statement as normal, but set its contents to #NULL. On execution, this creates a *zero-blob* of size *$blobsize*- bytes padded with zeros.

The session object provides several methods for accessing and modifying BLOBs:

- **$blobopen()**
  Opens a handle to a BLOB column, identified by its database name, table name, column name and row number, optionally as read-only
- **$blobclose()**
  Closes a BLOB handle; you have to close any BLOB handles opened during the session, but any handles left open when the session ends are closed automatically
- **$blobcloseall()**
  Closes all BLOB handles
- **$blobbytes()**
  Returns the size in bytes that was allocated to a BLOB column when it was created
- **$blobhandles()**
  Returns a list of all open BLOB handles including the corresponding database, table name, column name and row number
- **$blobreopen()**
  Moves a BLOB handle to a new row within the same table
- **$blobwrite()**
  Writes binary data to a BLOB column
- **$blobread()**
  Reads binary data from a BLOB column

See the Session Methods section for more details and syntax for these methods.

Calculated Columns, Functions and Sub-Queries

A column returned from SELECT statement that is the result of an expression or sub-query cannot be described automatically by the DAM because there is no corresponding entry in the sqlite_master table. In this situation, you can provide a “hint” using a column alias name containing the intended SQL data type, for example:

```
select ’2014-07-27’ as mydate from table
select substr(col1,1,2)||':'||substr(col1,3,2) as id_char from test
select 1 as boolval, col5+3 as intval from test
```

Other data types recognised include; “timestamp, time, national, tinyint, serial, sequence, dec, float”. When used the emulate OmnisSQL, the DAM also recognises various OmnisSQL function names such as upper(), lower(), ascii(), charindex(), length(), mod(), round(), cos(), sin(), etc. The DAM also recognises literal numeric and integer values, so there is no need to provide alias names for such columns, e.g.

```
select 3.14159, 33*3, col3+6, sin(0.6), length(col1) from test
```

Note that there is currently no way to specify the data sub-type when using aliased column names, hence Character data will be *Character 100000000* and numeric data will be *Number floating dp*. If the data type of a calculated column cannot be determined, it will be fetched as binary.

Similarly, result columns generated using *operators* have no SQLite “type affinity”. Using the UNION operator for example, it is necessary to CAST the entire operation:

```
select CAST(amount as FLOAT) as myfloat from (select 2.35 as amount
UNION ALL
select 2.46 as amount)
```

To avoid using column alias names, you can pre-define the fetch list or table instance using a schema, in which case the fetched data will be coersed into the required column types.

Additional Functions

In order to better support Omnis SQL emulation, the SQLite DAM supports the following scalar functions in addition to the SQLite core functions:

Function	Description
acos()	Angle in radians, the cosine of which is the specified number
ascii()	ASCII character corresponding to an integer between 0 and 255, inclusive
asin()	Angle in radians whose sine is the specified number
atan()	The angle in radians whose tangent is the specified number
atan2()	The angle in radians whose tangent is one number divided by another number
charindex()	The starting character position of one string in a second string
chr()	ASCII character corresponding to an integer between 0 and 255, inclusive
con()	Returns the concatenation of zero or more string arguments.
cos()	Cosine of an angle
dim()	Increments a date string by some number of months
dtcy()	A string containing the year and century of a date string
dtd()	A string containing the day part of a date string or a number representing the day of the month, depending on context
dtm()	A string containing the month part of a date string or a number representing the month of the year, depending on context
dtw()	A string containing the day of the week part of a date string or a number representing the day of the week, depending on context
dty()	A string containing the year part of a date string or a number representing the year, depending on context
exp()	exponential value of a number
Initcap()	Transforms string by capitalizing the initial letter of each word in the string and lowercasing every other letter
len()	Synonym for length(). Number of characters in a string
log()	Natural logarithm of a number
log10()	Base 10 logarithm of number
mod()	Modulus of a number given another number
pos()	Position of substring with a string
power()	The value of a number raised to the power of another number
sin()	Sine of an angle
sqrt()	Square root of a number
string()	Concatenates some number of strings into a string. Synonym for con()
tan()	Tangent of an angle
todate()	Converts a date string or number to a date value using a format string. Refer to the corresponding Omnis function for details.

SQLite Encryption

As of Studio 8.0.3, the SQLite DAM supports native datafile *encryption*. When enabled, all data written to the SQLite datafile is encrypted and can only be read and decrypted using the SQLite DAM with the appropriate encryption key.

Encryption is enabled by setting the session object $encryptkey property before logging on to the SQLite datafile. $encryptkey accepts a string of hexadecimal characters. The string should be of even length and should be no longer than 32 characters. The key value will be truncated if it does not meet either of these criteria. The accepted key value is then used to seed an internal private key which is subsequently used by all statement objects belonging to that session object.

To create a new encrypted datafile, the $opencreate property should also be set to kTrue before logging on. For example:

```
Do sessObj.$opencreate.$assign(kTrue)   ## create a new datafile if it does not exist
Do sessObj.$encryptkey.$assign('1a2b3c4d5e6f') Returns #F
Do sessObj.$logon('/Users/user1/Desktop/sqlite.db','','','session1') Returns #F
```

Once encrypted, $logon() will fail unless the correct $encryptkey is supplied. $encryptkey will be ignored (cleared) if the DAM detects a connection to a non-encrypted datafile. Please note that you cannot change the $encryptkey property while the DAM is logged on. Errors encountered during assignment of $encryptkey are written to session.$nativeerrorcode and session.$nativeerrortext.

The DAM provides two session methods that facilitate encryption/decryption of existing SQLite datafiles:

- $encrypt(filename)
  opens a non-encrypted datafile and encrypts it using the $encryptkey. A backup copy of the non-encrypted datafile is created at the file location named *filename.bak*

- $decrypt(filename)
  opens a previously encrypted datafile and decrypts it using the $encryptkey. A backup copy of the encrypted datafile is created at the file location named *filename.bak*

$encrypt() and $decrypt() return kTrue on success but will fail, unless the DAM is logged off, if the process cannot get exclusive read/write access to the specified datafile or if *filename.bak* already exists and cannot be overwritten. Once encrypted, connection via third-party tools should be avoided as this may result in undefined behaviour and cause datafile corruption.

Important note: Your attention is drawn to the terms of the Omnis End User License Agreement and to the following excerpt pertaining to the use of this encryption mechanism and subsequent loss of data and/or of the encryption key(s):

Omnis Software disclaims any responsibility for or liability related to the use of this software. IN NO EVENT WILL OMNIS SOFTWARE BE LIABLE FOR ANY INDIRECT, PUNITIVE, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES HOWEVER THEY MAY ARISE AND EVEN IF OMNIS SOFTWARE HAS BEEN PREVIOUSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Session Properties

Property	Description
$blobsize	The default value for $blobsize is set at 32KB for the SQLite DAM since this property is used routinely when creating empty BLOB columns for use with incremental input/output methods.
$encryptkey	Enables encryption. Accepts a string of hexadecimal characters of even length and no longer than 32 chars. The key value is truncated if it does not meet either of these criteria
$nullbinary	If kTrue, null-valued binary bind variables are inserted as NULL. If kFalse (the default) they are inserted as zero-blobs of length $blobsize (see Incremental BLOB I/O above). (Studio 8.1.6 and later)
$opencreate	If kTrue, the data file specified at $logon() will be created if it does not exist; in this case, the datafile is encrypted. If kFalse (the default), an error will be generated if the data file is not found.
$readonly	If kTrue, the connection will be opened in read-only mode. All attempts to write to the data file will fail with an error. If kFalse (the default), read and write operations are permitted.
$transactiontype	Specifies the locking behavior for manual transactions. This is one of the following constants: kSQLiteTranImmediate, kSQLiteTranExclusive or kSQLiteTranDeferred (the default).

Session Methods

Method	Description
$blobbytes()	$blobbytes(iBlobHandle) returns the size in bytes that was allocated to a BLOB column when it was cre
$blobclose()	$blobclose(iBlobHandle) closes a BLOB handle. You should close any BLOB handles opened during the open when the session ends are closed automatically however. Always returns kTrue
$blobcloseall()	$blobcloseall() closes all BLOB handles. This method always returns kTrue
$blobhandles()	$blobhandles(lHandleList) returns a list of all BLOB handles including their corresponding database, ta and row numbers. Aborted/invalid handles are shown with a row number set to zero. Returns kTrue on
$blobopen()	$blobopen(cDatabase, cTable, cColumn, iRow [,bReadOnly]) opens a handle to a BLOB column, identifie table name, column name and row number, optionally as read-only. Returns a BLOB handle on success SQLite DAM numbers BLOB handles incrementally starting from 1001
$blobread()	$blobread(iBlobHandle, xBinary [,iSize ,iOffset]) reads binary data from a BLOB column into the supplied omitted, the value of $blobsize is assumed
$blobreopen()	$blobreopen(iBlobHandle, iRow) moves a BLOB handle to a new row within the same table. If iRow exce the table, this invalidates the handle. Only the row number can be modified. To change the database, ta a new handle should be opened
$blobwrite()	$blobwrite() writes binary data to a BLOB column
$encrypt()	$encrypt(cFilename) opens a non-encrypted datafile and encrypts it using the $encryptkey. A backup c datafile is created at the file location named *filename.bak*

Method	Description
$decrypt()	$decrypt(cFilename) opens a previously encrypted datafile and decrypts it using the $encryptkey. A bac datafile is created at the file location named *filename.bak*
$insert_list()	$insert_list(cTable, lListData) inserts each row from lListData into cTable. The database must be logged- already exist. It is also assumed that the list definition is compatible with the table definition (Studio 10 a
$lastrowid()	$lastrowid() returns the rowid of the most recent successful INSERT into the database from the current
$rowsmodified()	$rowsmodified() returns the total number of database table rows that have been affected by INSERT, UP operations since the connection was opened (includes all statement objects)

Data Type Mapping

Omnis to SQLite

The SQLite DAM creates custom data types in order to preserve information about Omnis subtypes, notably: DATE(n) as well as PICTURE, LIST, ROW, OBJECT and OBJECTREF. There are also single mappings for Omnis Character and National data, implying that CHAR(n) and NCHAR(n) can store up to the maximum field length supported by Omnis (10000000 characters). This is contrary to other relational databases which impose a fixed size on such columns.

Although this greatly improves compatibility between Omnis and SQLite, if portability of the data file is of concern, then it may be preferable to avoid using $createnames() / $coltext() in favor of manual statements that use standard SQL types, e.g. VARCHAR(n), DATE, TEXT and BLOB.

Omnis Data Type	SQLite Data Type
CHARACTER	
Character n	CHAR(n)
National n	NCHAR(n)
NUMBER	
Integer 64 bit	BIGINT
Integer 32 bit	INTEGER
Short integer	TINYINT UNSIGNED
Number 0..14dp	NUMERIC(15, 0..14)
Short number 0/2dp	NUMERIC(9, 0/2)
Number floating dp	FLOAT
DATE/TIME	
Short date 1900..1999	DATE(1900)
Short date 1980..2079	DATE(1980)
Short date 2000..2099	DATE(2000)
Short time	TIME
Datetime (all subtypes)	TIMESTAMP
OTHER	
Boolean	BIT
Sequence	INTEGER PRIMARY KEY (auto increments when inserted as NULL)
Picture	PICTURE
List	LIST
Row	ROW
Object	OBJECT
Object reference	OBJECTREF
Binary / other	BINARY

SQLite to Omnis

The SQLite DAM recognises several additional SQL data types in order to maximise compatibility with externally generated data files as well as those generated by Omnis.

SQLite Data Type	Omnis Data Type
NUMBER	
TINYINT UNSIGNED	Short integer 0..255
TINYINT, INT, SMALLINT, INTEGER	Integer 32 bit
SEQUENCE, INT AUTO INCREMENT	Sequence
BIGINT	Integer 64 bit
FLOAT, REAL, DOUBLE	Number floating dp

SQLite Data Type	Omnis Data Type
NUMERIC(p,s), DEC(p,s), DECIMAL (p,s)	Short number s dp (p <=9, s=0 or 2) Number s dp (p <= 15) Number floating dp (p > 15)
DATE/TIME	
DATE(1900)	Short date 1900..1999
DATE(1980)	Short date 1980..2079
DATE(2000)	Short date 2000..2099
DATE, TIMESTAMP, TIME	Date Time (#FDT)
CHARACTER	
CHAR, VARCHAR, TEXT, CLOB,	Character
NCHAR, NVARCHAR, NATIONAL	National
OTHERS	
BOOLEAN, BOOL, BIT	Boolean
PICTURE	Picture
LIST	List
ROW	Row
OBJECT	Object
OBJECTREF	Object reference
BINARY / other	Binary

Troubleshooting

The following points may help in resolving programming issues encountered using SQLite session and statement objects.

For additional updated troubleshooting issues, refer to the readme file which accompanies the installation media.

For a detailed explanation of the SQL syntax supported by SQLite, please refer the SQLite website: www.sqlite.org

- SQLite does not currently support dynamic creation of SQL stored procedures or functions. The associated methods; $rpcprocedures(), $rpcparameters() & $rpc() therefore return kFalse.
- The SQLite API handles the transfer of binary data automatically. The $blobsize, $lobchunksize and $lobthreshold properties are therefore ignored.
- For performance reasons, journaling mode is set to PERSIST for the SQLite DAM. For optimum performance, especially on Linux it may be desirable to turn off journaling, ("PRAGMA journal_mode = OFF"). **Note:** in this mode however it will not be possible to rollback manual transactions.
- You may experience slow performance during certain INSERT operations. Each INSERT and UPDATE operation is normally committed to the disk drive so as to preserve integrity of the data in the event of a crash or power failure. Executing "PRAGMA synchronous=OFF" tells SQLite not to wait for data to reach the disk surface between writes which results in much faster performance. This risks data loss or corruption in the event of a crash however. Alternatively, you can use manual transaction mode (kSessionTranManual) to commit several INSERT operations at once.
- A column returned from SELECT statement that is the result of an expression or sub-query may require an additional column name alias containing the intended data type. Refer to *Calculated Columns and Sub-Queries* above for further details.

Oracle

This section contains additional information you need to access an Oracle database, including server-specific programming, PL/SQL, data type mapping to and from Oracle, as well as troubleshooting. For general information about logging on and managing your database using the Omnis SQL Browser, refer to the earlier parts of this manual.

Properties and Methods

In addition to the "base" properties and methods described in the *SQL Programming* chapter, the Oracle DAM provides the following additional features.

Session Methods

Method	Description
$proxyas()	SessionObj.$proxyas(cUsername [,cPassword, lRoles]). Allows the supplied user to connect to Oracle throu current connection, which must already be logged-on. The session then acquires the roles and privileges associated with that user. An additional list of roles to be used with the proxy session can also be supplied required. The list should consist of a single column of type Character. The password should be supplied if t proxy requires authentication by password. $proxyas() can be called repeatedly with different usernames required, in which case the current proxy is implicitly terminated before the new proxy is established.
$endproxy()	SessionObj.$endproxy(). Explicitly tests for and terminates a proxy session if one exists, returning the sessi and privileges back to those of the primary connection.
$getnames()	SessionObj.$getnames(&list). Retrieves a list of TNS names defined in the local tnsnames.ora file together their connection attributes supplied as sub-rows. Returns kTrue on success or kFalse if the tnsnames.ora f cannot be read or parsed (tnsnames.ora is located using the TNS_ADMIN environment variable). Requires 10.2 revision 31232 or later.

Session Properties

Property	Description
$authmode	Oracle 8 or above: Specifies the authentication mode to be used with the connection. By d sufficient privileges however, kAuthSysOper or kAuthSysDba can be supplied. $authmode
$binaryfloat	If kTrue, Omnis Number floating dp columns are mapped to BINARY_FLOAT. If kFalse (the
$binarytoblob	If set to kTrue, Omnis binary fields will be mapped to the BLOB data type. This affects inser by $createnames().When set to kFalse, the DAM maps binary fields to the Oracle LONG RA property is read-only for DAMORA7 and defaults to kTrue for DAMORA8.
$booltonum	If set to kTrue, Omnis Boolean fields will be mapped to the Oracle NUMBER(1,0) data type. written as 1 and 0, respectively. $createnames() will return NUMBER(1,0) as opposed to VAR old behavior is retained.
$credentials	Specifies the type of credentials to be used for establishing the connection. Valid modes a -Authentication via username and password (kCredRDBMS) -Authentication using the Wi proxy connection, please refer to the $proxyas() method.
$datetype	The Oracle data type used to map Omnis Datetime fields. Also used for input bind variabl
$emptyasnull	When kTrue, empty Omnis strings are inserted as NULL. When kFalse, they are inserted as
$internalcharmapping	If set to kFalse, conversion of non-Unicode character data to and from the Omnis characte $charmap=kSessionCharMapOmnis or kSessionCharMapTable, thus allowing custom chara characters if required. Default setting is kTrue.
$longchartoclob	If set to kTrue, Omnis large character fields > $maxvarchar2 in length will be mapped to th updates as well as the text returned by $createnames(). When set to kFalse, the DAM map data type (Oracle7 behaviour). This property is read-only for DAMORA7 and defaults to kTr
$maxvarchar2	Default is 2000. Specifies the length above which Omnis character columns will be mapp 8. The max value is 4000 for DAMORA8 and 2000 for DAMORA7. Setting $maxvarchar2 to to the LONG/CLOB data type.
$nationaltonclob	Oracle 8 or above: is used to alter the default mapping of Omnis Character and National ty National fields with a subtype greater than $maxvarchar2 are mapped to the NCLOB data only National fields with a subtype >$maxvarchar2 are mapped as NCLOBs. Character field as non-Unicode CLOBs. Character fields mapped in this way are subject to data loss/trunca characters.
$nationaltonvarchar	Only available in the Unicode DAM, $nationaltonvarchar is used to alter the default mappi By default, Omnis Character and National fields with a subtype <= $maxvarchar2 are mapp $nationaltonvarchar to kTrue only National fields with a subtype <= $maxvarchar2 are map subtype <= $maxvarchar2 are mapped as non-Unicode VARCHAR2 columns. Character fiel loss/truncation where such fields contain Unicode characters. Please note VARCHAR2 and bytes. Hence NVARCHAR2 columns are limited to 2000 UTF-16 characters.
$nativewarncode	A warning code issued by the clientware in response to the last session method to be exec
$nativewarntext	A warning message issued by the clientware in response to the last session method to be
$newpassword	If set, the DAM will attempt to change the password during $logon(). Intended to allow exp be used in the general case. If successful, the logon will proceed as normal, the $password property will be updated an changing the password, the existing username and password should be passed via the $lo
$nullasempty	Default value kFalse. If kTrue Null values are converted to empty values when fetched fro
$timezone	Timezone used to modify incoming/outgoing timestamps that contain time zone attribut session at logon.
$trailingspaces	Default value kFalse. If kFalse is specified any trailing spaces on character data being inser spaces are kept.

Property	Description
$truetext & $falsetext	Studio 5.0 and later. Contain the text that will be inserted for Boolean bind variables. Wher the Omnis localization datafile, these properties now permit localization-independent valu backwards compatibility, default values are taken from Omnisloc.df1. Affects the text retur buffers used to insert data.
$querytimeout	Studio 4.3.2/5.0.1 and later. This is the timeout in seconds for any statement executed via $e detect Unix network hangs, this property has no effect for Win32 and macOS. When a time and a re-connect is necessary. Default value is 10 seconds.

Statement Methods

Methods	Description
$plsql()	StatementObj.$plsql(cPLSQLtext[,iColNo…]). This function should be used instead of $prepare() w procedures that contain bound OUT or IN/OUT parameters.
$prepareforupdate()	StatementObj.$prepareforupdate(vTableDef,cTablename[,cWhere]) creates and prepares a 'select specific use with positioned updates and deletes. vTableDef is a row or list variable defined with c
$posdelete()	StatementObj.$posdelete(oStatement) deletes a row positioned by the specified statement obje statement object prepared previously using $prepareforupdate(), and executed.
$posupdate()	StatementObj.$posupdate(oStatement,wRow) updates a row positioned by the specified statem statement object prepared previously using $prepareforupdate(), and executed.

Statement Properties

Property	Description
$nativewarncode	A warning code issued by the clientware in response to the last statement method to be executed.
$nativewarntext	A warning message issued by the clientware in response to the last statement method to be executed
$plsqlarraysize	When retrieving data into an Omnis list via PlSql, the number of rows that will be fetched is not known Historically, the DAM reserved a pre-determined buffer size of 32512 bytes per list column to be fetched rows fetched * column size (in bytes) exceeds this limit for a given column, ORA-06513 is returned. The according to $plsqlarraysize (default value 32512), thus the buffer size can be raised (or lowered) as req accommodate the entire result set.

Connecting to your Database

To connect to your database, you need to create a session object with a subtype ORACLE8SESS (or the legacy subtype: ORACLE7SESS). In order to log on to the database using the SessObj.$logon() method, the hostname must contain a valid Oracle host alias previously generated by the Oracle client tools.

Mixing Unicode and Non-Unicode Data Types

This section summarises recent changes made to the Unicode Oracle Object DAM designed to enable insertion and retrieval of mixed ANSI and Unicode character types.

In the case of Oracle 8i and later, these data types are:

Data type	Description
CHAR	Fixed single-byte character data, limited to 2000 bytes.
NCHAR	Fixed multi-byte character data, limited to 2000 bytes. (1000 UCS-2 encoded characters)
VARCHAR2	Varying length, single-byte character data, limited to 4000 bytes.
NVARCHAR2	Varying length, multi-byte character data, limited to 4000 bytes. (2000 UCS-2 encoded characters)
CLOB	Character Large Object- single-byte character data.
NCLOB	National Character Large Object- multi-byte character data.
LONG	Varying length, single-byte character data, limited to 2GB. Supported for backward compatibility only.

By default, the Unicode Oracle DAM maps all Omnis character data to the NVARCHAR2 and NCLOB data types, dependent on the field length of the Omnis bind variable. However, the Oracle DAM provides session properties which affect the Omnis to Oracle data type mappings:

- $nationaltonvarchar
  If set to kTrue, Character and National data types are treated differently when being inserted to VARCHAR2 / NVARCHAR2 columns.
- $nationaltonclob
  If set to kTrue, large Character and National data types are treated differently when being inserted to CLOB / NCLOB columns.
- $maxvarchar2
  Sets the byte limit above which Omnis character fields will be mapped to CLOB/NCLOB data types as opposed to VARCHAR2 / NVARCHAR2 columns. The maximum value is 4000 bytes.
- $longchartoclob
  If set to kTrue (the default), Omnis large character fields > $maxvarchar2 in byte length will be mapped to the CLOB/NCLOB data type. If set to kFalse, the LONG data type is used.

Reading Unicode and Non-Unicode Data

The Oracle DAM automatically detects the data type of retrieved character columns and converts the data accordingly. There is no need to modify any properties in order to retrieve mixed ANSI and/or Unicode Data.

Insertion/Update of CHAR and VARCHAR2 data

To write short character data to ANSI columns it is necessary to set $nationaltonvarchar to kTrue. In this mode, Omnis Character fields will be mapped to VARCHAR2 and National fields will be mapped to NVARCHAR2.

When set to kFalse (the default), both Character and National types will be mapped to NVARCHAR2.

Insertion/Update of CLOB data

Where the Omnis field length exceeds $maxvarchar2, the DAM will map to either CLOB, NCLOB or LONG dependent on the value of the $nationaltonclob and $longchartoclob properties. To write long character data to ANSI CLOB columns, it is necessary to set $nationaltonclob to kTrue. In this mode, Omnis Character fields will be mapped to CLOB and National fields will be mapped to NCLOB. When set to kFalse (the default), both Character and National types with byte sizes exceeding $maxvarchar2 will be mapped to NCLOB.

Note that where Omnis fields are mapped to NCLOB columns, $maxvarchar2 is interpreted as the length in bytes. Thus when set to 4000, this mapping will be applied for Character and/or National fields with a field length > 2000 characters

Fetching Very Large Objects

The Oracle DAM has the ability to fetch very large objects (up to 2GB) directly to the local file system. Two new properties have been added to the session object:

- $filethreshold
  the file threshold which is initially set to 50MB
- $filedirectory
  the directory to receive the file which is initially set from the USERPROFILE environment variable on Windows or HOME on macOS and Linux

Any CLOB, NCLOB, BLOB or BFILE column which exceeds $filethreshold will now be fetched in chunks using $lobchunksize directly to $filedirectory. The filename used will conform to “colname_xxxxxx.BIN” where xxxxxx is a unique identifier (based on #CT). Any character data written to file will be converted to UTF8, otherwise raw data will be written. For BFILEs this means that changing the file extension later (e.g. from .BIN to .AVI) will result in a facsimile of the original file. When a VLOB is written to file, its filename is returned into the result list column. Since the result column was previously described as binary it is necessary to extract the filename using the utf8tochar() function, e.g. Calculate filename as utf8tochar(lResult.1.colLOB).

Using Worker objects to fetch VLOBs

Fetching VLOBs on the main thread can cause Omnis to pause while the data is being transferred. Therefore, for large transfers it may be preferable to SELECT and FETCH each VLOB using an Oracle worker object. The main thread is then free to continue and will be notified when the VLOB has been fetched.

Insertion/Update of LONG data

When $longchartoclob is set to kFalse, Omnis Character and National fields which would otherwise map to CLOB or NCLOB will be mapped to the LONG data type. Since Oracle tables may contain only one column of type LONG, this may lead to problems if not used judiciously.

PL/SQL

Prior to Studio 10.2, the Oracle DAM does not support the remote procedure call methods such as $rpc() which are described in the *SQL Programming* chapter. Server procedures can be executed via PL/SQL. The Oracle DAM fully supports Oracle PL/SQL; a procedural language that the server executes.

You create a PL/SQL script and send it to Oracle in a similar way as any SQL statement and the server executes it. The statement object method $plsql() should be used instead of $prepare() when you want to call server procedures that contain bound OUT or IN/OUT parameters. Any PL/SQL bind variables being passed to Oracle should be passed as Omnis bind variables i.e. @[Xvar]. When $plsql() is called with bind variables, the DAM will check to see whether any return values are present after execution and will return them back into the Omnis variables. This does not happen after a $prepare(). If you are creating stored procedures or executing a server procedure for which you do not expect a return value, $prepare() will be sufficient.

```
# lEmpId is local Integer 32-bit with initial value of 0
Do StatObj.$plsql("begin select empno into @[lEmpId] from scott.emp where ename = 'JONES'; end;")
Do StatObj.$execute()
If lEmpId <>7566
  # Incorrect value returned from procedure
End If
```

After the PL/SQL executes, the Omnis variable lEmpId has the value associated with the column id from the row with the name 'Jones'.

The Oracle DAM supports select tables returned through PL/SQL procedures. However Oracle can only return single column tables (arrays). The statement object method, $plsql(), has optional parameters that follow the cPLSQLtext parameter. These are necessary when calling server procedures that return single column tables. To bind a column of an Omnis list to the select table being returned requires a list variable to be bound in the SQL statement and the list column number to be passed as an additional parameter. If more than one select table is being returned, multiple lists will need to be bound and a column number parameter passed for each list, in the same order as the lists are bound. The bound lists do not have to be different lists, the same list can be bound more than once in the PL/SQL statement, but you must take care to specify a different column number for each occurrence of the bound list. If no column number parameter is passed for a bound list, the first column of the list is used by default.

Consider the following example table and PL/SQL package:

Table: accounts

idNUMBER(3,0)	nameNVARCHAR2(256)	balanceNUMBER(16,2)	limitNUMBER(16,2)
1	Bill	2000	2200
2	Sally	120	100
3	Bob	1000	190
4	John	1700	1500
5	Graham	3000	21087
6	Helen	2000	1860
7	Betty	9000	1490
8	Walter	25000	17200
9	Sarah	9100	10000

```
create or replace package test as
  type account_name is table of accounts.name%TYPE index by binary_integer;
  type account_balance is table of accounts.balance%TYPE index by binary_integer;
end test;
create or replace procedure credit(accnum IN number, amount IN number,pname out test.account_name, pbalance
is
  cursor c1 is select name,balance from accounts where balance > limit;
  row_count BINARY_INTEGER;
begin
  row_count := 1;
  update accounts set balance = balance + amount where id = accnum;
  open c1;
  LOOP
    FETCH c1 INTO pname(row_count),pbalance(row_count);
    row_count := row_count + 1;
    exit when c1%NOTFOUND;
  end LOOP;
  close c1;
end;

# Local variables:
# lName=Character 256, lBalance=Number2dp, lCreaditList=List
Do SessObj.$blobsize.$assign(32767) ## Prevents ORA-06505
Do lCreditlist.$define(lName,lBalance)
Do StatObj.$plsql('begin credit(2,490,@[lCreditlist], @[lCreditlist]); end;',1,2) Returns #F
Do StatObj.$execute() Returns #F
If Creditlist.$linecount<>6
  # Incorrect data returned from stored procedure

End If
```

In the above example, the same list has been bound twice, the first bind variable binds the first column of the list and the second bind variable binds the second column as defined by the second and third parameters of the $plsql() method.

After the PL/SQL procedure executes, lCreditList should contain 6 rows as follows:

name	balance
Sally	610.00
Bob	1000.00
John	1700.00
Helen	2000.00
Betty	9000.00
Walter	25000.00

The balance of account id 2 (Sally) has been increased by 490. The procedure returns details of those accounts where the balance column is greater than the limit column.

$rpc() Support

As of Studio 10.2 (rev 30204), there is support for $rpcprocedures(), $rpcparameters(), $rpcdefine() and $rpc(). $rpc() basically executes a PL/SQL begin... end statement block that calls the stored procedure or function. Operation is as described in the SQL Programming chapter with one exception. When bindng single-column SELECT tables as in the previous example, it is necessary to pass the required list column numbers along with the parameter definitions. To do this, the DAM uses column 5 of the list returned by $rpcparameters(). For example:

```
Do cStat.$rpcparameters('credit') Returns #F
Do procList.$define()
Do cStat.$fetch(procList,kFetchAll)     ## returns 4 rows
Do procList.3.5.$assign(1)   ## Assign the list column number to 1

Do procList.4.5.$assign(2)   ## Assign the list column number to 2
Do cSess.$rpcdefine('credit',procList) Returns #F

Do lCreditList.$define(lName,lBalance)
Do cStat.$rpc('credit',1,10,lCreditList,lCreditList) Returns #F
```

The additional values assigned to *procList* correspond to the column numbers that would otherwise be passed via the $plsql() method in the previous example.

You can also call a stored function using the $rpc() method and the return value will be written to the statement object's $rpcreturnvalue property. For example:

```
Begin statement
Sta: CREATE OR REPLACE FUNCTION test_function
Sta: RETURN VARCHAR2 IS
Sta: BEGIN
Sta: RETURN 'This is being returned from a function';
Sta: END test_function;
End statement

Do cStat.$execdirect() Returns #F
Do cStat.$rpcparameters('test_function') Returns #F
Do procList.$define()
Do cStat.$fetch(procList,kFetchAll)

Do cSess.$rpcdefine('test_function',procList) Returns #F
Do cStat.$rpc('test_function') Returns #F   ## now check the value of $rpcreturnvalue
```

Alternatively, could can invoke functions in SQL statements. For example:

```
Do cStat.$execdirect('select test_function() from dual') Returns #F

Do cStat.$fetchinto(lCharVar)
```

$rpc() is limited to calling a single stored procedure or function. To execute more complex PL/SQL constructs, you can continue to use the $plsql() method.

Positioned Updates and Deletes

You can use positioned updates and deletes to update and delete specific rows from the select table you are fetching. To enable positioned updates and deletes the statement object method, $prepareforupdate(), should be used. This method creates and prepares a 'select for update' statement for specific use with positioned updates and deletes. A 'select for update' SQL statement can be prepared using the $prepare() method. However, it will not store the current ROWID in the statement object.

You can use $prepareforupdate() in conjunction with $posupdate() and $posdelete() to update or delete a row, which is determined by the current row in the specified statement. The 'select for update' statement is built based on the parameters passed. The columns will be derived from the list or row passed as the first parameter. If the cTablename parameter is omitted, the name of the list/row is assumed to be the name of the table. A SQL WHERE clause is appended to the select statement if it has been specified. After this statement has been executed the last row fetched will be seen to be the current row. If the statement does not perform a fetch, there will not be a current row.

Note: For all position update and delete functionality the transaction mode must be kSessionTranManual.

```
# lEmpSt,lEmpUpSt,lEmpDelSt are Statement Object instances derived from the same session
# lTableName is a Character(32) variable
# iTableList and lTempList are list variables defined from schema class scPos1
# lDataRow is a row defined from schema class scPos1
# Fetch the row
Calculate lTableName as $classes.scPos1.$servertablename()
Do lEmpSt.$prepareforupdate(iTableList,lTableName)
Do lEmpSt.$execute()
Do lEmpSt.$fetch(lDataRow,1)
# Update this row
Calculate lDataRow.vala as 5 ## change the value of one of the columns
Do lEmpUpSt.$posupdate(lEmpSt,lDataRow)
Do lEmpUpSt.$execute()
# Fetch the next row
Do lEmpSt.$fetch(lTempList,2)
# delete this row
Do lEmpDelSt.$posdelete(lEmpSt)

Do lEmpDelSt.$execute()
```

Oracle 8 Data types

The Oracle8 DAM (DAMORA8) supercedes the older Oracle7 DAM (DAMORA7, which is no longer in development). DAMORA8 has been specifically written to connect to Oracle8 (and later) databases but can be used against an Oracle7 server using the recommended clientware. In this case, the DAM will encounter restrictions as described below, mainly concerned with data type mapping of large objects.

Large Objects (LOBS))

CLOBs, NCLOBs and BLOBs are data types introduced for Oracle 8 that deal with large objects. Internal LOBs (BLOBs, CLOBs, NCLOBs) are stored in the database tablespaces in a way that optimizes space and provides efficient access. These LOBs have the full transactional support of the database server. The maximum length of a LOB/FILE is 4 gigabytes. Internal LOBs have copy semantics. Thus, if a LOB in one row is copied to a LOB in another row, the actual LOB value is copied, and a new LOB locator is created for the copied LOB.

The Oracle8 DAM uses locators to point to the data of a LOB or FILE. These locators are invisible as the DAM performs operations on the locator to insert, update, delete and fetch the values. This means that you are only ever dealing with the values of the LOBs and not the locators.

You can work with the locators rather than just the values using PL/SQL in conjunction with the dbms_lob package. Further information can be found in the Oracle8i Supplied Packages Reference.

External LOBs (FILEs) are large data objects stored in the server's operating system files outside the database tablespace. FILE functionality is read-only. Oracle currently supports only binary files (BFILEs). The Oracle8 DAM uses locators to point to the data of a FILE. The FILE locator will be invisible, as the DAM will return the value of the external file and not the locator when performing transactions with the BFILE data type. Even though the BFILE data type is read-only you can insert a directory alias and filename into the column. These values are assigned to a single Omnis binary variable and separated by the '&' symbol. The DAM will assign these values to the locator so that when a fetch is performed on the locator the binary representation of the external file corresponding to the alias and filename will be returned. An example is shown below.

```
# A Directory alias needs to be created on the server that points
# to an OS folder
myStat.$execdirect("create or replace directory sound as 'c:\bfiles'")
# BFILE1 and BFILE2 are Omnis variables of type Binary.
# The variable is calculated as
# '<DirectoryAlias>&<Filename>'.
Calculate BFILE1 as 'sound&wav2.wav'
Calculate BFILE2 as 'sound&wav3.wav'
myStat.$execdirect('insert into bfiletest values(1,@[BFILE1],@[BFILE2])')
# You can now select the data back and this time you can receive the binary representation of the file.
myStat.$execdirect('select * from bfiletest')
myStat.$fetch(myRow)
# The values contained in col2 and col3 of myRow can now be
# written to the local drive using the Omnis Fileops commands.
Calculate File as 'c:\windows\desktop\wavtest.wav'
Do Fileops.$createfile(File) Returns lErr
Do Fileops.$openfile(File) Returns lErr
Do Fileops.$writefile(myRow.2) Returns lErr

Do Fileops.$closefile() Returns lErr
```

Ref Cursor Data Types

The REF CURSOR is an Oracle 8 data type that is used to point to a set of results from a multi-row query. When executing a multi-row query, Oracle opens an unnamed work area that stores processing information. To access the information, you can use a variable of type REF CURSOR, which points to the work area. To create cursor variables, you define a REF CURSOR type and then declare cursor variables of that type.

A REF CURSOR type can be defined in an Oracle Package object. For example:

```
create or replace package OmnisPackage
as
type cursorType is ref cursor;
end;
```

This data type can be used in other Oracle objects, such as procedures and functions in order to process result sets. An example of a Stored function follows:

```
create or replace function OmnisFunction return OmnisPackage.cursorType
as
l_cursor OmnisPackage.cursorType;
begin
open l_cursor for select * from scott.dept;
return l_cursor;
end;
```

An example of a stored procedure that uses the defined REF CURSOR type follows:

```
create or replace procedure OmnisProcedure
( p_cursor in out OmnisPackage.cursorType )
as
begin
open p_cursor for select ename, empno from scott.emp order by ename;
end;
```

Cursor variables are like pointers, which hold the memory location (address) of some item instead of the item itself. So, declaring a cursor variable creates a pointer, *not* an item.

REF CURSOR data types can be returned in three different ways; via a PL/SQL block, an Oracle Stored Function or an Oracle Stored Procedure. The REF CURSOR type is a pointer to a result set. The SQL statement that returns the REF CURSOR must be prepared using the $plsql() method. The Oracle 8 DAM maps the REF CURSOR type to an Omnis Oracle8 Statement Object*. The statement object will be created by the DAM and will belong to the same session object as the Statement object that prepared the initial SQL. It will have a $state of kStatementStateExecuted and, assuming that there is data in the result set, will have $resultspending set to kTrue. Therefore, the statement object will be in a 'Ready-For Fetch' state. Below are examples of the three ways to return and use a REF CURSOR in Omnis. The connection code and the creation of initial Statement Object (myStatement) have been removed for clarity.

*As of Studio 5.2, the Object variable used to return the REF CURSOR result set may instead be passed as an Object reference if preferred.

PL/SQL Block

The PL/SQL method does not require any SQL objects created on the server. All the PL/SQL code can be encapsulated in an Omnis Statement block.

```
# declare vars: cursor1 (Object), myList1 (List)
Begin statement
Sta: begin
Sta: OPEN @[cursor1] FOR SELECT * FROM scott.emp;
Sta: end;
End statement
If myStatement.$plsql()
  If myStatement.$execute()
    Do cursor1.$fetch(myList1,kFetchAll)
    # myList1 will contain the rows of the result set.
  Else
    OK message Error {[ myStatement.$nativeerrortext]}
  End If
Else
  OK message Error {[ myStatement.$nativeerrortext]}

End If
```

Stored Functions

Returning a REF CURSOR from a Stored Function requires an Oracle Stored Function on the database. The Function must have a return type that has been defined as a REF CURSOR. For this example we will assume that the example Oracle Stored Function described above has been created on the server.

```
# declare vars: cursor2 (Object), myList2 (List)
If myStatement.$plsql('begin @[cursor2] := OmnisFunction; end;')
  If myStatement.$execute()
    Do cursor2.$fetch(myList2,kFetchAll)
    # myList2 will contain the rows of the result set.
  Else
    OK message Error {[ myStatement.$nativeerrortext]}
  End If
Else
  OK message Error {[ myStatement.$nativeerrortext]}
End If
```

Stored Procedures

Returning a REF CURSOR from an OUT or IN OUT parameter of a Stored Procedure requires an Oracle Stored Procedure on the database. The Procedure must have an OUT or IN OUT parameter type that has been defined as a REF CURSOR. For this example we will assume that the example Oracle Stored Procedure described above has been created on the server.

```
# declare vars: cursor3 (Object), MyList3 (List)
If myStatement.$plsql('begin getemps(@[cursor3]); end;')
  If myStatement.$execute()
    Do cursor3.$fetch(myList3,kFetchAll)
    # myList3 will contain the rows of the result set.
  Else
    OK message Error {[ myStatement.$nativeerrortext]}
  End If
Else
  OK message Error {[ myStatement.$nativeerrortext]}
End If
```

Oracle 9i Data types

The Oracle 8 Object DAM includes support for data types added for Oracle 9i, namely the XML and URI data types. The XML data type lets you store native XML documents directly in an Oracle database and eliminates the need to parse the documents coming into and out of the database. Server specific properties and methods have been added to the DAM to support these enhanced database operations.

Oracle 9i also introduced several new Universal Resource Identifier (URI) types. These are used to identify resources such as Web content anywhere on the Web and can be used to point to data either internally or externally from the database itself. In addition to support for URIs, the Oracle DAM includes support for querying and other abstract functions provided for the URI types.

These changes were introduced with Omnis Studio version 3.2. The old-style, single threaded DAM (DORACLE8) can connect to Oracle 9i databases, but does not support the XML and URI data types.

XMLType

The XMLType is a system defined data type with predefined member functions to access XML data. You can perform the following tasks with XMLType:

- Create columns of XMLType and use XMLType member functions on instances of the type.
- Create PL/SQL functions and procedures, with XMLType as argument and return parameters.
- Store, index, and manipulate XML data in XMLType columns.

URIType

The URIType is an abstract object type that can store instances of HttpUriType or DBUriType**.** Universal Resource Indicator references can point to XML, HTML and custom internet content which can be located either locally within the database, externally to the database but local to the server or remotely across an internet or network connection.

DBUriType

The DBUriType can obtain data pointed to by a DataBaseUri-reference. A DBUri-Ref is a database relative URI that is a special case of the Uri-ref mechanism, where ref is guaranteed to work inside the context of a database and session. This ref is not a global ref like the HTTP URL, instead it is local ref (URL) within the database.

HttpUriType

The HttpUriType implements the HTTP protocol for accessing remote pages.

UriFactoryType

It is not possible to generate table columns using the UriFactoryType. Rather, this is a PL/SQL package containing factory methods that can be used to generate the appropriate instance of the Uri types without having to hard code the implementation in the program. Custom URI types can be defined and registered using this package.

For further information on the application of these data types, refer to the *Oracle9i Application Developer's Guide – XML*.

Retrieving XML and URI data

In Oracle9i version 9.1, it is not possible to directly SELECT data from columns defined using these types. Instead, the appropriate accessor functions should be used.

XMLType provides the extract(), getClobVal(), getStringVal(), and getNumberVal() functions for data query and retrieval. The extract() function has to be used in conjunction with one of the data type conversion functions, since it returns an object of type XMLType. The following example SQL statement can be used to extract an XML document from an XMLType column:

```
SELECT a.xmlcol.extract('*').getStringVal() AS mycol FROM mytable a
```

If required, the XPath expression parameter to the extract() function can be supplied using a character bind variable. For further information on the extract() function and the supported Xpath syntax, refer to the *Oracle9i Application Developer's Guide – XML*.

URITYPE and its derivatives provide the getClob(), getUrl() and getExternalUrl() functions for data retrieval. When getClob() is executed, the URL stored in the database column is read. The document pointed to by the URL is then accessed and returned as CLOB data which can be read into Omnis:

```
SELECT a.myuri.getclob() AS mycol FROM mytable a
```

The getUrl() and getExternalUrl() functions return the URL contained in the database column. (getExternalUrl() differs from getUrl in that it *escapes* the URL so that it better conforms to the URL specification):

```
SELECT a.myuri.geturl() AS myurl FROM uritest a
```

The getClob(), getUrl() and getExternalUrl() functions can be overridden when creating custom URI types as defined using the UriFactoryType. For information on the UriFactoryType, refer to the *Oracle9i Application Developer's Guide – XML*.

Inserting XML and URI data

To insert XML data into an XMLType column, the data must be a valid XML document or fragment. This is because XMLType validates the XML before storing it. A simple insert statement would be of the form:

```
INSERT INTO xmltable VALUES(…,sys.XMLType.createXML ('<DOC> </DOC>'),…)
```

If required, the XML text can be supplied using a character bind variable.

XML data can also be supplied as CLOB data by inlining a SELECT statement (or some other expression which returns a CLOB):

```
INSERT INTO xmltable SELECT id, sys.XMLType.createXML(myclob) FROM clobtable;
```

Inlining ensures that createXML() receives a CLOB field, which is not possible from Omnis since CLOBs are converted into Omnis character strings when fetched.

To insert a URL into a URIType column or one of its derivatives, use the createUri() function. For example:

```
INSERT INTO uritest VALUES (…,sys.httpUriType.createUri('http://www.omnis.net'),…)
```

If required, the URL can be supplied using an Omnis character bind variable.

Updating XML and URI data

In version 9.1, Oracle9i stores XMLType internally using the CLOB data type. Updates on CLOBs have to be performed on the entire column and for this reason, when updating an XMLType column, it is necessary to re-insert the XML data.

```
UPDATE xmltest
SET xmlcol = sys.XMLType.createXml(@[iXMLText]) WHERE IDcol = @[iRecordNum]
Similarly, with URITypes, updates are performed as follows:
UPDATE uritable
SET uricol = sys.httpUriType.createUri(@[iHTMLURL]) WHERE IDcol = @[iRecordNum]
```

Oracle Data Type Mapping

Omnis to Oracle

Omnis Data Type	ORACLE8 Data Type
CHARACTER	
[1]Character/National <= the value of $maxvarchar2 (default is 2000)	NVARCHAR2(n)
[1]Character/National > the value of $maxvarchar2 (default is 2000)	NCLOB
DATE/TIME	
Short date (all subtypes)	DATE
Short time	DATE
Date time (#FDT)	DATE
NUMBER	
Short integer (0 to 255)	NUMBER(3, 0)
Integer 64 bit	NUMBER(19,0)
Integer 32 bit	NUMBER(11, 0)
Short number 0dp	NUMBER(10, 0)
Short number 2dp	NUMBER(10, 2)
Number floating dp	FLOAT
Number 0..14dp	NUMBER(16, 0..14)
OTHER	
Boolean	[2]VARCHAR2(3)
Sequence	NUMBER(11, 0)
Picture	[3]BLOB
Binary	[3]BLOB
List	[3]BLOB
Row	[3]BLOB
Object	[3]BLOB
Item reference	[3]BLOB

[1] Dependant on the values of $nationaltonvarchar, $nationaltonclob and $maxvarchar2
[2] Dependant on the value of $booltonum
[3] Dependant on the value of $binarytoblob

Oracle to Omnis

Omnis Data Type	ORACLE8 Data Type
CHARACTER / BINARY	
CHAR / NCHAR	Character
VARCHAR2 / NVARCHAR2	Character
CLOB / NCLOB	Character
BLOB	Binary
BFILE	Binary
LONG	Character
RAW	Binary
LONG RAW	Binary
DATE/TIME	
DATE	Date time (#FDT)
NUMBER	
NUMBER(p,0) p>10	Integer 64 bit[1]

Omnis Data Type	ORACLE8 Data Type
NUMBER(p,s) p<=10 or s>0	Number floating dp
NUMBER (NUMBER(0,0))	Number floating dp
FLOAT	Number floating dp

[1] $fetch64bitints must be kTrue

Oracle Troubleshooting

- When performing transactions that use the new LOB data types the transaction mode must be set to either kSessionTranAutomatic or kSessionTranManual. This is because LOB and file locators cannot be used across transactions. The DAM performs functionality on these locators and when the transaction mode is either automatic or manual, the DAM can control when to commit the command, which would be after all the LOB functionality has been performed. When the transaction mode is server, Oracle commits (or rollbacks) after every statement and any LOB functionality performed by the DAM would result in an error.
- Oracle has a client character set and a server character set. If the two are not the same character set, Oracle will convert between the two. If a character in the client character set also exists in the server character set, Oracle will store that character on the server. If the character doesn't exist in the server character set, Oracle will do one of two things. Firstly, Oracle will see if there is a close-fit character. For example, if you are trying to insert the character 'à', but that particular character is not in the server character set, Oracle will store that character as an 'a' as it is a close-fit. If there is not a close-fit character, Oracle will store that character as an 'unknown' character on the server. This 'unknown' character is usually '¿'. If the client character set is the same as the server character set, no conversion takes place and all ASCII values of characters are preserved. On retrieval of characters Oracle will again convert but this time from the characters stored in the Oracle database to the client's character set being used to retrieve the text.
- As with inserting Oracle will convert a character in the server character set to the same character in the client character set if it exists. If not, it will try a 'best fit' solution before returning its 'unknown' character. To guarantee that Oracle doesn't convert the data in the database, the client character set should be the same as the server character set. It is possible to use Omnis' character maps to change some of the characters which aren't correct to ones that are preferred (Non-Unicode session objects only). This is useful when Oracle returns a 'best fit' character and not the original character.

Further troubleshooting notes, "how-tos" and tips can be found on the Omnis website at:

Sybase

This section contains the additional information you need to access a Sybase database, including server-specific programming, data type mapping to and from Sybase, as well as troubleshooting. For general information about logging on and managing your database using the Omnis SQL Browser, refer to the earlier parts of this manual.

Properties and Methods

In addition to the "base" properties and methods documented in the *SQL Programming* chapter, the Sybase DAM provides the following additional features.

Session Properties

Property	Description
$programname	The program name that is registered by Sybase at logon. The default is $clib().$name.
$logontimeout	The timeout in seconds for a logon. The default is 60 seconds. Set this to 0 for no timeout. Note that $failtimeout is kFalse.
$querytimeout	Timeout in seconds for a query. The default is 0 for no timeout. Note that a timeout is ignored if $fai
$failtimeout	Set to kTrue to raise an error if a timeout occurs. If kTrue and a timeout occurs the connection is ma is logged off.
$encryptpassword	Set to kTrue to use password encryption when logging on. The default is kFalse.
$cterrorlayer	Layer at which the current session client error occurred. Read only.
$cterrororigin	Origin of current session client error. Read only.
$cterrorseverity	Severity of current session client error. Read only.
$cterrornumber	Error number of current session client error. Read only.

Property	Description
$moneydps	This property determines the number of decimal places used to store and display data retrieved fro used when creating schemas- provided that this property is set before dragging the table into the li 4 for backward compatibility but can be set to 0, 1, 2, 3, 4, 5, 6, 8, 10, 12 or 14.
$locale	The locale name that will be used by the connection. This is initially set to the default locale contain file. $locale may be set to a different value provided that the DAM is not logged on. Valid locale strin language-character set pairs contained in the locales.dat file for which the corresponding language Assignment fails if the locale information specified cannot be found or is not installed.
$nativewarntext	Information or warning text generated by the last operation (read-only)
$nationaltounichar	Studio 5.1. When this property is set to kTrue, Omnis National fields will be mapped to Sybase NVAR be mapped to VARCHAR. Also affects the text returned by $createnames(). When kFalse (the defaul mapped to VARCHAR columns (supporting the UTF-8 encoding).
$sdbsocket	Studio 8.0.2. Server-side TCP socket being used by a data bridge connection (read-only)

Session Methods

Method	Description
$setremotepassword()	SessObj.$setremotepassword(cServerName, cPassword). Set a password for a remote serve This will fail if the session is logged on. If cServerName is NULL, the password is used as a u password for all servers with no specified password.
$clearremotepasswords()	SessObj.$clearremotepasswords(). Clear all passwords for remote server connections.
$getnames()	SessObj.$getnames(&list). Retrieves a list of directory service names contained in the local sql.ini/interfaces/freetds.conf file together with their connection attributes supplied as sub- Returns kTrue on success or kFalse if the file cannot be opened or parsed. Uses the SYBASE FREETDSCONF environment variables to locate the configuration file. Requires Studio 10.2 31237 or later.

Statement Properties

Property	Description
$cterrorlayer	Layer at which the current session client error occurred. Read only.
$cterrororigin	Origin of current session client error. Read only.
$cterrorseverity	Severity of current session client error. Read only.
$cterrornumber	Error number of current session client error. Read only.
$rpcparamspending	If kTrue this denotes that an rpc parameter result set is pending.
$bindshort0dpassmallint	If kTrue, Short number 0dp parameters are bound to the server as SMALLINTs. Otherwise t mapping is used- NUMERIC(9,0).
$emptystringisblank	When set to kTrue, the DAM inserts empty character strings into VARCHAR columns as sin characters. When set to kFalse, a NULL or chr(0) is inserted. $emptystringisblank defaults t $emptystringisblank does not affect Omnis character strings >255 characters which map to columns. Empty TEXT values are always inserted as single space characters.

Statement Methods

Method	Description
$cancelresultset()	StatementObj.$cancelresultset(). Cancel the current result set. This will allow any further result sets statement is using a cursor, the cursor is closed and its results are discarded.
$writelob()	StatementObj.$writelob(vVariable, cTablename, cColumnname [,cWHERE-clause, bUselog = kTrue]). text column with the value of vVariable. The cTablename, cColumnname and optional WHERE claus to be updated. If bUselog is kTrue, changes may be rolled back

Connecting to your Database

To connect to your database, you need to create a session object with a subtype of SYBASESESS. In order to log on to the database using the SessObj.$logon() method, the hostname must contain a valid Sybase host alias previously generated by the Sybase client tools (see your Sybase documentation for more details).

Multiple Select Tables

```
create proc multi_select as
SELECT firstName, lastName FROM Agents
SELECT id, name FROM Customers
```

To execute this from an Omnis method and to fetch the results:

```
Do tStatement.$execdirect('exec multi_select') Returns #F
Do My_List1.$define()
Do tStatement.$fetch(My_List1,kFetchAll) Returns #1
If #1=kFetchFinished & tStatement.$resultspending=kTrue
  Do My_List1.$define()
  Do tStatement.$fetch(My_List1,kFetchAll) Returns #1

End If
```

The Sybase DAM reports the select tables exactly as Open Client reports them. If a select result set has no rows, $fetch() will return kFetchFinished the first time it is invoked for that set.

Program Name

The Sybase sysprocesses table (in the master database) has a program_name column that stores a separate name for each connection to the server. The session $programname property lets you put a name into this column for the current session so that you can use it to distinguish multiple sessions.

The default name is the name of the current library, i.e. $clib().$name If you wish to change this you must set this property before logging on to the server, because the value gets set at logon. If the value is set after logon it does not take effect until the session is logged on again. The value persists across logons and logoffs, and $clear() does not reset it.

If the string assigned to the property is too long, the DAM truncates it without reporting an error. The DAM can store a maximum of 255 characters but the program_name field in the sysprocesses table currently only allows 16.

Error Handling

If an error is raised on either a Sybase session or statement object, the $errorcode and $errortext properties associated with the object will provide the generic error code and error text. If there is an associated native Sybase server or client error this will be returned in the $nativeerrorcode and $nativeerrortext properties associated with the object.

It is possible for an Omnis command to generate multiple Sybase server and client error messages. If this is the case, the object's $nativeerrorpending property will be set to kTrue. To retrieve the next set of error information the application can use the $nextnativeerror() method. The DAM will return server errors and then client errors in the order in which they were generated. Any informational messages returned from Sybase are ignored.

If a new Sybase DAM command is issued or a Sybase property is set, the current error information for that object is cleared. The error set for the session is shared between the session and all statements in that session. If a session or statement clears the current error set, any other statement with multiple errors pending will only be able to retrieve the last cached error since the error set will have been cleared.

You should be aware that the Sybase server and client errors reported may have codes and messages that sometimes differ between the macOS and Windows Sybase clients.

The Sybase DAM defines several of its own internal error codes. These are returned in the $nativeerrorcode and $nativeerrortext properties of the session and statement.

Error Code	Description
20000	The Omnis bind variable could not be mapped to an equivalent Sybase type.
20005	The session must not be logged on.
20010	The login timeout value must be >= 0.
20011	The query timeout value must be >= 0.
20030	Sybase TEXT and IMAGE columns cannot be bound as part of an RPC call.
20050	The Omnis field cannot be null or empty.
20051	The table name must be a character string which is not null or empty
20052	The column name must be a character string which is not null or empty

Error Code	Description
20053	The WHERE-clause must be a character string
20054	The column to be updated was not a TEXT or IMAGE column
20055	A memory allocation error occurred during the $writelob command

If a Sybase session or statement object generates an Open Client error, the error code is decoded into the $cterrorlayer, $cterrorseverity, $cterrororigin and $cterrornumber. Sybase Open Client errors use these 4 error components to provide more detail about the error raised.

If a timeout error occurs and the session's $failtimeout property is kTrue, a timeout error will be raised, the connection will be marked as dead and the session will be logged off. If the $failtimeout is set to kFalse (the default) the connection or query will be re-tried.

Large Objects (LOBs)

The Sybase Object DAM can send and retrieve text and image fields which are referred to as LOBs or "large objects". You can insert, update and fetch the fields using the standard SQL object methods or you can use the $writelob() method to update a text or image field on the DBMS, with the implicit functionality to retrieve a large text or image field.

Transferring BLOBs is very memory-intensive, since each layer of software has a copy of at least part of the blob. Thus, sending a simple 40K picture can demand several times that amount of RAM before it gets passed over to the DBMS. Therefore an application must have sufficient memory resources to transfer large text and image data. The built in chunking mechanism can be used to reduce the amount of memory Omnis requires to transfer a LOB value. For example, the default settings for the session's $lobthreshold and $chunksize properties ensure that data greater than 32K is sent in chunks no greater than 32K. The chunksize and threshold can be altered to suit the resources available. If a system has more memory, the threshold and chunksize can be increased to send fewer, larger chunks.

There are no limitations, aside from memory concerns, on sending or retrieving multiple LOBs in one SQL statement.

You should not forget to set the textsize parameter on the server. This parameter tells the server to truncate all outgoing values to this setting (see your Sybase documentation for more details). Therefore, if you set the textsize parameter to the default of 32,767 and select a 500K image, you get a 32,767 byte value in Omnis.

```
Do tStatement.$execdirect('set textsize 123456')
# this sets the textsize parameter, for this session, to about 123K
```

This setting is for fetching values only. When fetching LOBs under Mac Classic using the standard commands, you should not set this parameter to its largest value. Increasing this also causes Open Client to allocate more memory to deal with the larger LOBs. Therefore, setting it too small will truncate your fetched data while setting it too large may cause Open Client to kill your connection. If you are retrieving a variety of LOBs, you should try to set it as closely as you can to the size of the largest LOB; you can set this for each SQL statement sent. You can also use the Sybase datalength() function to find out how long the value is that you want to retrieve, and use this to set the textsize parameter.

The Sybase DAM provides a faster and more memory efficient way to update a text or image column through the use of the statement method $writelob(). To use the $writelob() method you must already have the row in the database and the column value that is being updated must have non-NULL data in it.. You would usually create the row with a blank (' ') in the column, for instance, use the $writelob() method to update the value with the LOB data.

```
Do tStatement.$execdirect ("insert into mytable (x, mycol) values (2, ' ')")
Do tStatement.$writelob(LOB_DATA,'mytable','mycol','where x=2',kTrue)
```

This command places the value of the Omnis field LOB_DATA into the column mycol of the table mytable in the row where x has the value of 2. Thus, the method places a single LOB value into a location that you specify.

The method is defined as:

```
StatementObj.$writelob(vVariable, cTablename, cColumnname [,cWHERE-clause, bUselog = kTrue])
```

The **vVariable** parameter is the Omnis variable containing the data to be sent to the server, in the example this is LOB_DATA. This variable cannot be NULL or empty.

The **cTablename** and **cColumnname** parameters identify the table and column to update. These cannot be NULL or empty.

The **cWHERE-clause** parameter supplies an optional WHERE clause for a SQL SELECT statement, including the word 'WHERE'. If your WHERE clause is ambiguous Omnis updates the first LOB value it finds, so the value updated may not be the one you intended. Make sure your clause specifies a unique row.

The **bUseLog** parameter denotes whether to log this action in the transaction log. If you do not log the action, you cannot roll it back. The default is kTrue to log the update. Setting this parameter to kFalse requires that the select into/bulkcopy option be set to true with the system procedure sp_dboption for the database on the DBMS. If you do not wish to log updates, you must consult your documentation and system administrator as this may have significant ramifications on being able to backup and recover your database.

The $writelob() method sets the flag false and sets error information in the same way as a standard statement method.

Sybase recommend that data should be updated using the $writelob() method if the data size exceeds 100K.

Remote Procedure Calls

The Sybase DAM supports the use of the session method $rpcdefine() to define a remote procedure and the statement method $rpc() to call a procedure. The DAM does not support the statement methods $rpcprocedures() and $rpcparameters().

An application must generate the parameter list passed to $rpcdefine() to describe the parameters in the Sybase procedure. When a $rpc() call invokes the procedure the parameters passed are mapped from their Omnis type definition to the equivalent Sybase type in the same way as standard Omnis bind variables. A $rpc() call will fail if a parameter definition includes an Omnis character or binary field larger than 255 since the parameter will map to a text or image field which are not valid for use as parameters in Sybase stored procedures.

If an rpc definition defines parameters of type kParameterInputOutput, these are treated as output parameters since Sybase does not support updateable input parameters. The DAM cannot update output parameters directly. If the procedure uses output parameters these must still be specified in the call to $rpc() and are returned as a parameter result set which is available when the statement property $rpcparamspending is kTrue. They must be processed by the application in the same way as a normal result set. Any result sets generated by the stored procedure must be processed before the parameter results become available. The return status should not be included in the call to $rpc() as this will be set in the statement $rpcreturnvalue property which is set after the stored procedure results are processed.

The following creates a Sybase stored procedure which takes 2 input parameters and 1 output parameter. This procedure returns two result sets.

```
Do tStatement.$execdirect(\
    'create procedure Test_SYBRPC @parm1 varchar(30), \
    @parm2 varchar(30), @parm3 varchar(60) \
    OUTPUT AS SELECT @parm3 = @parm1+@parm2 \
    SELECT * from sys execute sp_who RETURN 12345')
```

A list is used to define this procedure in the Sybase session.

```
Do #L1.$define(#1,#2,#3,#4) Returns #F
Do #L1.$add(kInteger,kLongint,0,kParameterReturnValue)
Do #L1.$add(kCharacter,kSimplechar,30,kParameterInput)
Do #L1.$add(kCharacter,kSimplechar,30,kParameterInput)
Do #L1.$add(kCharacter,kSimplechar,30,kParameterOutput)
Do tSession.$rpcdefine('Test_SYBRPC',#L1) Returns #F
```

The procedure can then be called.

```
Calculate Parm1 as 'Hello ' ## Character 30
Calculate Parm2 as ' There' ## Character 30
Calculate Parm3 as ## Character 60
Do tStatement.$rpc('Test_SYBRPC',Parm1,Parm2,Parm3) Returns #F
```

Since the stored procedure generates two result sets these must be processed first.

This will fetch the results from the sysusers and sp_who queries.

```
If tStatement.$resultspending=kTrue
  Do #L1.$define()
  Do tStatement.$fetch(#L1,kFetchAll) Returns #1
```

```
End If
If tStatement.$resultspending=kTrue
  Do #L1.$define()
  Do tStatement.$fetch(#L1,kFetchAll) Returns #1
End If
```

The stored procedure return status is placed in the statement’s $rpcreturnvalue property and the parameter result set is then available.

```
Calculate #1 as tStatement.$rpcreturnvalue ## will set #1 to 12345
If tStatement.$rpcparamspending=kTrue
  Do #L1.$define()
  Do tStatement.$fetch(#L1) Returns #1
End If

Calculate Parm3 as #L1.1.1 ## will set Parm3 to 'Hello There'
```

Multiple Cursors

If a statement is issued without using a cursor, i.e. $usecursor is set to kFalse, any results generated will block the connection and no other operation on any other statement can be performed until the blocking result set is completely fetched or cancelled. To avoid blocking the connection with pending results, use a statement which has the $usecursor set to kTrue. Note that a statement using a Sybase cursor must have a unique statement name and only allows SQL SELECT and EXECUTE procedure commands to be issued.

Meta-Data Queries

As of Studio 5.1.1, the $indexes() meta data method returns additional information via the DamInfoRow column. The DamInfoRow will be defined with the following columns (as returned by the *sp_statistics* stored procedure):

Column	Description
TableName	Character column containing the table name passed previously.
IndexQualifier	Character column indicating the index owner. For Sybase, this is usually the same as *TableName*.
IndexType	Character column indicating the index type, e.g. “Clustered” or “Non-Clustered”.
Collation	Character column indicating the collation type, either “Ascending” or “Descending”.
Cardinality	Integer column containing the number of indexed or unique rows.
Pages	Integer column containing the number of pages used to store the index.

Logon Problems using the SYBASEDAM

In the event of connection problems, there are a number of Technotes available on the Omnis website (https://www.omnis.net/developers/resources/technotes/) which discuss Sybase connection issues in greater detail.

Possible causes:

- The $SYBASE/interfaces file is missing or the contents are invalid- follow the installation tasks outlined above.
- The logon hostname does not match the name contained in the $SYBASE/interfaces file- check the contents of the interfaces file, paying attention to upper and lower case characters.
- The supplied username and/or password were incorrect- check at the server.
- The xcomp:ini:sybasedam.ini file was not found or one or more of the environment variables are set to incorrect values. Review this file.
- The DAM does not load. The dynamic linker may be unable to locate the required Sybase client libraries. Check environment variables (DYLD_LIBRARY_PATH for macOS, LD_LIBRARY_PATH for Linux)

Sybase Troubleshooting

- Sybase is a case-sensitive RDBMS. Check the case of the table or column names if you can see a table but cannot select anything out of it
- Sybase defaults to NOT NULL columns; you must initialize columns to a specific value while inserting data, or insertion will fail
- Any number with no digits after the decimal point, that is > +/- 231 will generate an error and not be inserted. This is because Sybase tries to parse numbers without decimal points as integers
- Sybase does not support binding a NULL Boolean field in Omnis to a Sybase bit field
- Sybase does some character mapping where required, but you may need to do character conversion explicitly using the Omnis character mapping tables.
- Sybase interprets empty strings as single spaces.
- Fetching pictures from Sybase stored there by other applications, even in standard formats, is likely to cause problems, since Omnis stores all pictures in a special format. This occurs even in platform-specific graphics formats such as PICT or BMP.
- The $tables() session method can only report information about tables in the current database and does not return system tables.
- The $columns() session method can only report information about tables owned by the current user in the current database.
- The $indexes() session method can only report information about indexes on tables in the current database.
- Sybase does not allow DDL statements to be issued within a user defined transaction, i.e. do not use statements such as CREATE, DROP and ALTER when the session's transaction mode is kSessionTranManual. Do not use the $indexes() method using kSessionTranManual since this method creates a table.
- Sybase automatically strips spaces from character data returned to Omnis.
- "Data buffers could not be allocated" error following a logon attempt:
  This error normally occurs if the Sybase environment variables; SYBASE, SYBASE_OCS and/or LANG/LC_ALL are not correct. Check the sybinit.err file (in the Omnis folder) for more details about the error.

Further troubleshooting notes, "how-tos" and tips can be found on the Omnis website at: https://www.omnis.net/developers/resources/technotes/

Sybase Data Type Mapping

Omnis to Sybase

Omnis Data Type	Sybase Data Ty
CHARACTER	
Character/National 0	varchar(1)
Character/National 1 <= n <= 255	varchar(n)
Character/National > 255	text
DATE/TIME	
Short date (all subtypes)	datetime
Short time	datetime
Date time (#FDT)	datetime
NUMBER	
Short integer (0 to 255)	tinyint
Integer 32 bit	int
Integer 64 bit	bigint
Short number 0dp	numeric(9,0)
Short number 2dp	numeric(9,2)
Number floating dp	double precisio
Number 0..14dp	numeric(15,0..14
OTHER	
Boolean	bit
Sequence	int
Binary/Picture/List/Row/Object/Item reference where $blobsize <= 255	varbinary($blob

Omnis Data Type	Sybase Data Ty
Binary/Picture/List/Row/Object/Item reference where $blobsize > 255	image

Sybase to Omnis

Sybase Data Type	Omnis Data Type
CHARACTER	
char(n)	Character n
varchar(n)	Character n
nchar(n)	Character n
nvarchar(n)	Character n
text	Character 10,000,000
DATE/TIME	
datetime	Date time (#FDT)
smalldatetime	Date time (#FDT)
NUMBER	
tinyint	Short integer (0 to 255)
smallint	Short number 0dp
int	Integer 32 bit
bigint	Integer 64 bit
numeric(p,n)	Number (n)dp
decimal(p,n)	Number (n)dp
real	Number floating dp
float	Number floating dp
double precision	Number floating dp
money	Number 4dp
smallmoney	Number 4dp
OTHER	
bit	Boolean
binary(n)	Binary
varbinary(n)	Binary
image	Binary

DB2

This section contains the additional information you need to access a DB2 Universal Server database, including server-specific programming, data type mapping to and from DB2, as well as troubleshooting. For general information about logging on and managing your database using the Omnis SQL Browser, refer to the earlier parts of this manual.

Properties and Methods

Session Properties

Property	Description
$datetimeformat	This stores an Omnis date format string used to map a Date time (#FDT) bind variable to the correct DB2 supports different regional timestamp formats. The date is stored on the server in an internal b 'y-M-D H:N:S' This method is equivalent to the old-style keyword.
$drivername	The name of the session driver.
$driverodbcversion	The version number of the session driver.

Session Methods

Method	Description
$getdatasources()	SessionObj.$getdatasources(lListOrRow) populates the list with the name and description of the data machine. The list is redefined as having two columns- *DataSourceName* and *Description*. DataSourc 32. Description is defined as Character 255. This method is equivalent to the old-style **<**GET_DATASOU

Statement Properties

Property	Description
$erroronnodata	If set to kTrue (default), $execute() and $execdirect() will fail if execution returns SQL_NO_DATA, i.e. if a row statement could not be found. If set to kFalse, SQL_NO_DATA errors are ignored to be consistent with othe does not affect SELECT statements.

Connecting to your Database

To connect to your database, you need to create a session object with a subtype of DB2SESS. In order to log on to the database using the SessObj.$logon() method, the hostname must contain the catalog database name entered using the DB2 Command Line Processor or using the Client Configuration Assistant if installed. The user name and password should contain the values required by the database. For example:

```
Do SessObj.$logon('MyDatabase','UserID','Password','MySession') Returns #F
```

In the event of connection failure, the DAM will timeout as dictated by any timeout policy in use by the server. Logon failures are usually reported immediately.

Transactions

Generally, using manual transaction mode results in increased performance because the session object does not force a commit after each statement.

If you do not have a results set pending, the DB2 session object will commit each statement if the transaction mode is automatic. If the transaction mode is server, the server commits the statement automatically.

Dates

The session property $defaultdate allows default values to be added to date values mapped to the server where the Omnis date value does not contain complete information, e.g. when a Short time is mapped to a server DATETIME. The date stored in this property is in a generic format, i.e. it is compatible with any regional date format that the server may be using.

Boolean Type

DB2 does not include a specific type for storing single bit data. The Omnis Boolean type is therefore converted to a CHAR(3) value and stored as 'YES' or 'NO' in the server table. The string representation can then be mapped back to an Omnis Boolean type when the data is retrieved.

BLOB Type

The session property $blobsize can be used to specify the size argument for columns of type BLOB generated when the $createnames and $coltext methods are used.

Values range from 1 to 10000000. The default value for $blobsize is 10000000 which is also the maximum size of an Omnis binary variable.

This property is equivalent to the old-style <SETBLOBSIZE> keyword.

Meta-Data Queries

The meta-data statement methods $columns(), $indexes() and $tables() allow you to receive information about the objects in your database. The $tables() method takes an optional owner name as a parameter. The $indexes() and $columns() methods optionally take the database and/or owner name with the table name parameter. See SQL Programming for more information on these methods.

When a database, owner or table name is specified, the result set is constrained by those schemas which meet the filter criterion, i.e. to return column information about the "addressbk" table owned by "robert" in database "acc_db" the following can be issued.

Do tStatement.$columns('acc_db.robert.addressbk') Returns #F

DB2 Troubleshooting

Reserved Words

This section covers the DB2 specific reserved words.

The following schema names are reserved: SYSCAT, SYSFUN, SYSIBM & SYSSTAT.

In addition, it is strongly recommended that schema names never begin with the SYS prefix, as SYS is by convention used to indicate an area reserved by the system.

There are no words that are specifically reserved words in DB2. Keywords can be used as ordinary identifiers, except in a context where they could also be interpreted as SQL keywords. In such cases, the word must be specified as a delimited identifier. For example, COUNT cannot be used as a column name in a SELECT statement unless it is delimited.

IBM SQL and ISO/ANSI SQL92 include reserved words, these reserved words are not enforced by DB2 Universal Database, however it is recommended that they not be used as ordinary identifiers, since this reduces portability. Please see the final chapter in this manual which lists the SQL reserved words.

Further troubleshooting notes, "how-tos" and tips can be found on the Omnis website at: https://www.omnis.net/developers/resources/technotes/

DB2 Data Type Mapping

The following table describes the data type mapping for Omnis to DB2 connections. This mapping is predefined and is based on the best fit for each of the Omnis data types.

Omnis to DB2 UDB

Omnis data type	Server data type
CHARACTER	
Character/National(n) <= 4000	VARCHAR(n)
4000 < Character/National(n) <= 32,700	LONG VARCHAR(n)
Character/National(n) > 32,700	CLOB(n)
DATE/TIME	
Short date (all subtypes)	DATE
Short time	TIME
Date time (#FDT)	DATETIME
NUMBER	
Short integer	SMALLINT
Integer 32 bit	INTEGER
Integer 64 bit	BIGINT
Short number 0 dp	DOUBLE
Short number 2 dp	DOUBLE
Number 0..14 dp	DOUBLE
OTHER	
Boolean	CHAR (3)
Sequence	INTEGER
Picture	BLOB($blobsize)
Binary	BLOB($blobsize)
List	BLOB($blobsize)
Row	BLOB($blobsize)
Object	BLOB($blobsize)
Item reference	N/A

DB2 UDB to Omnis

Server Data Type	Omnis Data Type
NUMBER	
SMALLINT	Integer 32 bit
INTEGER	Integer 32 bit
BIGINT	Integer 64 bit
DECIMAL(p,s)	Number (s)dp
NUMERIC(p,s)	Number (s)dp

Server Data Type	Omnis Data Type
FLOAT	Number floating dp
REAL	Number floating dp
DOUBLE	Number floating dp
CHARACTER	
CHAR(n)	Character (n)
VARCHAR(n)	Character (n)
LONG VARCHAR(n)	Character (n)
CLOB(n)	Character (n)
DATE/TIME	
DATE	Short date
TIME	Short time
TIMESTAMP	Date time (#FDT)
BINARY	
BINARY	Binary
VARBINARY	Binary
LONGVARBINARY	Binary
BLOB	Binary
EXTENDERS	
IMAGE	Binary
AUDIO	Binary
VIDEO	Binary
TEXT	Binary

MySQL

This section contains the additional information you need to access a MySQL database, including server-specific programming, data type mapping to and from MySQL, as well as troubleshooting. For general information about logging on and managing your database using the Omnis SQL Browser, refer to the earlier parts of this manual.

Properties and Methods

Session Properties

Property	Description
$clientflags	SessObj.$clientflags sets the optional client flags logon parameter before executing $logon(), (The added together if required). Their use is beyond the scope of this text and the default value of zer purposes. Client flags are discussed further in the MySQL C API reference under mysql_real_conn
$database	SessObj.$database sets the additional *database* logon parameter before executing $logon(). Onc new value to this property causes the current database to change. When the session is created, a to this property.
$defaultdateisempty	If kTrue, fetched datetimes matching $defaultdate are treated as empty values. (Studio 8.0.2)
$hostinfo	SessObj.$hostinfo describes the type of connection in use, including the server host name. (Read
$logontimeout	The number of seconds before a connection attempt times out. SessObj.$connectoption() can als required.
$port	SessObj.$port sets the additional *port* logon parameter before executing $logon(). This will be the connection. The default port number is 3306.
$protoversion	SessObj.$protoversion is the version of the protocol in use by the current connection. (Read-only)
$socket	SessObj.$socket is the socket or named pipe that should be used for the connection, applicable t
$sslcipher	SessObj.$sslcipher returns the name of the SSL cipher being used for the current SSL connection
$threadid	SessObj.$threadid is the thread ID of the current connection. (Read-only)
$threadsafe	SessObj.$threadsafe is kTrue if the client library was compiled as thread-safe. (Read-only)

Session Methods

Method	Description
$changeuser()	SessObj.$changeuser({cUsername,cPassword}) changes the current user. The new user will be identified by the $database property.
$characterset()	SessObj.$characterset() returns the name of the default character set for the current connectio

Method	Description
$connectoption()	SessObj.$connectoption({iOption,vArgument}) specifies extra connect options and affects the l method may be called multiple times to set several options. Available option constants can be under MYSQLDAM-ConnectOptions. For further details, refer to the MySQL C API documentati Studio 8.0.2, $connectoption() can also be called with named connection attributes, e.g. Do sessObj.$connectoption('program_name','My Program')
$getdatatypemapping()	SessObj.$getdatatypemapping({lMappings}) retrieves a list of Omnis-to-MySQL data type map session. This list is formatted as described in the section below and is suitable for re-assignmen $setdatatypemapping() method if required. On successful retrieval of the list, $getdata typema otherwise kFalse is returned. $getdatatypemapping() can be called either before or after the s
$insertid()	SessObj.$insertid() returns the ID of an AUTO_INCREMENT column generated by the most rece function after you have performed an INSERT query into a table that contains an AUTO_INCRE zero if the previous query did not generate an AUTO_INCREMENT value, or was not an INSERT/
$ping()	SessObj.$ping() checks whether the connection to the server is working. If it has gone down, a attempted. $ping() returns kTrue if the connection is alive.
$query()	SessObj.$query(cSqlText) allows SQL statements not supported by the MySQL prepared staten directly on the connection. At the time of writing, such statements include the following admi DROP DATABASE/USER, HANDLER, TRUNCATE, ALTER DATABASE/INDEX/USER, CREATE DATA TABLES, UNLOCK TABLES, SAVEPOINT, ROLLBACK TO SAVEPOINT, FLUSH, CACHE INDEX, LOA CONNECTION/QUERY, RESET. cSqlText can be either a single SQL statement or multiple stater bracket notation if required. Bind variables are not supported; this functionality is provided by returns kTrue on success, otherwise kFalse. Error messages are returned via the session object
$queryinfo()	SessObj.$queryinfo() retrieves a string providing information about the most recently executed derived from the session, but only for certain INSERT, UPDATE and ALTER statements. For furth MySQL C API documentation for mysql_info().
$queryresult()	SessObj.$queryresult(lResult) allows the result set generated by a previous call to $query() to b variable which is cleared and redefined from the columns of the result set. The entire result set returned via a single call to $queryresult(). $queryresult() performs limited data type conversion recognising binary, integer and decimal numbers, defaulting to Character for all other types. T there is no result set pending on the session object. As of Studio 10.1 $queryresult() can be calle result sets. $queryresult() returns kFalse when there are no more result sets to return.
$serverdebuginfo()	SessObj.$serverdebuginfo() instructs the server to write some debug information to its error lo connected user must have the SUPER privilege. The log file name can be specified when the s –log-error[=filename] on the command line.
$servershutdown()	SessObj.$servershutdown() asks the database server to shut down. The connected user must h Note: no further confirmation is sought before severing the connection and shutting the serve server was successfully shutdown.
$serverstatus()	SessObj.$serverstatus({cInfo}) returns information about the server's current status, including t number of running threads, questions, reloads, and open tables.
$setdatatypemapping()	SessObj.$setdatatypemapping({lMappings}) sets the Omnis-to-MySQL data type mappings for contains a prioritised list of mappings for Omnis data types and subtypes to their intended My section below on the format of the mapping list. $setdata typemapping() can be used to map custom MySQL data types, e.g. SET and ENUM types. This is also explained in the section below $setdatatypemapping() kTrue is returned, otherwise kFalse is returned. SessObj.$setdata typer before or after the session has logged on.
$sslset	SessObj.$sslset([cKey, cCert, cCA, cCAPath, cCipher]) is used for establishing a secure connectio before $logon(). cKey is the path name to the key file. cCert is the path name to the certificate f certificate authority file. cCAPath is the path name to a directory that contains trusted SSL CA c cCipher is a list of permissible ciphers to use for SSL encryption. Any unused SSL parameters w

Statement Methods

Method	Description
$columns()	StatObj.$columns({cTableName}). Returns information describing the columns of the supplied table n with an optional database name; [database.]tablename if required. The DamInfoRow column returned information for each column described. The row is defined with the following columns: UniqueKey: kT MultipleKey kTrue if col is part of a compound index Unsigned: kTrue if col has the UNSIGNED attribut ZEROFILL attribute Binary: kTrue if col contains binary (BLOB/TEXT) data AutoIncrement: kTrue if col h Number: kTrue if col contains numeric data DefaultValue: Returns the default value for col as char dat

Method	Description
$rpcprocedures()	StatObj.$rpcprocedures([cOwner]) generates a result set containing the names of stored procedures a created by the named user. The DamInfoRow column returned by $rpcprocedures() contains addition described. The row is defined with the following columns: Type: Specifies whether the row describes a Name The specific name of the procedure. Language: The programming language contained within t Describes data usage characteristics of the procedure. Deterministic: kTrue if the procedure is 'determ same result for the same input parameters. Security Type: Describes the permissions used when execu Contains a comma separated list of input/output parameters. Returns: Describes the data type returne text content of the procedure. Created: The date and time when the procedure was created. Modified: procedure was last modified. SQL Mode: Describes the SQL syntax supported by the procedure. Comn the procedure was created. Stored procedures and functions are not supported in versions of MySQL p

Logging on to MySQL

The MySQL DAM interfaces directly with the MySQL client library, therefore the way the DAM logs on to the server is slightly different to the other Object DAMs.

Specifically, the $logon() *hostname* parameter is taken as the server hostname or IP address. The username and password parameters are supplied as normal.

As well as the parameters supplied to $logon(), there are some additional parameters which you can set using the following session properties:

- $port - The port number of the MySQL server
- $database - The database name
- $clientflags - Sets additional behavior for the logon
- $socket - Specified if you do not want to use a TCP/IP connection

When logging on using the SQL Browser, default values are used when the Port and Database fields are left blank.

Transactions

If you require transaction support with MySQL, your server needs to support BDB or InnoDB table types. Manual transactions may only be made on tables of these types.

When creating tables, you need to specify the table type required if you do not want the default type (MyISAM).

The following properties and methods apply to transactions. Use of these properties or methods is equivalent to executing the SQL statements shown:

Method/Property	Description
$begin()	SessObj.$begin() = Begin
$commit()	SessObj.$commit() = Commit
$rollback()	SessObj.$rollback() = Rollback
$transactionmode	SessObj.$transactionmode.$assign(SessionMode) kSessionTranAutomatic: Autocommit = 1 (the default) kSessionTranServer: Autocommit = 1 kSessionTranManual: Autocommit = 0

If you are not using InnoDB or BDB table types, you can achieve table locking using the MySQL lock tables or unlock tables SQL commands. Refer to the MySQL language reference for further details.

Using LOAD DATA

The MySQL DAM supports use of the LOAD DATA INFILE SQL syntax via the sessionobject.$query() method. You need to set the kMySqlOptLocalInfile connect option to enable *local* files to be loaded. The default command syntax will load the contents of a tab-delimited text file into the specified table, for example:

```
Do cSess.$connectoption(kMySqlOptLocalInifile,1) Returns #F
Do cSess.$logon('192.168.00.100','myUser','myPass','session1') Returns #F
Do cSess.$newstatement() Returns cStat
Do cStat.$execdirect('create table loadtest(col1 int, col2 varchar(32), col3 date, col4 numeric(9,2))') Ret

# Now load the data into the table..

Do cSess.$query("load data local infile 'C:/Users/myUser/Desktop/data.txt' into table loadtest") Returns #F
```

Example text file contents (tab-separated values):

```
1   One    2020-08-26   1.23
2   Two    2020-07-31   2.45
3   Three  2019-06-06   33.75
4   Four   2018-01-30   127.0
5   Five   1999-10-02   32.333
```

Please refer to the MySQL documentation for further details on the LOAD DATA statement.

MySQL Data Type Mapping

Omnis to MySQL

The default data type mappings from Omnis to MySQL are shown below.

Omnis Date Type	MySQL Data Type
CHARACTER	
Character n (n<=255)	VARCHAR(n)
National n (n<=255)	NATIONAL VARCHAR(n)
Character/National n (n<=65534)	TEXT
Character/National n (65534<n<=10000000)	MEDIUMTEXT
NUMBER	
Integer 64 bit	BIGINT
Integer 32 bit	INT
Short integer	TINYINT UNSIGNED
Number 0..14dp	DECIMAL(15,0..14)
Number floating dp	DOUBLE
Short number 0/2dp	DECIMAL(9,0/2)
DATE/TIME	
Short date (all subtypes)	DATE
Short time	TIME
Datetime (#FDT)	DATETIME
OTHER	
Boolean	BOOL
Picture	MEDIUMBLOB
List	MEDIUMBLOB
Row	MEDIUMBLOB
Object	MEDIUMBLOB
Binary	MEDIUMBLOB
Item reference	TINYBLOB
Sequence	INT UNSIGNED AUTO_INCREMENT PRIMARY KEY

Note that this is equivalent to the list returned by a call to $getdatatypemapping() on a newly created session object:

OmnisType	OmnisSubtype	Parameter	MySqlType
char	simple	255	VARCHAR($)
char	national	255	NATIONAL VARCHAR($)
char	national	65534	TEXT
char	national	10000000	MEDIUMTEXT
integer	64 bit	0	BIGINT
integer	32 bit	0	INT

OmnisType	OmnisSubtype	Parameter	MySqlType
integer	shortint	0	TINYINT UNSIGNED
number	14dp	0	DECIMAL(15,$)
number	float	0	DOUBLE
number	2dpShortnum	0	DECIMAL(9,$)
boolean		0	BOOL
date	date2000	0	DATE
date	time	0	TIME
date	datetime	0	DATETIME
picture		0	MEDIUMBLOB
list		0	MEDIUMBLOB
row		0	MEDIUMBLOB
object		0	MEDIUMBLOB
binary		0	MEDIUMBLOB
itemref		0	TINYBLOB

Assigning a new mapping table using $setdatatypemapping()

If you need to make a change to the default Omnis to MySQL data type mappings, you should probably base your new mappings on the default mappings obtainable by calling $getdatatypemapping() and add/remove lines to the returned list as required before calling $setdatatypemapping() to install the new mapping table.

There are a number of points to note regarding the format and processing of the list used by these methods and these are discussed below.

Omnis subtype precedence

Note that where multiple occurrences of Omnis types appear in the list, the Omnis subtype(s) should be specified in ascending *numerical* order, especially if you intend a mapping to apply to all Omnis subtypes <= the supplied value. This is because when searching for a match, the list is processed in order from the first entry to last- the search ends at the first matching entry. The full list of acceptable data subtypes can be found in the Omnis catalog (F9) and their text equivalents are shown in the OmnisSubtype column, as summarised below:

Omnis constant	Numeric value (precedence)	Character equivalent (OmnisSubtype)
Character subtypes		
kSimplechar	0	simple
kNatchar	1	national
Integer subtypes		
k32bitint	0	32 bit integer
kShortint	32	shortint
k64bitint	64	64 bit integer
Number subtypes		
k0dp	0	0dp
k1dp	1	1dp
k2dp	2	2dp
k3dp	3	3dp
k4dp	4	4dp
k5dp	5	5dp
k6dp	6	6dp
k8dp	8	8dp
k10dp	10	10dp
k12dp	12	12dp
k14dp	14	14dp
kFloatdp	24	float
k0dpShortnum	32	0dpShortnum
k2dpShortnum	34	2dpShortnumv
Datetime subtypes		
kDate1900	0	date1900
kDate1980	1	date1980
kDate2000	2	date2000
kTime	6	time
kDatetime	1000	Datetime

Note that where an Omnis type does not have a subtype (e.g. Binary), it is acceptable to leave the subtype column blank.

Parameter column

The ‘Parameter’ value in the data type mapping list specifies the maximum length or size of data to which the mapping will apply, e.g. a value of 255 specified for a character data type signifies that that mapping will apply to Omnis character data with a length of <= 255 characters. You should therefore ensure that where there are multiple occurrences of the same Omnis data type and subtype, these are entered in ascending order of parameter value. The maximum size of an Omnis character/binary field is 10,000,000 bytes.

MySqlType column

The MySqlType value specifies the string which will be returned by session.$createnames() when that row matches the supplied Omnis data type & subtype. This can be (but is not restricted to) any string which constitutes a valid MySQL column type, e.g.

```
SET(‘One’,‘Two’,‘Three’,‘Four’)
```

Where a ‘$’ character is used as part of the MySQL type, the appropriate length or scale attribute will be substituted when $createnames() is called.

For valid MySQL data type assignments, the DAM also uses the data type mapping table to map outgoing bind variables to their corresponding MySQL column types.

Example applications of $setdatatypemapping()

The intended use of $setdatatypemapping() is to allow schema columns to conditionally map to custom MySQL data types, not implemented by default, e.g. the SET, ENUM, GEOMETRY and YEAR types and also to allow extra type qualifiers to be added, such as UNSIGNED, PRIMARY KEY, AUTO_INCREMENT, etc.

To get $createnames() to return one of these types, you might for example isolate a specific character string length and add a data type mapping for your new type.

Then whenever $createnames() encounters a string of that specific length, the mapping to your new type will occur. If you want to implement the SET data type, you could insert a new mapping entry between the first and second default entries, adjusting the length parameters as shown below:

char	simple	254	VARCHAR($)
char	simple	255	SET(‘One’,’Two’,’Three’,’Four’)
char	national	255	NATIONAL VARCHAR($)

Alternatively, you could dedicate the ‘national’ character subtype for your custom data types, leaving the ‘simple’ character subtype for standard character mappings.

MySQL to Omnis

The following mappings are hard-coded and cannot be altered.

MySQL Column Type	Range	OmnisType/Subtype
NUMBER		
BIT/BOOL/TINYINT(1)		Boolean
TINYINT	(-128..+127)	Integer 32 bit
SMALLINT	2^16 (-32768..+32767)	Integer 32 bit
MEDIUMINT	2^24 (-8388608..+8388607)	Integer 32 bit
INT/INTEGER	2^32 (-2147483648..+2147483647)	Integer 32 bit
BIGINT	2^64 (-2^63..+2^63-1)	Integer 64 bit
FLOAT		Num floating dp
DOUBLE/REAL		Num floating dp
DEC/DECIMAL/NUMERIC	(precision<=9 & scale<=2)	ShortNum 0..2dp
DEC/DECIMAL/NUMERIC	(precision>9 or scale>2)	Num 0..14dp
DATE/TIME		
DATE		Datetime (#FDT)
DATETIME		Datetime (#FDT)
TIMESTAMP		Datetime (#FDT)
TIME		Datetime (#FDT)
YEAR		Integer 32 bit
CHARACTER		
CHAR	1 to 255 bytes fixed	Character
VARCHAR	0 to 255 bytes varying	Character

MySQL Column Type	Range	OmnisType/Subtype
TINYTEXT	255 (2^8 – 1) bytes	Character
TEXT	65535 (2^16 – 1) bytes	Character
MEDIUMTEXT	16777215 (2^24 – 1) bytes	Character
LONGTEXT	4294967295 (2^32 – 1) bytes	Character
ENUM		Character
SET		Character
BINARY		
TINYBLOB	255 (2^8 – 1) bytes	Binary
BLOB	65535 (2^16 – 1) bytes	Binary
MEDIUMBLOB	16777215 (2^24 – 1) bytes	Binary
LONGBLOB	4294967295 (2^32 – 1) bytes	Binary

MySQL Troubleshooting

The following points may help in resolving programming issues encountered using MySQL session and statement objects. For additional updated trouble shooting issues, refer to the readme file which accompanies your Omnis download

- $sqlstripspaces has no effect for MySQL sessions. MySQL automatically strips trailing spaces from data inserted into CHAR and VARCHAR columns. Character data returned from the server will already be stripped of trailing spaces.
- MySQL 4.1 does not support chunking of fetched (output) LOB data (TEXT and BLOB types). Chunking of input LOB data is supported.
- $rowcount will be –1 following execution of a SELECT, SHOW or EXPLAIN statement. This is because MySQL cannot determine this value until the final row has been fetched.
- The MySQL DAM is compatible with MySQL Server version 4.1 and later. Prior to Omnis Studio version 4.1, connection is only possible to a commercial version of MySQL Server (i.e. MySQL “pro” or “classic”). Later versions of Studio do not have this restriction.
- Authentication plug-in “caching_sha2_password” cannot be loaded. This error occurs attempting to connect to MySQL server 8.0 and later using an imcompatible client library. In Studio 10.2 and later, refer to technote TNSQ0039 for details on applying an external MySQL client library.

Further troubleshooting notes, “how-tos” and tips can be found on the Omnis website at: https://www.omnis.net/developers/resources/technotes/

ODBC

This section contains the additional information you need to access a database using ODBC middleware, including server-specific programming, data type mapping, as well as troubleshooting. For general information about logging on and managing your database using the Omnis SQL Browser, refer to the earlier parts of this manual.

Properties and Methods

In addition to the “base” properties and methods documented in the *SQL Programming* chapter, the ODBC DAM provides the following additional features.

Session Properties

Property	Description
$dbmsname	Once a session has been established this is the name of the database that the object is connecte $logoff. (Read only)
$dbmsversion	Once a session has been established this is the version of the database that the object is connect $logoff. (Read only)
$defaultdatabase	When set, the session will attempt to log on to the database specified. A change to $defaultdatab logging on, otherwise the change will not take effect until the session is re-used. To stop using a session, set $defaultdatabase to an empty string (the default value). This property may not be sup

Property	Description
$drivername	Once a session has been established this is the name of the ODBC driver that the object is using. (Read only)
$driverversion	Once a session has been established this is the version of the ODBC driver that the object is usin (Read only)
$driverodbcversion	Once a session has been established this is the version of the ODBC API that the driver supports. (Read only)
$fetchnumericaschar	If kTrue, NUMERIC & DECIMAL columns are defined and fetched as CHAR(64). Use this property t driver-specific issues when fetching numeric values.
$infoaserror	If kTrue (the default), execution results that report SQL_SUCCESS_WITH_INFO are reported as err this the same as SQL_SUCCESS and ignores the accompanying message. Studio 5.2 and later.
$logontimeout	The timeout in seconds for a $logon() call. The default is 15 seconds. A value of 0 represents no tin be specified to indicate that the DAM should not attempt to set a timeout value.
$mode	macOS and Linux only. A *kODBCMode*...value used to select the ODBC driver manager library for connections, e.g. SAP SQL Anywhere.
$odbencrypt	If kTrue (the default) ODBC Bridge connections use end-to-end encryption. Improved network pe by disabling encryption. The ODBC Bridge uses the value that is in effect when $logon() is called, called, fetch results will still be encrypted for the duration of the connection even if $odbencrypt i
$programname	The name to be registered at the server for the process associated with the session. By default, $p current library name. This property may not be supported by all DBMS vendors: see the $useprog
$querytimeout	The timeout in seconds for a query. A value of 0 represents no timeout, which is the default.
$savefile	Used in conjunction with $usefiledsn. If kTrue, invokes the SAVEFILE ODBC connection attribute connection details back to the specified File DSN.
$timezone	The local timezone offset relative to GMT. The initial value is read from the OS when the session ol the form "+/-HH:MM". See $usetimezone.
$trustedconnection	Supported values are kODBCIgnoreTrusted (the default,) kODBCUseTrusted and kODBCNotTrust kODBCUseTrusted is specified, the session attempts to log on to the DBMS using a server trusted $username and $password will be ignored. When a value of kODBCNotTrusted is specified, the s the DBMS with an explicitly non-trusted connection. This property may not be supported by all D enforce trusted connections, it may be necessary to disable server prompting by setting $uselogo
$usefiledsn	If kTrue, the hostname specified at logon will be treated as a file DSN. The default is kFalse. Not al file DSNs.
$uselogonprompt	Governs the use of logon prompts where there is insufficient information to connect Can also use Administrator library to display a configuration dialogue when connecting to File DSNs. $uselogo values of: kODBCPromptNever (0), kODBCPromptComplete (1), kODBCPromptAlways (2). And kC Not all drivers can support this feature.
$useprogramname	If kTrue, the session attempts to register $programname as the process name when logging on t may not be supported by all DBMS vendors. Hence the default value for $useprogramname is kF program name can be found in the sysprocesses table of the master database.
$usequalifiers	When set to kTrue, the DAM treats qualified table names as owner.tablename. To prevent this, fo file driver, set $usequalifiers to kFalse. This property affects the behaviour of the $columns(), $tab session methods. $usequalifiers is ignored until the session logs on, at which time the default val $usequalifiers is overwritten.
$usescale	If kTrue, Omnis number dp columns are bound using a precision of 15 + the dp value. If kFalse (th columns are bound using a precision of 15. Also affects $createnames(). Studio 8.1.5 and later.
$usetimezone	If kTrue, $timezone will be applied to values inserted and fetched from TIME & TIMESTAMP colum are subject to modification by comparing the local timezone with the server's timezone. Applies t connections only.
$usevarcharmax	If kTrue, Character columns > 4000 map to VARCHAR(MAX) / NVARCHAR(MAX) dependent on the they map to TEXT / NTEXT. Assumes the connection is to MS SQL Server.
$nationaltowchar	Available only with the Unicode DAM. By default, Omnis Character and National fields are mappe SQL_WVARCHAR and SQL_WLONGVARCHAR data types. By setting $nationaltowchar to kTrue o mapped to these types (to the equivalent server data types) and *Character* fields will be mapped SQL_VARCHAR and SQL_LONGVARCHAR as determined by the Omnis field length. Character fie subject to data loss/truncation where such fields contain Unicode characters. When setting this p Unicode data types usually have precision limits half that of their corresponding ANSI data types. the SQL Server VARCHAR() data type but 4000 for NVARCHAR(). $nationaltowchar affects both th $createnames() method and the binding of input parameters.
$datesecdp	The $datesecdp property specifies the number of decimal places used for server date columns. F Server 2008 TIME and DATETIMEOFFSET data types which include a scale parameter. $datesecd by $createnames() as well as input binding. Defaults to 2 but valid values are in the range 0 to 7. T a MyODBC connection to allow correct type mapping to DATETIME.

Property	Description
$defaultschema	For use with Microsoft SQL Server 2005 and later. $defaultschema returns the schema name whi the current user. This should be used in place of username in methods such as $tables(). Assignin ALTER USER statement which changes the default schema for the user.

A number of additional session properties have been added to the ODBC DAM in Studio 4.3.1 to facilitate better understanding and control of cursors and transactions. Use of these properties assumes that the session is logged-on and has been placed in manual transaction mode (kSessionTranManual):

Property	Description
$autobegintran	(Read Only). This property always returns kTrue for DAMODBC because the ODBC API implicitl even in manual transaction mode.
$cursorsensitivity	(Read Only). This property returns kTrue if SQL cursors are sensitive to changes made by other transaction. kFalse is returned if results returned by cursors are not sensitive to changes made same transaction.
$txncapability	(Read Only). Returns one of the constant values listed in the Catalog under ODBCDAM-Transac drivers only support use of DML statements within transaction blocks (SELECT, INSERT, UPDAT ignore or permit use of DDL statements (CREATE TABLE, DROP INDEX, and so on) but may rec block to be committed immediately. kODBCTxnAll specifies that transactions can contain DDL statements in any order.
$multipletransactions	(Read Only). Returns kTrue if the driver supports more than one active transaction at the same transaction can be active at any time.
$multipleresultsets	(Read Only). Returns kTrue if the data source supports multiple result sets, kFalse if it does not.
$isolationoptions	(Read Only). Returns a bitmask value representing the transaction isolation levels supported b positions correspond to the constant values listed in the Catalog under ODBCDAM-Isolation Le
$isolationlevel	Returns the current transaction isolation level in use by the session. To change the isolation lev constants listed in the Catalog under ODBCDAM-Isolation Levels. The isolation level must be o by $isolationoptions. Changing the isolation level implicitly invokes a $commit().

Session Methods

Method	Description
$getdrivers()	SessObj.$getdrivers(lResult) retrieves a list of all ODBC drivers installed on the system. lResult is popu installed and is defined with the following character columns: DriverName The alternate driver name The version string reported by the driver CompanyName The Company name embedded within the path and file name of the driver CompanyName is only obtainable for Win32 and will return as empty is obtainable directly from the driver only for Win32. Other platforms require each driver to be loaded version string. Hence, there will be a commensurate delay when calling this method. Example: `Do sessObj.$getdrivers(iDriverList) Returns #F`
$getdatasources()	SessObj.$getdatasources(lResult, kDSNMode) retrieves a list of ODBC DSNs of type specified by kDSN either kODBCSystemDSN or kODBCUserDSN. $getdatasources() does not support File DSNs (see bel with two character columns: DSNName The User assigned name for the data source Driver The altern with the data source Example: `Do sessObj.$getdatasources(iDSNList,kODBCSystemDSN) Ret`
$getinfo()	$getinfo(lResult, cDSNName, kDSNMode) retrieves the information defined for the specified data sou keyword-value pairs. kDSNMode should be passed as either kODBCSystemDSN, kODBCUserDSN or k not support File DSNs for which standard FileOps methods can be used to read/modify as required. C the following character columns: KeyWord The name of the DSN/driver attribute Value The value of th `Do sessObj.$getinfo(iDSNinfo,'myDsn',kODBCUserDSN) Returns #F`
$setinfo()	$setinfo(cDSNName, kDSNMode, lData) writes the information contained in lData to the specified Da system information. lData should be defined with Keyword and Value columns as returned by $getin kODBCDriverInfo, this has the effect of modifying system information for the specified driver. cDSNN descriptive name of the ODBC Driver as opposed to the physical file name. If kDSNMode is kODBCSy has the effect of modifying the specified data source. $setinfo() does not register a new data source o to the DSN as though it already exists. To properly create a data source; use the $configdsn() method driver, you should refer to the vendor's installation program. Example: `Do sessObj.$setinfo('myD Returns #F`

Method	Description
$configdsn()	$configdsn(kDSNMode, kRequestType, cDriverName, lAttributes) allows the specified datasource to b kDSNMode should be either kODBCSystemDSN or kODBCUserDSN. $configdsn() does not support c which an alternative strategy is provided. kRequestType should be passed as either kODBCAddDSN, kODBCRemoveDSN. cDriverName should correspond with the descriptive name of the driver (i.e. not lAttributes should be defined with two character columns and is used to pass keyword-value pairs to perform the required action. Usually this involves adding a single line to the list to identify the DSN to e.g. *KeyWord Value* DSN dsnname but can also include other keywords that are allowed by the driver to kODBCPromptNever, this prevents $configdsn() from opening setup dialogues. The DSN is created read from the attribute list instead. Example: `Do sessObj.$configdsn(kODBCUserDSN,kODBCAdd Server’,lAttribList) Returns #F`
$getoption()	$getoption(kOption, cAttribute) allows the value of an ODBC configuration attribute to be retrieved. k of the following constants:kODBCTrace Requests the TRACE on/off flag kODBCTraceLib Requests the trace library kODBCTraceFile Requests the name and path to the ODBC trace log kODBCFileDSNDir containing file DSNs kODBCPerfMon Requests the Performance monitoring on/off flag kODBCRetry pool RetryWait timeout On return, cAttribute contains the value of the requested option as a charact `sessObj.$getoption(kODBCFileDSNDir,iFileDSNDir) Returns #F`
$setoption()	$setoption(kOption, cAttribute) allows the value of an ODBC configuration attribute to be modified. k kODBCTrace, kODBCTraceLib, kODBCTraceFile, kODBCFileDSNDir, kODBCPerfMon or kODBCRetryW character string representing the new value for the specified configuration option. Example: `Do sessObj.$setoption(kODBCTraceFile,iTraceFile) Returns #F`

Connecting to your Database

```
Do SessObj.$logon('MyDataSource', , , 'MySession') Returns #F
```

When the session property $usefiledsn is set to kTrue, this specifies that the hostname parameter is to be treated as a file data source name by the driver manager.

When the session property $uselogonprompt is set to kODBCPromptComplete, this specifies that the driver will prompt for missing logon information. Note that not all drivers support prompting and this may result in the logon failing.

Making a DSN-less Connection

To make a connection without using an ODBC DSN, all of the information necessary to make a connection needs to be passed as an *ODBC connection string* and the session $uselogonprompt property needs to set to kODBCPromptDsnLess.

The connection string is passed to the $logon() method via the hostname parameter. The username and password parameters are left blank.

ODBC connection strings consist of keyword-value pairs separated by semi-colons and are database specific. Examples of such strings include:

For SQL Server:

```
Driver=SQL Server; Server=192.168.0.10; Database=accounts; Uid=fred; Pwd=secret
```

For the Omnis ODBC Driver:

```
Driver=Omnis ODBC Driver; DataFilePath=c:\TRAVEL.DF1; Username=myuser; Password=mypassword
```

For specific details on connection strings, please refer to the documentation supplied with the RDBMS or with the ODBC driver.

Connecting using the ODBC Databridge

For macOS and Linux in particular, or to make an ODBC connection in the absence of a platform-specific ODBC driver or driver manager, you can “bridge” an ODBC connection across to a Windows PC which hosts the required ODBC driver necessary to complete the connection.

This process is described more fully in the ODBC Databridge documentation.
However, to make connection using the ODBC Databridge, you should use a URL of the form:

```
Do SessObj.$logon('odbc://192.168.0.22:8063/dsnName','user1','pwd') Returns #F
```

The above example assumes that the ODBC Databrige is running on the specified IP address using the default port, and that there is a User or System DSN named ‘*dsnName*’ defined on that machine.

Transactions

Generally, using manual transaction mode results in increased performance because the session object does not force a commit after each statement.

If you do not have a result set pending, ODBC session objects will commit each statement if the transaction mode is automatic. If the transaction mode is server, the session may be committed depending on the behavior of the ODBC driver.

Dates

The session property $defaultdate allows default values to be added to date values mapped to the server where the Omnis date value does not contain complete information, e.g. a Short time mapped to a server date time.

Multiple cursors

To allow multiple select cursors when connecting to Microsoft SQLServer the statement issuing the SELECT must have the $usecursor property set to kTrue before the statement is executed. If a statement is issued when $usecursor is kFalse and this statement returns a result set, this will prevent all other statements in the same session from returning data. The blocking results must be completely processed or cleared before another result set can be generated. If a commit or rollback is performed on the session, all the session's statement cursors will be closed and all pending results will be lost. Note that a SQLServer cursor only allows SQL SELECT and EXECUTE procedure commands to be issued.

SQL Server 2000 Data Types

The following new types were introduced with Microsoft SQLServer2000:

SQL_BIGINT

Values fetched into Omnis from BIGINT columns are converted into the Character 20 type. This is necessary since BIGINTs are stored in 64 bits, giving them a range of $\pm2^{63}$ or

-9223372036854775808 to 9223372036854775807. The largest numeric value which can be stored using the Integer 32 bit type is $\pm2^{31}$ or –2147483648 to 2147483647.

Omnis Character variables can be input into BIGINT columns provided that the character length (precision) does not exceed 19.

Note: Hash variables such as #S1 cannot be bound as input variables for BIGINT columns since their length is preset to 10,000,000.

SQL_VARIANT

Values fetched from SQL_VARIANT columns are converted into the Binary type so they can be preserved in their raw format.

Since the data type, precision and scale are not known prior to fetching, it may be necessary to pre-process the table before retrieving the variant data. This is done using the SQL_VARIANT_PROPERTY() function to build a list of variant types contained in a specified column.

```
Do myStatement.$execdirect('select cast(SQL_VARIANT_PROPERTY(col3,'BaseType') as char(32)) from mytable') r
```

When fetching the data into Omnis, the CAST() function can then be used inside the SELECT statement to ensure that the incoming data gets converted to the proper Omnis types.

```
Do myStatement.$execdirect('select CAST(col3 as smalldatetime) from mytable') returns #F
```

For this reason, tables containing SQL_VARIANTs are probably best used in tandem with an index column, which is used to associate a cast type with each row. Since there is no variant type in Omnis, there is no systematic way of reading a whole column of variants.

Note: text, ntext, image & timestamp types are not supported by SQL_VARIANT.

Custom Data Types

When custom data types are fetched from SQL Server their base type is abstracted from the custom type and returned to Omnis. For example, if a custom data type is created using;

```
Do myStatement.$execdirect('EXEC sp_addtype birthday, datetime, 'NOT NULL') returns #F

Do myStatement.$execdirect('create table test(col1 integer,col2 birthday)') returns #F
```

Then, a $columns() performed on the table, describes col2 as DATETIME

If a custom data type is specified when creating or altering a table, this is passed straight through the DAM.

TABLE

The TABLE data type can be used in two ways.

The first is as a local variable in a user SQL function. Local variables are defined as type **table** and can be used to temporarily store the result of a query. This usage is beyond the scope of Omnis however.

The second is as the **return value** from a user-defined function. For example, the following function defines a table as it's return value. (The table must be defined in the RETURNS section.):

```
CREATE FUNCTION Function1 ( @Param integer )
RETURNS @myTable TABLE
  (
    col1 integer, col2 char(32)
  )
AS
BEGIN
  INSERT @myTable
    SELECT * FROM valuetest WHERE col1 > @Param
  RETURN
END
```

Omnis can then call the function to obtain the table results, e.g.:

```
Do mylist.$define(col1,col2)
Do myStatement.$execdirect('select * from Function1(50)') returns #F

Do myStatement.$fetch(myList,kFetchAll)
ODBC Administration
```

ODBC Administration

The $getdrivers(), $getdatasources(), $getinfo(), $setinfo(), $configdsn(), $getoption() and $setoption() session methods (documented above) allow the ODBC DAM to be used to add, modify and remove ODBC Data Source Names (DSNs) as well as to retrieve and modify information about ODBC drivers and general ODBC administration attributes.

The $uselogonprompt property has been modified to allow driver prompting to be forced if required.

All of these methods return a boolean value to indicate successful operation. Errors generated by these methods are returned as normal via the session object's $nativeerrorcode and $nativeerrortext properties.

Configuration of File DSNs

Using $getoption() to retrieve the default directory for file DSNs allows the FileOps component to be used together with other 4GL techniques in configuring File datasources.

To create a File DSN, you should prompt for the new filename (the DSN Name) and use FileOps to create the new file. One or more KeyWord-value pairs should also be written to the file, e.g. DRIVER=drivername- the minimum requirement for a File DSN. To complete the setup of the new DSN, you should follow the procedure for modifying/testing the File DSN.

To modify or test a File DSN, use the following procedure:

- Set $usefiledsn to kTrue
- Set $savefile to kTrue
- Set $uselogonprompt to kODBCPromptAlways
- Execute $logon() with the name of the File DSN as the hostname.

Under Win32, this prompts the ODBC Administrator library to display the driver specific logon dialogue which prompts for information necessary to make the connection. If the DAM is successful in logging on (and $savefile is set to kTrue), the Administrator library writes the additional information back to the File DSN, hence modifying the datasource.

To remove a File DSN, you should use the FileOps component to manually delete the specified filename.

macOS and Linux Considerations

Under Unix, the DAM locates user DSN information (odbc.ini) using the value of the ODBCINI environment variable if is set. If ODBCINI is not set, the DAM attempts to use the ".odbc.ini" (hidden file) in your user's home directory or failing that, it defaults to "/Library/ODBC/odbc.ini".

The DAM locates system DSN information using the value of the ODBCSYSINI environment variable if it is set. If ODBCSYSINI is not set, the DAM attempts to locate the system driver information using the value of ODBCINSTINI instead. If ODBCSYSINI is set, this location is also assumed for the location of the driver information file (odbcinst.ini). Note that if set, ODBCSYSINI should identify a folder only- not a file name.

The ODBC Driver manager used on your system must support the necessary API calls in order to perform certain administration functions (editing and modifying DSN information, for example). If the required API calls are missing, these functions will not be available.

ODBC Troubleshooting

The following points may help in resolving issues in programming applications that use ODBC session objects.

- ODBC does not support any extended ORACLE cursor operations such as positioned update and delete.
- You must specify literals in SQL statements with single quotes ('), not double quotes (").
- Some data sources may strip trailing spaces prior to sending it to the session object. SQL Server behaves in this way.

Further troubleshooting notes, "how-tos" and tips can be found on the Omnis website at: https://www.omnis.net/developers/resources/technotes/

ODBC Data Type Mapping

The following table describes the data type mapping for Omnis to ODBC connections.

The Omnis to ODBC mapping will attempt to pick the best match based on the types the driver supports in the order listed. For example, if the driver supports SQL_VARCHAR and SQL_CHAR data up to a maximum column size of 255, but SQL_LONGVARCHAR data up to 2 GB, an Omnis Character(1000) will map to whatever the associated server native type is for SQL_LONGVARCHAR, e.g. TEXT.

Omnis to ODBC

Omnis Data Type	ODBC Data Type
CHARACTER	
Character(n) National(n)	[1]SQL_VARCHAR(n) [1]SQL_CHAR(n) [1]SQL_LONGVARCHAR(n) SQL_CLOB(n) (DB2 o
DATE/TIME	
Short date (all subtypes)	SQL_DATE SQL_TYPE_DATE SQL_TIMESTAMP SQL_TYPE_TIMESTAMP
Short time	SQL_TIME SQL_TYPE_TIME SQL_TIMESTAMP SQL_TYPE_TIMESTAMP
Date time (#FDT)	SQL_TIMESTAMP SQL_TYPE_TIMESTAMP
NUMBER	
Short integer (0 to 255)	SQL_TINYINT (unsigned) SQL_SMALLINT
Integer 64 bit	SQL_BIGINT SQL_CHAR(20)
Integer 32 bit	SQL_INTEGER
Sequence	SQL_NUMERIC(10,0) SQL_DECIMAL(10,0) SQL_ SQL_DOUBLE
Short number 0-2dp	[2]SQL_NUMERIC(p,s)
Number floating dp, 0..14 dp	[2]SQL_DECIMAL(p,s) SQL_FLOAT SQL_DOUBL
OTHER	
Boolean	SQL_BIT SQL_TINYINT SQL_SMALLINT SQL_NU SQL_DECIMAL(1,0) SQL_CHAR(1) SQL_VARCHA SQL_FLOAT

Omnis Data Type	ODBC Data Type
Picture, Binary, List, Row, Object, Item reference	SQL_VARBINARY(blobsize) SQL_BINARY(blobsi SQL_LONGVARBINARY(blobsize) SQL_BLOB(bl only) Where blobsize is SessObj.$blobsize

[1] Refer to the $nationaltowchar property for use with the Unicode version of Omnis Studio
[2] As of Studio 8.1.5, $usescale can be used to calculate p as 15 + s. E.g. In this mode a Number 14dp will map to NUMERIC(29,14) giving 15 scalar digits plus 14 mantissa digits.

ODBC to Omnis

ODBC Data Type	Omnis Data Type
CHARACTER	
SQL_CHAR(n) SQL_VARCHAR(n) SQL_LONGVARCHAR(n) SQL_WCHAR(n) SQL_WVARCHAR(n) SQL_WLONGVARCHAR(n) SQL_CLOB(n)	Character(n)
DATE/TIME	
SQL_DATE SQL_TYPE_DATE	Short date 1980
SQL_TIME SQL_TYPE_TIME	Short time
SQL_TIMESTAMP SQL_TYPE_TIMESTAMP	Date time (#FDT)
NUMBER	
SQL_DECIMAL(p,s)	Number (s)dp (p<=15)
SQL_NUMERIC(p,s)	Number floating dp (p>15)
SQL_SMALLINT	Integer 32 bit
SQL_TINYINT (unsigned)	Short integer
SQL_TINYINT (signed)	Integer 32 bit
SQL_INTEGER	Integer 32 bit
SQL_BIGINT	Integer 64 bit
SQL_REAL SQL_FLOAT SQL_DOUBLE	Number floating dp
SQL_BIGINT	Character 20
OTHER	
SQL_BIT	Boolean
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY SQL_BLOB SQL_GUID SQL_VARIANT	Binary
Custom Data Types	N/A
Table Type	N/A

Amazon SimpleDB

The Amazon DAM (DAMAZON) allows you to access the SimpleDB from Amazon Web Services LLC. According to Amazon, “SimpleDB is a highly available, scalable, and flexible non-relational data store that offloads the work of database administration. Developers simply store and query data items via web services requests, and Amazon SimpleDB does the rest.” For further information about Amazon SimpleDB, please refer to the Amazon SimpleDB website:

- **General information**
  http://aws.amazon.com/simpledb
- **Developers Guide**

  http://docs.amazonwebservices.com/AmazonSimpleDB/latest/DeveloperGuide/

This section also discusses various topics which differentiate cloud-based connectivity from traditional RDBMSs and the impact this has on the various properties and methods.

Dependencies

The Amazon DAM has runtime dependencies on several other dynamic libraries which must be present on your system's library search path before the DAM can be used. When a DAMAZON session object is created, the DAM attempts to locate and resolve the symbols it needs from each of the external libraries.

If one or more symbol references cannot be resolved, these are reported to the Omnis trace log as warnings, $logon() is disabled and you should not attempt to call session or statement methods, otherwise a crash may occur.

The additional files required by the Amazon DAM for each platform are as follows:

Windows

libcurl.dll (requires msvcr90.dll)

libeay32.dll (requires msvcrt.dll)

libxml2.dll (requires iconv.dll & zlib1.dll)

macOS

libcurl.dylib (where libcurl.dylib -> /usr/lib/libcurl.4.dylib, for example)

libcrypto.dylib (where libcrypto.dylib -> /usr/lib/libcrypto.0.9.7.dylib, for example)

libxml2.dylib (where libxml2.dylib -> /usr/lib/libxml2.2.dylib, for example)

Linux

libcurl.so (/usr/lib/libcurl.so)

libcrypto.so (/usr/lib/libcrypto.so)

libxml2.so (/usr/lib/libxml2.so)

If these libraries are not present on your system, the appropriate package(s) may need to be installed or alternatively, downloaded and compiled from source. The principal libraries shown are all available under open source licence agreements.

For developers interested in downloading and compiling client libraries from source, information about each of the projects can be obtained from:

libcurl: http://curl.haxx.se/

libxml2: http://xmlsoft.org/

libcrypto/libeay32 : http://www.openssl.org/ (Links accurate at time of publishing)

Binary releases of these libraries may also be available to download from these and other sources.

Logging on to SimpleDB

To connect to SimpleDB, the *endpoint* required is supplied via the $logon() hostname parameter. In the case of Amazon SimpleDB, the endpoint is "**sdb.amazonaws.com**" or "**sdb.eu-west-1.amazonaws.com**" in Europe.

Your *access key id* and *secret* are supplied via the username and password parameters, for example:

```
Do SessObj.$logon('sdb.amazonaws.com',' AGIBJ5LOYFITD3BR7',' H/z6t3ARzuJL26uIE07 GTS1AkK+p5') Returns #F
```

For other databases, the endpoint may be specified using http syntax, for example:

```
Do SessObj.$logon('http://www.remoteserver.com/?','user_1','password') Returns #F
```

If the hostname parameter is omitted, i.e. substituted with a comma, the DAM uses sdb.amazonaws.com by default.

Meta Data

SimpleDB does not provide information about tables, columns and indexes in the same way as traditional relational databases. Instead, *domains* can be likened to tables, *items* can be likened to rows and *attributes* can be likened to columns. This has an impact on the behavior of the following meta-data methods:

Method	Description
$tables()	StatObj.$tables() returns a list of available domain names in the TableOrView column of the result set. Other resu SimpleDB does not support views.
$columns()	StatObj.$columns(cDomain) returns meta data information about the specified domain. This information is spec via the DamInfoRow column of the result set. Other result columns can be ignored.
$indexes()	StatObj.$indexes() is not implemented since SimpleDB handles indexing automatically.

The information returned by $columns() for a domain is summarised as follows:

Column	Description
Timestamp	The date and time when metadata was calculated in Epoch (UNIX) time.
ItemCount	The number of all items in the domain.
AttributeNameCount	The number of unique attribute names in the domain.
AttributeValueCount	The number of all attribute name/value pairs in the domain.
ItemNamesSizeBytes	The total size of all item names in the domain, in bytes.
AttributesValuesSizeBytes	The total size of all attribute values, in bytes.
AttributeNamesSizeBytes	The total size of all unique attribute names, in bytes.

SimpleDB Attributes and Multi-Values

Unlike Relational databases, SimpleDB attributes support multiple values. For example:

Domain	Item	Attribute	Value
Suits	Gents Formal Suit	Colour	Navy
Suits	Gents Formal Suit	Colour	Black
Suits	Gents Formal Suit	Colour	Grey

In addition, SimpleDB effectively supports only a single data type: Character. All data inserted into and retrieved from SimpleDB will be character data optionally encoded as UTF-8 bytes. Once fetched into Omnis, data can be assigned to typed variables as required. Such data will be automatically converted to the appropriate data type where possible.

Each item fetched from SimpleDB can potentially have a different number of attributes and attribute names. This prevents the use of Omnis Schema classes with SimpleDB since these require rigid column names and types. When dragging a schema class onto a SimpleDB session in the Omnis SQL Browser, all that can sensibly be achieved is to create a domain with the supplied *table name*.

SimpleDB does not support SQL in the traditional sense. You cannot use $prepare() & $execute() or $execdirect() to execute INSERT, UPDATE or DELETE statements as these are not supported. Instead, these statement methods can be used only to execute SELECT statements conforming to the SimpleDB SELECT syntax.

Creating a Domain

To manually create a domain (analogous to a *table*), use the StatObj.$createdomain() method. For example:

```
Do StatObj.$createdomain('Project810') Returns #F
```

Inserting Data

To insert items and attributes into SimpleDB, use the StatObj.$putattrib() method.

Each call to $putattrib() inserts a new attribute-value pair into the specified domain item. (There is no need to create the item before inserting an attribute, the item is created implicitly). Since SimpleDB supports multiple attribute values, you can assign several different values to the same attribute if required. Duplicate values are ignored. For example:

```
Do StatObj.$putattrib('Project810','Materials','Tools','13mm Wrench') Returns #F
Do StatObj.$putattrib('Project810','Materials','Tools','Quick release clamps') Returns #F
```

If many attributes are to be inserted, it may be preferable to assign the domain name to the StatObj.$domain property and the item name to the StatObj.$item property. These parameters can subsequently be omitted in calls to $putattrib()- and any of the other statement methods discussed below. The above example becomes:

```
Do StatObj.$domain.$assign('Project810')
Do StatObj.$item.$assign('Materials')
Do StatObj.$putattrib(, ,'Tools','13mm Wrench') Returns #F
Do StatObj.$putattrib(, ,'Tools','Quick release clamps') Returns #F
```

Deleting Data

To delete items, attributes and values from SimpleDB, use the StatObj.$delete() method.

Deleting Values

To delete a specific attribute value, the domain, item, attribute name and value should be specified. For Example:

```
Do StatObj.$delete('Project810','Materials','Timber','50x50x2.4m pse') Returns #F
```

Deleting Attributes

To delete an attribute including all of its values, the domain, item and attribute name only should be specified. For example:

```
Do StatObj.$delete('Project810','Materials','Timber') Returns #F
```

Deleting Items

To delete an entire item including all its attributes and values, the domain and item name only should be specified. For example:

```
Do StatObj.$delete('Project810','Materials') Returns #F
```

Deleting a Domain

StatObj.$delete() cannot be used to delete a domain. To do this- use StatObj.$deletedomain(). This method should be used with caution as it will permanently delete all items, attributes and values contained in the domain before removing the domain itself. For example:

```
Do StatObj.$deletedomain('Project810') Returns #F
```

Replacing Data

Whereas $putattrib() is used to append new attributes and values, StatObj.$replaceattrib() is used to replace all values for a specified attribute with the supplied single value. For example:

```
Do StatObj.$replaceattrib('Suits','Gents Formal Suit','Colour','Navy only') Returns #F
```

Fetching Data

The Amazon DAM uses Amazon SELECT statements to fetch multiple items. These are issued using the statement object's $prepare(), $execute() and $execdirect() methods in a similar way to traditional SQL SELECT statements. The general form of a SimpleDB SELECT statement is as follows:

```
select output_list from domain_name [where expression] [sort_instructions][limit limit]
```

The output_list can be:

- * (all attributes)
- itemName() (the item names only)
- count(*)
- An explicit list of attributes (attribute1,..., attributeN)

For further information on the SELECT statement syntax, please refer to Amazon SimpleDB Developer Guide.

Items in the result set are returned one row-at-a-time. StatObj.$resultspending indicates whether there is a further item each time a call to StatObj.$fetch() is made and StatObj.$itemcount is initially set to the number of items in the response. The destination list or row variable is automatically redefined each time $fetch() is called. For example:

```
Do StatObj.$execdirect('select * from Suits where stocklevel > 1') Returns #F
Repeat
  Do StatObj.$fetch(lvRow)
  …
Until StatObj.$resultspending = kFalse
```

Retrieving an Item

You can retrieve all attributes for a specific item using the StatObj.$getall() method. The result set (a single row) generated by this call is returned using $fetch(). For example:

```
Do StatObj.$getall('Suits','Gents Suits') Returns #F
Do StatObj.$fetch(lvRow)
```

Retrieving Item Names

You can retrieve the names of items contained within a domain by calling the StatObj.$getitems() method. The result is returned as a single item containing a single attribute. The item names will be returned either as a comma-separated list or as a single column list- as dictated by the $attribcsv property.
A SELECT where-clause may be optionally specified if required, in which case only the names of items which satisfy the expression will be returned. For example:

```
Do StatObj.$getitems( ,"where Colour like 'Red%'") Returns #F
Do StatObj.$fetch(lvItems)
```

Retrieving an Attribute

You can retrieve the contents of a specific attribute using the StatObj.$getattrib() method. The result set (a single row containing a single column) generated by this call is also returned using $fetch(). For example:

```
Do StatObj.$getattrib('Project810','Materials','Tools') Returns #F
Do StatObj.$fetch(lvRow)
```

Handling Multiple Values

When fetching data, each row returned to Omnis represents one item from the specified domain. Item attributes containing multiple values are handled in one of two ways; either as single-column lists or as comma-separated values as dictated by the StatObj.$attribcsv property.

When $attribcsv is set to kTrue (the default), rows fetched from SimpleDB will be defined with Character columns. Attributes (columns) with multiple values will be returned as a string of comma-separated values.

When $attribcsv is set to kFalse, rows fetched from SimpleDB will contain single-column lists in each column. Each single-column list will contain one row for each attribute value.

Handling Multiple Attributes

You can put, delete and replace several attribute values at once using the StatObj.$putmany(), StatObj.$deletemany() and StatObj.$replacemany() methods. The attribute-value pairs to be processed are supplied via a list variable defined with two character columns. Column 1 contains the attribute names, column 2 contains the corresponding values. For example:

```
Do myList.$define(lvChar1, lvChar2)
Do myList.$add('Tools','Posidrive screwdriver')
Do myList.$add('Tools','Metal hammer')
Do myList.$add('Charges','1½ hours labour')
Do StatObj.$putmany( , , myList) Returns #F
```

You can retrieve the values of multiple attributes using the StatObj.$getmany() method. The attribute names to be retrieved are supplied via a *single*-column list, for example:

```
Do myList.$define(lvChar1)
Do myList.$add('Tools')
Do myList.$add('Materials)
Do myList.$add('Charges')
Do StatObj.$getmany( , , myList) Returns #F
```

Each subsequent call to $fetch() returns a row containing separate attribute- either as a comma-separated-value or as a single column list, as dictated by $attribcsv.

Handling Multiple Items

When executing queries, the StatObj.$itemcount property is set to the number of items in the response- implying that each call to $fetch() retrieves one item.

When the response contains only attribute values, $itemcount will be set to zero.

Handling Multiple Requests

The SimpleDB DAM uses the transaction management features of the DAM interface to allow multiple requests to be executed as a combined batch of requests. To enable multiple-execution, the SessObj.$transactionmode property should set to kSessionTranManual.

In this mode, actions such as $createdomain(), $putattrib(), $getattrib(), $replacemany() and $execdirect() are accepted unconditionally into a queue. Nothing is sent to or received from the database until a SessObj.$commit() is executed, at which point each request is submitted in turn.

Unlike single request execution, every multiple request generates a response. Although actions to put, create, replace and delete attributes will return empty responses, this enables any errors and execution information associated with each action to be returned. For example:

```
Do cSess.$begin()
Do cStat.$putattrib(,,'Materials','White Paint') Returns #F
Do cStat.$putattrib(,,'Materials','Cement 25Kg') Returns #F
Do cStat.$replacemany(,,lvAttribList) Returns #F
Do cStat.$execdirect('select * from Project810') Returns #F
Do cSess.$commit() Returns #F
```

Handling Multiple Responses

When in manual transaction mode, each call to $commit() generates one or more responses. The number of responses available is returned via the SessObj.$responses property.

When $commit() is executed, StatObj.$itemcount and StatObj.$columncount are set to reflect the number of items and attributes in the initial response.

Items/attributes from the response are then retrieved using one or more calls to $fetch(). When all items/attributes from the current response have been retrieved, the StatObj.$endofresponse property is set to kTrue at which point, $itemcount and $columncount are also set to reflect the next response.

When fetching an empty response, note that $endofresponse will effectively remain set to kTrue. If the corresponding action generated an error, then StatObj.$nativeerrorcode and StatObj.$nativeerrortext will be set accordingly. Otherwise, the empty response (and empty row) can be discarded.

When all responses have been retrieved, the $resultspending property is set to kFalse, otherwise $resultspending remains set the kTrue while there are still responses waiting.

It is safe to abandon and/or replace multiple requests before executing them by simply calling SessObj.$begin() or changing the transaction mode back to kSessionTranAutomatic. You can also discard pending responses in this way.

$rollback() is not supported by the SimpleDB DAM- this has no effect.

Machine Utilization

Amazon SimpleDB measures usage of remote resources (and hence the charge it imposes on the end-user) in terms of "box usage". Each action sent to the database incurs a box usage- quoted as a decimal fraction of one hour. StatObj.$boxusage returns the box usage for each action which generates a response.

The *session* object also has a $boxusage property which accumulates a total box usage for the open connection. When retrieving multiple responses, the box usage for each response is received (and added) in turn.

$boxusage may not be supported by all Simple databases in which case, the value remains set to zero.

Read Consistency

Amazon SimpleDB supports two types of read consistency, defined as follows:

- **Eventually Consistent Reads**
  the eventual consistency option maximizes your read performance (in terms of low latency and high throughput). However, an eventually consistent read (using Select or GetAttributes) might not reflect the results of a recently completed write (using PutAttributes, BatchPutAttributes, DeleteAttributes). Consistency across all copies of data is usually reached within a second; repeating a read after a short time should return the updated data.
- **Consistent Reads**
  in addition to eventual consistency, Amazon SimpleDB also gives you the flexibility and control to request a consistent read if your application, or an element of your application, requires it. A consistent read (using Select or GetAttributes with ConsistentRead=true) returns a result that reflects all writes that received a successful response prior to the read.

The Amazon DAM implements this functionality using the $consistentread session property. When set to kFalse (the default setting), the eventual consistency option is used. When set to kTrue, all $getattrib() and SELECT statement results are fetched using consistent reads.

Conditional Puts and Deletes

The PutAttributes and DeleteAttributes API calls used by Amazon SimpleDB support conditional put and delete operations which enable you to insert, replace or delete values for one or more attributes of an item if the existing value of an attribute matches the value you specify. If the value does not match or is not present, the update is rejected. Conditional Puts/Deletes are useful for preventing lost updates when different sources write concurrently to the same item.

The Amazon DAM implements this functionality using the $whereclause statement property. This property affects all *put, replace* and *delete* attribute calls and accepts a SQL-style where clause of the form:

```
"where <name> [= <value>] [exists|does not exist]"
```

<name> and <value> can be literal values; in which case they must be double-quoted, or bind variables. Double quotes inside literal values should be escaped using \". For example:

```
Do cStat.$whereclause.$assign('where "Color" = "Light Brown"')
Do cStat.$whereclause.$assign('where "Undo" does not exist')
Do cStat.$whereclause.$assign('where "Project \"X\"" = @[lvChar]')
```

Once bound, variable values should be assigned before each call to $putattrib(), $delete(), etc:

```
Do cStat.$whereclause.$assign('where "Name" = @[lvChar]')
Calculate lvChar as "Brookes"
Do cStat.$putattrib('StockDB','Supplier1','Frequency','Daily') Returns #F
Calculate lvChar as "Robinson"
Do cStat.$delete('StockDB','Supplier2','Frequency') Returns #F
```

Currently, the *exists* condition may only be specified if both <name> and <value> attributes are also specified. To use *does not exist*, only the <name> attribute should be specified.

Subsequent calls to put, replace or delete attributes return kFalse if the condition is not met.

$whereclause is not affected by $clear(). To remove the where condition for a statement object; assign $whereclause to an empty string.

Session Properties

Property	Description
$boxusage	Returns the cumulative total of remote machine resources consumed since the session connected. Co box-usages from statement methods as well as box-usages from multiple actions (manual transaction
$consistentread	If set to kTrue, all read operations (e.g. $getattrib() & $fetch()) are executed guaranteeing that the resul updates are seen immediately. If set to kFalse, the default (faster) eventual consistency option is used.
$responses	Returns the number of responses generated by the last call to $commit(). Applies to manual transacti Read-only.
$transactionmode	Used to implement multiple request processing. When set to kSessionTranAutomatic each request is database immediately. When set to kSessionTranManual, requests and queued until a $commit() is ca

Session Methods

Method	Description
$begin()	Initialises/clears multiple responses in preparation for execution of a new batch of requests. Manual transaction m only.
$commit()	Executes a batch of statements and retrieves multiple responses from the database. Manual transaction mode on

Statement Properties

Property	Description
$attribcsv	If set to kTrue (the default), attributes with multiple values are returned as comma-separated values, i.e. row will be defined with character columns. If set to kFalse, attributes will be returned as single column fetched row will contain a single column list in each column.
$boxusage	For statement methods which generate a response from the database, $boxusage returns the portion c hour used to complete a particular request. See SessObj.$boxusage. Read-only.
$domain	The current domain name. $domain will be used with various statement methods if set. Statement met require a domain parameter will assume this value if the method parameter is omitted.
$endofresponse	Returns kTrue if the last item/attribute of the current response has been fetched in which case, $boxcou $itemcount and $columncount are set to reflect the next response. Read-only.
$item	The current item name. As with $domain, $item will be used with various statement methods if set. Sta methods which require an item name will assume this value if the method parameter is omitted. Wher an item list from the database, $item is also set to the name of the last item to be fetched.
$itemcount	Returns the number of items in the current response. Returns zero if the response contains only attribu information. Read-only.
$resultspending	Returns kTrue while there are still items/attributes waiting to be fetched from one or more responses. R
$whereclause	Affects all put, replace and delete attribute methods. This property accepts a SQL-style where clause of "where <name> [= <value>] [exists|does not exist]" <name> and <value> can be literal values; in which ca must be double-quoted, or bind variables. Double quotes inside literal values should be escaped using \

Statement Methods

Method	Description
$createdomain()	StatObj.$createdomain([cDomainName]) creates a domain with the specified name. $createdomain() StatObj.$domain if the parameter is omitted in which case, $domain must be predefined. Returns kTr otherwise.
$delete()	StatObj.$delete([cDomain],[cItem],[cAttrib],[cValue]) deletes an item, attribute or value from the specifi cItem are omitted, the values of StatObj.$domain and StatObj.$item are assumed in which case, $dom predefined. If cAttrib and cValue are omitted, the entire item is deleted.If cValue is omitted, the specifie Otherwise, the specified value only is deleted from the attribute. Returns kTrue on success, kFalse othe
$deletedomain()	StatObj.$deletedomain([cDomain]) deletes the specified domain and all associated items/attributes. W confirmation is sought before the domain is permanently deleted. Returns kTrue on success, kFalse ot
$deletemany()	StatObj.$deletemany([cDomain],[cItem],lAttribs) deletes one or more values from the domain item. Th supplied via lAttribs, which should be defined with two character columns. Column 1 contains the attri contains the corresponding value to be removed. Returns kTrue on success, kFalse otherwise.

Method	Description
$getall()	StatObj.$getall([cDomain],[cItem]) executes a query to return all attributes belonging to the specified i retrieved by calling StatObj.$fetch(). Returns kTrue on success, kFalse otherwise.
$getattrib()	StatObj.$getattrib([cDomain],[cItem],cAttrib) executes a query to retrieve the value(s) associated with t result of the query is retrieved by calling StatObj.$fetch(). Returns kTrue on success, kFalse otherwise.
$getitems()	StatObj.$getitems([cDomain],[cWhere]) executes a query to retrieve the item names contained within cWhere is specified, the text is appended to the SELECT statement. The result of the query is obtained Returns kTrue on success, kFalse otherwise. $getitems() is not supported by all database vendors.
$getmany()	StatObj.$getmany([cDomain],[cItem],lAttribs) executes a query to retrieve one or more named attribut attribute names are supplied via lAttribs, which should be defined with a single character column. The retrieved by calling StatObj.$fetch(). Returns kTrue on success, kFalse otherwise.
$putattrib()	StatObj.$putattrib([cDomain],[cItem],cAttrib,cValue) inserts a new attribute. If cAttrib already exists, th the existing value(s), otherwise a new attribute-value pair is created. Returns kTrue on success, kFalse o
$putmany()	StatObj.$putmany([cDomain],[cItem],lAttribs) inserts one or more values into the domain item. The att supplied via lAttribs, which should be defined with two character columns. Column 1 contains the attri contains the corresponding value. Returns kTrue on success, kFalse otherwise.
$replaceattrib()	StatObj.$replaceattrib([cDomain],[cItem],cAttrib,cValue) replaces all values for the specified attribute w Existing values are deleted. Returns kTrue on success, kFalse otherwise.
$replacemany()	StatObj.$replacemany([cDomain],[cItem],lAttribs) replaces one or more attributes in the domain item. supplied via lAttribs, which should be defined with two character columns. Column 1 contains the attri contains the new value. Existing values are deleted. Returns kTrue on success, kFalse otherwise.

Implementation Notes

Bind Variables

Queries issued using $execute() and $execdirect() may contain bind variables- for example in the where clause of the SELECT statement. The DAM inlines variable values into the SQL text each time $execute() is called, placing single quotes around each value. Values containing single quotes are escaped by adding a second single quote for each occurrence. For example:

```
Calculate lVar as "Katharine O'Hara"
Do StatObj.$execdirect("select * from Customers where Name = @[lVar]") Returns #F
```

becomes

```
select * from Customers where Name = ‘Katharine O’’Hara’
```

Multiple Statement Objects

The SimpleDB API does not facilitate statement isolation- only session isolation. This means that each session object may spawn only one statement object.

An attempt to spawn a second statement object will result in an error.

Remote Procedures

SimpleDB does not support remote procedures, views or triggers. These are features of traditional relational databases.

Binary Data

SimpleDB attributes support only character data of maximum length 1024 bytes and are not suitable for storing binary data directly. A better approach (the intended approach) is that attribute values be used to store URLs or unique identifiers for pictures, files and other media which exist externally to the database.

OmnisSQL DAM

The OmnisSQL DAM provides an object-oriented interface to the Omnis data manager. As such the OmnisSQL DAM is a wrapper around the single-threaded Omnis DML engine.

Note the OmnisSQL DAM is only provided for backwards compatibility with legacy apps only; Omnis Datafiles and the OmnisSQL DAM should not be used for new apps.

Logging on to an Omnis Data File

To connect to a data file using the OmnisSQL DAM, create an object variable of subtype "OMSQLSESS".

You connect to a data file using the $logon() method. The hostname parameter should be the full path to the data file.

OmnisSQL does not require a username or password, but you can specify a session name that will appear in the SQL Browser and in the Notation Inspector under *$sessions*.

OmnisSQL expects a DOS-style pathname under Windows and an absolute POSIX-style path under macOS and Linux. For example:

```
Do mySession.$logon('C:\mydata\mydatafile.df1','','','session1') Returns #F ## on Windows
Do mySession.$logon('/Users/MyUser/mydatafile.df1','','','session1') Returns #F ## on macOS / Linux
```

Using the Omnis Databridge (ODB)

To logon to a data file being hosted by the Omnis Databridge, the hostname parameter should consist of an ODB URL of the form odb://, for example:

```
Do mySession.$logon('odb://192.168.0.150:5913:osxlocking','','','session1') Returns #F
```

Although analogous to the Open data file command, note that there is no *internal-name* parameter when using $logon(). Instead, use the session name (parameter 4).

Omnis SQL Language Definition

Note: The following sections contain legacy information and have been reproduced from the deprecated Omnis_SQL_v2api.pdf document.

The following sections show the grammar of Omnis SQL using BNF (Backus-Naur Form) diagrams, using the conventions from the ANSI standard. Each statement includes a note specifying what parts, if any, of the statement depart from the ANSI 1989 standard for SQL.

SQL Statement

```
SQL_statement ::=
     create_table_statement
   | create_index_statement
   | delete_statement_searched
   | drop_index_statement
   | drop_table_statement
   | insert_statement
   | select_statement
   | update_statement_searched
   | update_statement_positioned
   | alter_table_statement
```

The SQL statement is the text that goes in the DAM's $prepare() or $execdirect() methods or in a statetement block starting with Begin statement. The rest of the grammar depends on this main element.

ANSI SQL has the following statements that Omnis does not implement. Most statements involve cursors, and Omnis implements these as commands rather than as SQL statements.

- **close_statement**
  closes a cursor (see the Close cursor, Quit cursor, and Reset cursors commands).
- **commit_statement**
  commits a transaction (see the Commit current session command).
- **declare_cursor**
  declares a cursor (see the Declare cursor command).

- **delete_statement_positioned**
  deletes a row based on current cursor position.

- **fetch_statement**
  fetches a row using the current cursor (see the Fetch commands).

- **open_statement**
  opens a cursor (see the Open cursor command)

- **create_schema_statement**
  creates a schema containing tables and views; Omnis SQL does not support schemas.

- **create_view_statement**
  creates a view; Omnis SQL does not support views.

- **grant_privilege**
  grants an access privilege on an object to a user; Omnis SQL does not implement any SQL security.

CREATE TABLE

```
create_table_statement ::=
   CREATE TABLE table ( table_element_comma_list )
   CONNECTIONS ( table_comma_list )
```

The CONNECTIONS clause is an Omnis extension to the ANSI standard that lets you specify a list of file classes to which to connect a file class. Connections are parent-child relationships between file classes.

```
table_element ::=
   column_definition | UNIQUE ( column_comma_list )
```

You can define a file class using the SQL CREATE TABLE statement. The fields in the format come from the list of column definitions. You can also specify that the values for a group of columns are unique, taken together, with the UNIQUE constraint. You can have more than one UNIQUE constraint. All the columns in a UNIQUE constraint must be defined with the NOT NULL qualifier (see below).

The ANSI standard contains several other table constraints, namely PRIMARY KEY, FOREIGN KEY and CHECK that Omnis SQL does not implement.

```
column_definition ::=
   column_data [ [ NOT ] NULL ]
```

The NOT NULL constraint specifies that when you insert a row, the value for this column must not be NULL.

The ANSI standard specifies a default clause that lets you define a default value for the column. It also lets you specify that the column is UNIQUE, REFERENCES a primary key in another table, or satisfies a CHECK constraint. Omnis SQL does not implement any of these features.

```
column_data ::=
   column_name data_type
data_type ::=
   [ LONG ] VARBINARY
   | BIT
   | VARCHAR ( NUMBER )
   | CHAR ( NUMBER )
   | NATIONAL CHAR[ACTER] VARYING (NUMBER)
   | NCHAR VARYING ( NUMBER )
   | SEQUENCE_TYPE
   | DATE [ ( { 1900..1999 | 1980..2079| 2000..2099 } ) ]
   | TIME
   | TIMESTAMP
   | TINYINT
   | SMALLINT
   | INTEGER
   | NUMERIC ( number, integer)
   | DEC[IMAL] ( number, integer)
   | FLOAT_TYPE [ ( integer ) ]
   | REAL
   | LIST
   | PICTURE
```

ANSI data types include CHARACTER, NUMERIC, DECIMAL, INTEGER, INT, SMALLINT, FLOAT, REAL, and DOUBLE PRECISION. Omnis does not implement FLOAT and DOUBLE PRECISION directly, though FLOAT_TYPE is similar to FLOAT.

The other data types are Omnis specific. The integer value in the NUMERIC, DECIMAL, and FLOAT_TYPE types corresponds to the Omnis subtypes for numbers; 0-8, 10, 12, and 14 are the possible values.

ALTER TABLE

```
alter_table_statement ::= ALTER TABLE table ADD
   { column_data | ( column_data_comma_list ) }
```

The ALTER TABLE statement lets you add a column to an already existing table using the same syntax as in CREATE TABLE.

The ALTER TABLE statement does not exist in the 1989 ANSI standard.

DROP TABLE

```
drop table statement ::= DROP TABLE table_name
```

The DROP TABLE statement removes a file slot and any data for that slot from an Omnis datafile.

The DROP TABLE statement does not exist in the 1989 ANSI standard.

CREATE INDEX

```
create_index_statement ::=
   CREATE [CASE SENSITIVE] [UNIQUE] INDEX index
   ON table ( index_column_comma_list )

index_column ::=
   column_reference [ ASC ]
```

The CREATE INDEX statement lets you create an index on an Omnis database column. You can make the index UNIQUE, asserting that no two rows of the database have the same value for this combination of columns. You can also make the index CASE SENSITIVE, this will usually result in more efficient queries. The index column list contains columns from the table, and the table must already exist. You can also specify ASC on an individual column to sort it in ascending, as opposed to descending, order.

The CREATE INDEX statement does not exist in the 1989 ANSI standard.

DROP INDEX

```
drop_index_statement ::= DROP INDEX index
```

The DROP INDEX statement removes the named index, which must already exist.

The DROP INDEX statement does not exist in the 1989 ANSI standard.

SELECT

```
select_statement ::=
   SELECT [ ALL | DISTINCT ] { value_expression_comma_list | * }
   from_clause
   [ where_clause ]
   [ group_by_clause ]
   [order_by_clause ]
   [FOR UPDATE ]
```

The SELECT statement is the basic query statement in Omnis SQL. It largely matches the ANSI standard, one exception being the having clause, which in Omnis SQL is part of the group by clause instead of being a separate clause in the select statement. That is, in Omnis SQL you cannot have a HAVING clause separate from the GROUP BY clause. The FOR UPDATE clause initiates special locking for the records in the query. When you fetch a row from a cursor containing a SELECT statement with a FOR UPDATE clause, Omnis locks the row for update. One of three things can then happen:

- You update the record with an UPDATE … WHERE CURRENT OF cursor_name (see below), which on completion unlocks the row
- You fetch another row, which releases the lock on the previous row and locks the current one
- You terminate the transaction, which releases all locks

The order_by clause is separated out in ANSI SQL so that there is only one ordering for a query. Since Omnis SQL does not have any set operators, such as UNION, there is no need to separate out the ordering clause.

The ANSI 1989 standard has no for_update clause. This comes from embedded SQL, the syntax there is FOR UPDATE OF column_name_list.

Value Expression

```
value_expression ::=
   term
   | value_expression { + | - } term

term ::=
   factor
   | term { * | / } factor

factor ::=
   [ { + | - } ] primary

primary ::=
   literal
   | column_reference
   | function_reference
   | ( value_expression )
```

A value expression is a key element of SQL that lets you calculate a value using an arithmetic expression language. You build an expression out of literal numbers and strings, references to columns, or parenthesized, nested expressions. You can combine expressions with any of the four arithmetic operators. The grammar above expresses the precedence relationships between the operators: unary + and - take precedence over * and /, all of which take precedence over binary + and -.

Column and Table References

```
column_reference ::=
  [ table . ] column_name
  | [ alias . ] column_name
```

The column name corresponds to a field in a file class.

```
table ::=
  [ library_name . ] table_name
```

The table name corresponds to a file class or to a table alias in the same SELECT statement, and the library name corresponds to a library. The table must belong to the library.

Omnis SQL does not support the ANSI standard syntax alias.*, meaning all the columns from the table to which the alias refers. Also, if you use something other than a library name, or a name that Omnis cannot recognize as a library name, you will get a syntax error.

Function Reference

```
function_reference ::=
  scalar_function
  | aggregate_function
```

A function reference is either a scalar function or an aggregate function. Scalar functions operate on each row of data in the select; aggregate functions operate on groups of rows.

The ANSI SQL standard has no scalar functions.

```
scalar_function ::=
  scalar_function_name ( value_expression_comma_list )
```

There are a number of scalar functions, summarized below.

Function	Purpose	Parameters
ABS	absolute value of a number	number
ACOS	angle in radians, the cosine of which is a specified number	number
ASCII	ASCII character corresponding to an integer between 0 and 255, inclusive	integer
ASIN	angle in radians whose sine is the specified number	number
ATAN	the angle in radians whose tangent is the specified number	number
ATAN2	the angle in radians whose tangent is one number divided by another number	number 1, number 2
CHARINDEX	the starting character position of one string in a second string	index string, source string
CHR	ASCII character corresponding to an integer between 0 and 255, inclusive	integer
COS	cosine of a number	number
TODATE	converts a date string or number to a date value using a format string	date string/number, format string
DIM	increments a date string by some number of months	date string, months
DTCY	a string containing the year and century of a date string	date string
DTD	a string containing the day part of a date string or a number representing the day of the month, depending on context	date string
DTM	a string containing the month part of a date string or a number representing the month of the year, depending on context	date string
DTW	a string containing the day of the week part of a date string or a number representing the day of the week, depending on context	date string
DTY	a string containing the year part of a date string or a number representing the year, depending on context	date string
EXP	exponential value of a number	number
INITCAP	transforms string by capitalizing the initial letter of each word in the string and lowercasing every other letter	string
LENGTH	number of characters in a string	string
LOG	natural logarithm of a number	number
LOG10	base 10 logarithm of number	number
LOWER	transforms string by lower-casing all letters	string
MOD	modulus of a number given another number	number, modulo number
POWER	the value of a number raised to the power of another number	number, power
ROUND	rounds a number to an integer number of significant digits	number, significant digits
SIN	sine of a number	number
SQRT	square root of a number	number
STRING	concatenates some number of strings into a string	string[, string, ...]
SUBSTRING	extracts part of a string starting at a given index and moving a certain number of characters	string, start index, length

Function	Purpose	Parameters
TAN	tangent of a number	number
UPPER	transforms a string by upper-casing all letters	string

```
aggregate function ::=
   COUNT(*)
   | aggregate function name ( DISTINCT column reference )
   | aggregate function name ( [ ALL ] value expression

aggregate_function_name ::=
   AVG | MAX | MIN | SUM | COUNT
```

There are some departures from the ANSI standard for DISTINCT aggregates: you can use only one such function in a given SQL statement, and you cannot use aggregate functions in expressions in a GROUP BY clause or WHERE clause.

FROM Clause

```
from_clause ::=
   FROM table_reference_comma_list

table_reference ::=
   table_name [ AS ] [ alias ]
```

The FROM clause lets you specify the table to input into the SQL statement. Multiple tables in the list indicate a join, and the WHERE clause specifies the join condition. Each table reference can have an optional alias that lets you refer to the table in other parts of the SQL statement by the alias. You can use this to abbreviate references to the table in the other clauses.

The ANSI standard does not have the optional AS keyword.

WHERE Clause

```
where_clause ::=
   WHERE search_condition

search_condition ::=
   boolean_term | search_condition OR boolean_term

boolean_term ::=
   boolean_factor | boolean_term AND boolean_factor

boolean_factor ::=
   [ NOT ] boolean_primary

boolean_primary ::=
   predicate | ( search_condition )
```

The WHERE clause lets you select a subset of the input rows using a logical predicate. The above grammar defines the precedence of the logical operators AND, OR, and NOT.

```
predicate ::=
     comparison_predicate
   | between_predicate
   | in_predicate
   | like_predicate
   | relation_predicate
   | null_predicate
```

The ANSI standard has, in addition to the above predicates, the quantified and exists predicates (nested selects), which Omnis does not support. The relation_predicate is an Omnis extension to the standard that lets you use Omnis connections; see below.

```
comparison_predicate ::=
   value_expression comparison_operator value_expression

comparison_operator ::=
   < | > | = | <> | != | >= | <= | *= | =*
```

The standard comparison predicate involves one of the relational operators (greater than, less than, and so on).

ANSI SQL also allows you to use a nested select statement in place of the right-hand

value_expression; Omnis SQL does not support that. Omnis adds the !=, *=, and =* operators (not equal, left outer join, and right outer join, respectively) to the ANSI standard operators.

An outer join is a join that includes all the rows in the tables regardless of the matching of the rows. The *= operator includes all rows from the table on the left that satisfy the rest of the WHERE clause. The =* operator includes all rows from the table on the right that satisfy the WHERE clause. Rows from the other table (right and left, respectively, contribute values if there is a match and NULLs if not. This syntax is similar to the SYBASE outer join syntax.

```
between_predicate ::=
   value_expression [ NOT ] BETWEEN value_expression AND value_expression

in_predicate ::=
   value_expression [ NOT ] IN ( literal_comma_list )
```

The ANSI standard lets you use a subquery (a nested select) as well as a literal list; Omnis does not.

```
like_predicate ::=
   column_reference [ NOT ] LIKE literal
```

The ANSI standard adds an ESCAPE clause to the like_predicate to let you specify an escape character so you can match a % or _; Omnis does not implement this.

```
null_predicate ::=
   column_reference IS [ NOT ] NULL

relation_predicate ::=
   { CHILD | PARENT } OF table
```

The relation_predicate lets you test the current row as being either a child or a parent of rows in the specified table.

GROUP BY Clause

```
group_by_clause ::=
   GROUP BY column_reference_comma_list [ HAVING search_condition ]
```

The group_by_clause lets you group the input rows into groups according to a set of columns. The HAVING clause lets you select the groups, as opposed to the WHERE clause, which selects the rows going into the groups.

ANSI SQL has no ordering dependency between GROUP BY and HAVING, and you can have a HAVING clause without an accompanying GROUP BY. Omnis does not allow this.

Omnis SQL does not support the use of functions in a GROUP BY clause.

ORDER BY Clause

```
order_by_clause ::=
   ORDER BY order_column_comma_list

order_column ::=
   column_reference [ ASC | DESC ]
```

The order_by_clause lets you sort the output rows of the SQL statement using columns from the input tables.

The ANSI standard lets you sort by value_expressions in the select list by specifying the number of the expression; Omnis does not.

INSERT

```
insert statement ::=
   INSERT INTO table [ ( column_reference_comma_list ) ] { VALUES ( insert_value_comma_list ) | select_stat
```

The INSERT statement inserts rows into an Omnis table. The first list of columns names the columns you are creating; this exists to let you reorder the list to match your list of values or select statement.

There are two alternative ways to supply values to the INSERT statement. You can supply actual values through a VALUES clause that contains a list of values, or you can give a SELECT statement that creates a table of data matching the insert list. See the SELECT statement section above for details on SELECT.

```
insert_value ::=
   literal | NULL
```

An insert value is a literal value or the NULL value specified by the string ìNULLî.

UPDATE

```
update_statement_searched ::=
   UPDATE table SET assignment_comma_list [ where_clause ]

assignment ::=
   column_reference = { value_expression | NULL }
```

The searched update statement updates all rows that satisfy the predicate in the WHERE clause by assigning the indicated value or NULL to the column.

Omnis SQL will let you preface the column name in the assignment with the library and table names, which extends the ANSI standard. There is no need to specify the additional names, but you can do so for clarity if you wish. Specifying a table other than the table in the UPDATE table clause, generates an error.

```
update_statement_positioned ::=
   UPDATE table SET assignment_comma_list WHERE CURRENT OF cursor
```

The positioned update statement updates the current row, the row to which the current cursor points. See the description of the Declare cursor command in the Omnis Help. The WHERE CURRENT OF cursor clause works with the SELECT … FOR UPDATE statement to update rows locked for update.

DELETE

```
delete_statement_searched ::=
   DELETE FROM table [ where_clause ]
```

The DELETE statement deletes rows from the Omnis database based on the predicate in the WHERE clause. Omnis deletes all rows that satisfy the predicate.

JDBC

Note that support for JDBC has been removed in Studio 10 and above, but the supporting files can be obtained by contacting Omnis Support.

This section contains the information you need to access a database using the JDBC object DAM and JDBC drivers (plus their associated middleware where applicable), including server-specific programming, data type mapping and troubleshooting. For general information about logging on and managing your database using the SQL Browser, refer to the earlier parts of this manual.

Minimum Requirements

To use the JDBCSESS object the client machine must have the Java Runtime Environment v1.4 or higher installed. In addition, the JDBCSESS object will only support JDBC 2.x certified drivers. The JDBC DAM utilises the Omnis Java Engine (OJE), so it is important that the requirements for the OJE are also met. In particular a JVM_PATH environment variable must be set to the path of the JVM library in order for the OJE to start the Java Virtual Machine.

Properties and Methods

In addition to the "base" properties and methods documented in the *SQL Programming* chapter, the ODBC DAM provides the following additional features.

Session Properties

Property	Description
$dbmsname	Once a session has been established this is the type of database that the object is connected to. This defau (Read only)
$dbmsversion	Once a session has been established this is the version of the database software that the object is connecte a $logoff. (Read only)
$drivername	Prior to a session being established this should be set to the name of the JDBC driver that the object wish establish a connection. This can also be set using the $setdriver() method.
$driverversion	Once a session has been established this is the version of the JDBC driver that the object is connected to. $logoff. (Read only)
$logontimeout	The timeout in seconds for a $logon call. The default is 15 seconds. A value of 0 represents no timeout.

Session Methods

Property	Description
$setdriver()	$setdriver('drivername') sets the JDBC driver that the session object should use to establish a connection. Th assigning a name to $drivername.

Connecting to your Database

```
Do SessObj.$setdriver('sun.jdbc.odbc.JdbcOdbcDriver')
```

Failure to perform this step will cause the $logon() to fail. In order for the specified JDBC driver to be successfully loaded, it must exist in the system CLASSPATH environment variable.

To log on to the database using the SessObj.$logon() method, the hostname must contain the database URL required by the specified driver. The user name and password should contain the values required by the database.

Transactions

Generally, using manual transaction mode results in increased performance because the session object does not force a commit after each statement.

If you do not have a result set pending, JDBC session objects will commit each statement if the transaction mode is automatic. If the transaction mode is server, the session may be committed depending on the behavior of the JDBC driver.

Dates

The session property $defaultdate allows default values to be added to date values mapped to the server where the Omnis date value does not contain complete information, e.g. a Short time mapped to a server date time.

Multiple cursors

To allow multiple select cursors when connecting to Microsoft SQLServer, the statement issuing the SELECT must have the $usecursor property set to kTrue before the statement is executed. If a statement is issued when $usecursor is kFalse and this statement returns a result set, this will prevent all other statements in the same session from returning data. The blocking results must be completely processed or cleared before another result set can be generated. If a commit or rollback is performed on the session, all the session's statement cursors will be closed and all pending results will be lost.

JDBC Data Type Mapping

The following table describes the data type mapping for Omnis to JDBC connections. The Omnis to JDBC mapping will attempt to pick the best match based on the types the driver supports in the order listed. For example, if the driver supports VARCHAR and CHAR data up to a maximum column size of 255, but LONGVARCHAR data up to 2 gig, an Omnis Character(1000) will map to whatever the associated server native type is for LONGVARCHAR, e.g. TEXT.

Omnis to JDBC

Omnis Data Type	JDBC Data Type
CHARACTER	
Character(n) National(n)	VARCHAR(n) CHAR(n) LONGVARCHAR(n) CLOB
DATE/TIME	
Short date (all subtypes)	DATE TIMESTAMP
Short time	TIME TIMESTAMP
Date time (#FDT)	TIMESTAMP
NUMBER	
Short integer (0 to 255)	SMALLINT
Sequence Integer 32 bit	INTEGER NUMERIC(10,0) DECIMAL(10,0) < br>F
Integer 64 bit	BIGINT
Short number 0-2dp Number floating dp, 0..14 dp	FLOAT DOUBLE
OTHER	
Boolean	BIT SMALLINT NUMERIC(1,0) DECIMAL(1,0) CHA FLOAT
Picture, Binary, List, Row, Object, Item reference	VARBINARY(blobsize) BINARY(blobsize) LONGVARBINARY(blobsize) BLOB(blobsize) Wh SessObj.$blobsize

JDBC to Omnis

JDBC Data Type -	Omnis Data Type
CHARACTER	
CHAR(n) VARCHAR(n) LONGVARCHAR(n) CLOB(n)	Character(n)
DATE/TIME	
DATE	Short date 1980
TIME	Short time
TIMESTAMP	Date time (#FDT)
NUMBER	
SMALLINT INTEGER	Integer 32 bit
BIGINT	Integer 64 bit
DECIMAL(p,s) NUMERIC(p,s) REAL FLOAT DOUBLE	Number floating dp
OTHER	
BIT	Boolean
BINARY VARBINARY LONGVARBINARY BLOB	Binary

Chapter 10—Report Programming

You can create many different types of report using a template or wizard, each with very different layouts and data handling capabilities. With reports you can print out all or a subset of your data and collect up data from different sources and print it on a single report. Each type of report in your application is defined as a **Report class.** This chapter describes Report classes and Report sections and their properties. Reports can contain data fields, pictures, text, and graphics. You can also place graphs on your reports, or base a report on the data contained in an Omnis list.

For *web and mobile apps,* you can print a report class to a PDF and display it in the end user's web or mobile browser using the 'showpdf' and 'assignpdf' client commands, or you can allow the end user to download the report file to their device. You can use the PDF Device to print a report to a PDF file: see PDF Printing. This chapter describes how you can create a report class, which you will need to be able to print a report in web and mobile apps, but some sections only apply to creating and printing reports on desktop apps.

For *desktop apps,* you can print reports to a number of destinations, including a Page preview that is displayed on the end user's screen, the current Printer, a File, a Port, or the Clipboard; this and all aspects of printing reports in desktop apps are covered in this chapter.

Report Fields and Sections

You use report fields (report entry fields) and sections to build all types of report. You can use standard *data fields* that can contain data from your server or Omnis database. You can use *picture fields* to display picture data. You place s*ections,* or horizontal dividers, across your report class that structure and position the data in the printed output. You can create subtotals, totals, header and footer sections for most types of reports. By setting the appropriate properties in a report class you can print labels as well. Furthermore, you can add methods to a report class and the fields and section markers on the report.

Report Object Limit

You cannot place an unlimited number of objects on a Report class. The object limit is **8191** for a Report class, although in practice the limit is likely to be less due to platform limitations (the limit was 3000 in versions prior to Studio 11).

You are advised to split large reports (containing a large number of report fields) into a number of sub-reports, and print them to either the Printer or PDF using the *Begin* and *End print job* commands.

Report Field Types

The following fields or components are available in the Component Store for report classes:

Group	Component
Entry Fields	Entry Field
Graphs	Graph2
Labels	Label String Label for multi language support Text
Lists	Data Grid Wrapping List
Media	HTML HTML Link Objects incl Text, Icon, Picture HTML Raw Text JPEG Picture Rtf Viewer
Other	Calendar Page Count
Sections	Report Positioning Bar
Shapes	3d Rectangle (see Report Shapes) Line Oval Rectangle Rounded Rectangle

Report Entry Fields

Entry fields display the data from your database. The $dataname for an entry field specifies the instance variable or database column name to be displayed or printed in the report.

The $tooltip property allows you to add a tooltip to a report entry field, to be displayed in a Page Preview or PDF report.

The $linkaddress property can be a URL link address used by the Preview and PDF report destinations to provide a hyperlink; this provides similar functionality to the $address property of HTML Link objects.

The **omnisPreviewURLPrefix** item in the 'defaults' section of config.json allows you to set the report preview URL prefix for the $linkaddress property for report class Entry fields. The item defaults to 'omnis:' if empty.

Wrapping Lists and Data Grids

The **Wrapping List** and **Data Grid** report fields support the use of styled text, provided their $::styledtext property is enabled. You can insert styling characters or "text escapes" into text objects and other fields in Omnis reports using the style() function.

HTML Link Objects

Hyperlinks are supported in **PDF** and **Page Preview** report destinations. To add a hyperlink to a report, you can use one of the **HTML Link** objects, found under the **Media** group in the Component Store, including:

- **HTML Icon (Link)**
  adds a link behind an icon, specified in the $::iconid property
- **HTML Picture (Link)**
  adds a link behind a picture; the image can be from a variable specified in $dataname or from a calculation specified in $text.
- **HTML Data Text (Link)**
  adds a link behind some data bound text; the text can be from a variable specified in $dataname or from a calculation specified in $text.
- **HTML Text (Link)**
  adds a link behind some static text specified in the $::text property.

To add the hyperlink to the HTML Link Objects you need to set the **$address** property to the target of the link, which can be a URL or mailto, for example:

```
https://omnis.net
mailto:bob.smith@omnis.net
```

The $tooltip property can contain a tooltip used for the link specified in the $address property. It can contain expressions including square bracket notation.

For **Page Preview** reports only (ignored for PDF reports), you can create a special custom HTML link using the syntax:

```
omnis:p1,p2,p3,p4
```

where the data after omnis: is a comma separated list of parameter values, which can be integer or character, and must not contain " (double quotes).

In the latter case, when the user clicks the omnis: link, Omnis looks for a method called $previewurlclicked. Firstly, if the report has been sent to a window field, Omnis looks for the method in the window instance containing the screen report field, otherwise if the first test fails, Omnis looks for the method in the task that printed the report. If Omnis finds the method, it calls $previewurlclicked passing it the following parameters:

- An item reference to the report instance.
- The ident of the report object.
- A row created by adding a column for each comma delimited item in the data after omnis: in the link.

You can create a method called $previewurlclicked in either the window or task, as above, and react to the parameters passed.

Aspect Ratio for Icons and Images

For the image related report objects, such as HTML Icon (Link) and JPEG controls, you may need to set the $keepaspectratio property to kTrue for the icon or image to draw with the correct aspect ratio. If this property is kFalse (the default), the icon or image may not draw correctly. In addition, some of the report image controls have the $noscale property; if set to True, the control will not scale the image.

JPEG Report Control

The **JPEG** control allows you to place a JPEG image on a report. The image can be from a variable specified in $dataname or from a calculation specified in $text. Alternatively, you can specify the path to the JPEG image in **$path.**

Picture Report Control

The **Picture** control allows you to place an image on a report. The image can be from a variable specified in $dataname or from a calculation specified in $text.

Rtf Viewer Control

The **Rtf Viewer** control allows you to embed an RTF document into a report. The RTF file can be from a variable specified in $dataname or from a calculation specified in $text. Alternatively, you can specify the full pathname to the RTF file in the **$filename** property.

Report Labels

You can use the **Label** or **Text** report objects in your report classes to label report fields and so on. In addition, you can use the **String Label** object which allows you provide multi-language labels on your reports: for information about the use of String Labels and String tables, see the Localization chapter.

Report Shapes

You can add various graphical shapes to your reports using the objects in the **Shapes** group in the Component Store including **3d Rectangle, Line, Oval, Rectangle,** and **Rounded Rectangle.** You can control the appearance of these shapes under the Appearance tab in the Property Manager, includng the border, line styles, and colors, as appropriate.

Report Wizard

The SQL report wizard lets you create a report that contains fields that map directly to a SQL class in your library, which lets you print data on your SQL server database. Before you can use report wizards you must create the schema or query classes necessary for SQL (for legacy apps you can use an Omnis file class).

For SQL reports you need to select the Session to be associated with the new report. The SQL report wizard creates a report based on a schema or query class; each separate field on the new report maps to a schema column, which in turn maps to your server database

To create a new report using a wizard

- Select your library in the Studio Browser
- Click on the Class Wizard option, then click on the Report option
- Select the report **SQL Report wizard** and click on the Create button

When you finish in the Report Wizard the new report class is opened ready for you to modify or print. To modify your report, you need to edit its properties. You can also add new report objects from the Component Store, and you can add methods to the report class or the objects on the report.

(Note there is an “Omnis Report Wizard” which creates a report based on an Omnis file class, but this should only be used in legacy apps that use Omnis datafiles, and not for new applications.)

Report Tools

The toolbar at the top of the report editor lets you set the page size, preview the report on screen, and show or hide connections between the different sections of the report as shown down the left-hand side of the report editor. In addition, you set the sort levels in your report from this toolbar. Position your mouse over each tool to see what it does.

Some of the options in the report editor toolbar are available using the report context menu by Right-clicking on the report background.

The **Narrow Sections** option displays the section markers as narrow lines which shows you how the report will look when you print it. The **Show $ident** option displays the ident numbers for the fields and section markers in the report. The **Field List** option displays a list of fields and section markers in the report class. You can expand the tree in the Field list to show the fields within each report section.

The **Class Methods** option lets you add methods to the report class, and the Properties option shows the properties for the class. The Page Setup option or toolbar button opens the Page Setup dialog in which you can select the printer, and set the page size and orientation. This dialog will vary greatly across the different operating systems.

Zoom In/Out

The report class editor toolbar has **Zoom In** and **Zoom Out** buttons which control the DPI value used to convert report coordinates to and from pixels, and the DPI value used to create fonts used in the editor. “Zoom in” increases the DPI value, “Zoom out” decreases it. Note this is for design mode only, and you can zoom through a limited set of DPIs:

- 72 – objects displayed at standard resolution for macOS
- 96 – objects displayed at standard resolution for Windows (with default system scaling of 100%)

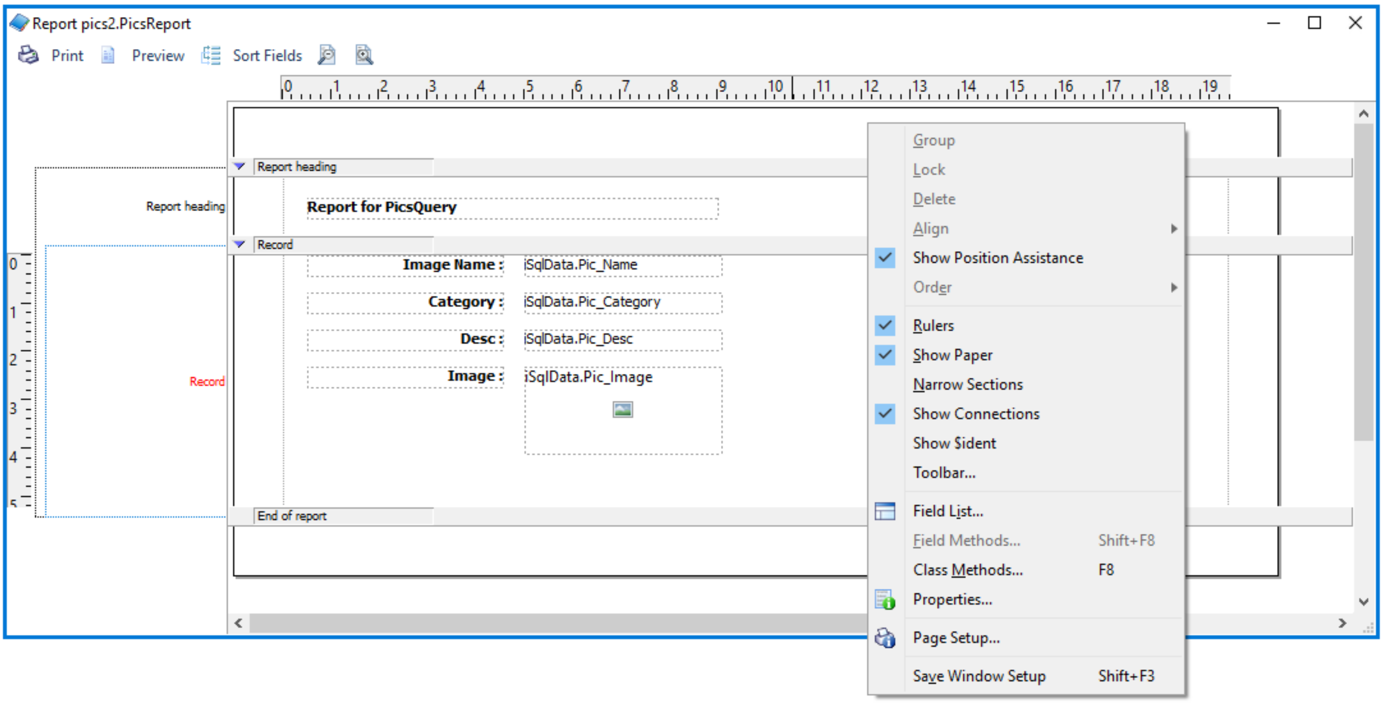


Figure 133:

- 144 – objects displayed at 2x resolution for macOS
- 192 – objects displayed at 2x resolution for Windows

In addition, if Windows is using a different scaling value, the editor inserts the system DPI into this list at the appropriate point.

These values correspond to the design coordinate system used in Omnis, so on HD displays 96 DPI maps to 192 physical pixels.

You can use Ctrl+ and Ctrl- (Cmnd+/Cmnd- on macOs) to zoom in and zoom out respectively. The current zoom level is saved with the window setup by the save window setup context menu item.

Note that the section bars and the text in the left panel do not increase in height when you zoom. Note also, that zoom does not affect the size of lines drawn in fields on the report - only the text, and in some cases images will scale.

The Modify Report field has a new runtime property, $dpi, that can be assigned to one of the values above.

External Components

If you create external components for reports then you will need to make some changes to draw text at the correct DPI. Typically, if the component just displays its name or dataname using the standard interface, you won't need to do anything, as the text DPI will be handled by the Omnis core. Where components that can be placed on reports draw custom text, there are some changes to make in the component:

- There is a new callback ECOgetFontDpi(HWND) that returns the current DPI to use to create fonts - this will return zero unless the component is on a report design window, in which case it will return one of the above values.
- There is a new class GDIfontCreator, that you construct with the HDC for drawing, and the return value from ECOgetFontDpi. This has a method createFont that you then use to create the font rather than calling GDIcreateFont. When you have finished with the font, call GDIdeleteObject as usual. You cannot cache the HFONT generated by createFont in your component.
- If you require font or text metrics, use the HDC versions of GDIfontHeight, GDIfontPart, GDItextWidth etc, with a font created using GDIfontCreator.
- In addition, for more advance use there are classes GDIhdcFontCacheHelper which removes all fonts cached by the Omnis font cache for a particular HDC at the end of the block and GDIoverrideHDCDPI which means that all fonts created for a specific HDC are created at a specified DPI while GDIoverrideHDCDPI is in scope. You need to use GDIoverrideHDCDPI if you are drawing styled text, as styled text drawing may create new fonts. In addition, when drawing styled text, you need to set mFontHdc in the GDIdrawTextStruct, in order for fonts to be created at the correct DPI.
- You can also call GDIcreateDcFont with a DPI parameter to manage fonts yourself.

Report Sections

Sections are horizontal markers or dividers across the report class that structure and position the data when your report is printed. To create a complex report with headers, footers, subtotals, and totals, such as an invoice or catalog listing, you have to place the appropriate sections in your report class in the right order. When you enable the various sections in your report using the Property Manager, their position and order is handled for you automatically.

There are two sections that you must have in a report:

- the ***Record section*** indicates the start of the display of records or rows of data, and
- the ***End of report*** section indicates the end of the report.

These sections appear automatically in every new report class. The following section types are available:

Section type	Description
Report heading	defines the area at the start of the report, which prints only once; you can use this to
Page header	defines an area at the top of each page below the top margin, printed at the top of e
Subtotal heading	prints before each subtotals section; it would normally contain column headings for
Subtotal heading 1 to 9	each subtotal heading prints before its corresponding subtotal level
Record	defines the section containing the fields that print your data; the record section exp extend over several pages when printed
Positioning	divides a section into two or more subsections; you can control exactly where on the
Subtotals level 1 to 9	defines the fields that will print subtotals; you can have up to 9 levels of subtotaling
Totals	prints at the end of the report and defines the fields that you want to total
Page footer	defines the area at the bottom of each page; printed at the bottom of each new pag
End of report	defines the end of the report; must be present on every report

To enable a particular report section, you have to set the appropriate property to true under the **Sections** tab in the Property Manager. When you enable a particular report section, it is shown on your report in the correct position. You can click on individual sections to change their properties in the Property Manager.

Page Headers and Footers

To create a page header for your report, you must set the **pageheader** property of the report to true under the **Sections** tab in the Property Manager. The Page header section will appear on your report above the Record section, or any subtotal headings if you have any.

Similarly, to create a page footer for your report, you must set the **pagefooter** property to true under the **Sections** tab in the Property Manager. The Page footer section will appear on your report below the Record section, or any Subtotals and Totals sections if you have any.

Any fields or graphics you place in the header section, that is between the Page header section and the next section marker will print at the top of each page. Likewise, any fields or graphics you place in the footer section, that is, below the Page footer section marker, will print at the bottom of each page. Note that the connection between the different sections is shown in the left margin of the report editor: the current section is shown in red.

When printing to a non-paged device such as File or HTML, by default the footer section is not printed. The Report header and first Page header sections are printed at the beginning of the report. However it is possible to force the footer section to be printed by calling $printsection(kFooter) for the report instance. The default positioning for a footer for a non-paged device is to follow on from where the last section stopped printing.

To change the height of any section, including the record, header and footer sections, you can click on the section marker (the gray bar) and drag it into position. All the sections below the one you move will adjust automatically.

To show you more how the report will look when you print it, you can view the sections as narrow lines. To view sections as narrow lines, click on the Narrow sections button in the report editor toolbar, or Right-click on the report background and select the Narrow Sections option.

Printing Sections as Record Sections

The $printsection method has a new parameter to force the report section to print as a record section. The method $printsection(iSection[,bPosnIsRecord=kFalse]) prints a section: note bPosnIsRecord applies to positioning sections only. If bPosnIsRecord is kFalse (the default), this method prints a section based on the position of the previous section; otherwise, when this parameter is true the method prints the section as a record.

Borders and shading

You can set the border and fill properties for each section in a report. The report class and section objects have the following new properties: $effect, $forecolor, $backcolor, $bordercolor, $backpattern, $linestyle. In addition, sections have $topmargin, $leftmargin, $bottommargin and $rightmargin properties to allow the sections border and fill to be inset from the sections boundary.

Positioning sections cannot have their own border and fill properties. The border and fill of the main section extends through all its position sections. The border and fill properties of each section are visible in design mode.

Watermarks

You can add a watermark effect (using a background image) to your reports by assigning a picture to the $backpicture property of the report. The alignment can be one of the following within the report margin: TopLeft, TopCenter, TopRight, CenterLeft, Center, CenterRight, BottomLeft, BottomCenter, BottomRight, Stretch (stretch the image to fit the report margin), or Tiles (tile the image within the report margins).

The horizontal and vertical DPI (dots per inch) specify the resolution at which the image is to be printed.

When using notation, $backpicture refers to the binary representation of the image and the formatting properties. To change the image or formatting properties you can use $backpicture.$picture or $backpicture.$align. The full set of notation is as follows:

Property	Description
$backpicture.$picture	the actual image (24bit color shared).
$backpicture.$picturealign	alignment of the image within the report margins. This can be one of the kPALxxx co
$backpicture.$horzdpi	the horizontal dpi (defaults to 150), disabled for kAlignMargins and kAlignPrintable
$backpicture.$vertdpi	the vertical dpi (defaults to 150) , disabled for kAlignMargins and kAlignPrintable.
$backpicture.$horzoffset	additional horizontal offset in cms or inches from the alignment. Disabled for alignme kAlignPrintable
$backpicture.$vertoffset	additional vertical offset in cms or inches from the alignment. Disabled for alignments

Icon IDs for background pictures

You can use the $backiconid property to assign an image to the background of a report using an image or icon ID, rather than using $backpicture. The property must refer to an icon ID in an icon set, or an alpha page in #ICONS in your library or an icon data file. If you specify a value for $backiconid it takes precedence over $backpicture.

Section Positioning

When you print a report, each section follows the previous section by default, and is positioned down the page according to the height of the previous section set in the report class. However, for some types of section, you can control where a section prints and whether or not a new page is forced using the **pagemode** and **startmode** properties of the section. You can use a special type of section marker called a *position* section to print part of your report literally anywhere on the page. To do all these things you have to modify the properties of the appropriate section marker.

To view the properties of a section, open your report class in design mode and click on the appropriate section to view its properties in the Property Manager.

Page Mode

You can control whether or not *Record, Subtotals, Totals*, and *Subtotal heading* sections start a new page when they are encountered by setting their **pagemode** property. You can select one of the following options for this property.

- **Nopage**
  does not force a new page; uses the pagination in the report class (the default)
- **Newpage**
  always starts a new page before this section
- **Testspace**
  starts a new page before starting this section if there is not the specified amount of space available on the current page

If you select the Testspace option, the **pagespacing** property is enabled in the Property Manager in which you can enter the amount of space required for the section. If this amount of space is not available on the page, a new page is started. The figure you enter in pagespacing is shown on the section marker.

Omnis works with units that are 1/72 of an inch; therefore, it may round exact numbers in centimeters or inches to the next real unit. For example, 1cm becomes 0.99cm.

Start Mode

All sections except for *Page footer* and *End of report* let you specify the **startmode**, which tells Omnis where to start the section. You can choose one of the following options.

- **Follow** previous section
  starts the section on the line following the end of the previous section (the default)
- **Fromtop** of previous section
  starts the section n inches/cms from the *top* of the previous section
- **Fromend** of previous section
  starts the section n inches/cms from the *end* of the previous section
- **Fromtopmarg**
  starts the section n inches/cms from the *top margin* of the report
- **Frombottommarg**
  starts the section n inches/cms from the *bottom margin* of the report

When you choose one of the start modes the **startspacing** property is enabled in the Property Manager, which lets you enter a measurement for the startmode. The startmode and spacing is shown on the section marker.

Omnis ignores previous section settings if the previous section was a *Page header* section or a *Positioning* section within a *Page header* section. The spacing comes before the page start test that examines the amount of space left on a page. Omnis ignores top and bottom margin settings for reports that are not paged.

Note that when you set up a report to print labels, you can use the Fromtop or Frombottom options to set the spacing between your labels.

You can enter a negative value for the start spacing of a positioning section, for example Start –1.000cms from end of previous section. This allows you to align fields with the bottom of an extending field.

Record Spacing

The default spacing between records or rows of data on your printed report is determined by the height of the *Record* section in your report class. However you can override this spacing by setting the **userecspacing** property for the Record section. This property forces the report to use the vertical spacing set in the **recordspacing** property of the report class.

Position Sections

A *Position* section is a special type of report section that you can print literally anywhere in the report. For example, using a positioning section, you could print a footnote or a logo at the bottom of a letter, regardless of the content or amount of data in the main report letter.

A positioning section placed over the second line of a two-line extending field with the Follow previous section property prevents the second line from printing as a blank. You can also follow extending fields by a positioning section with Follow previous section to prevent them from writing over any fields below. A positioning section within a subtotal section lets you trigger a print position change by changing a sort field value.

Section Print Height

Report sections have the property $printheight which lets you specify the printing height of a section more accurately than using positioning sections. The printing height of a section is displayed in the section bar. Setting the sections print height does not affect the positioning of sections in the report class or on the design window. If a section area is larger than its print height, the area below the print height is filled with gray. Any report objects that fall entirely in the gray sections ($top of object is greater than $printheight of section) are evaluated but not printed.

You can create label reports using a combination of the $printheight property and the positioning mode kFitOnPage. If a section is too high to print on the current page, it will be printed the next page. Some objects are normally moved to fit on a page at the time they are added to the report, e.g. a single line of text. This behavior is disabled for objects in sections that have kFitOnPage selected.

Standard Lines between sections can expect an overall vertical placement inaccuracy of +-1/72nd of an inch. Hair lines can expect an overall placement inaccuracy equal to that of the DPI of the printer, but not better than +-1/600th of an inch.

Section Calculations

Sections have the $printif property which can trigger the section to print based on a calculation. If a section has a calculation, it is skipped if the calculation evaluates to zero, otherwise the section is printed as normal. If a section has been given a calculation, the text "Print if calculation" will appear in the section bar.

If a main section is skipped, all positioning sections belonging to the main section are also skipped, regardless of their $printif property. Calling $printsection from a method will ignore the $printif calculation and always print the section.

Custom Sections

A custom section is effectively a positioning section with $printif calculation of '0'. This means the section will not print unless, $printsection() is called with a reference to the section. You can add a custom section (positioning bar) from the Component Store.

Custom sections can be placed inside any of the main sections, but can be printed before or after any of the other report sections, but not during. Changing the $printif calculation to anything other than '0', will change the section into a normal positioning section, and vice versa.

Hiding Sections in design mode

You can collapse or expand the sections in a report to simplify the visual appearance of the report in design mode. The collapse/expand icon in the report editor lets you collapse a section, leaving the section divider visible. The current collapse/expand state of a section will be saved in the report class.

When a 'main section' is collapsed, any positioning sections in that section are also hidden. Positioning sections can also be collapsed or expanded individually. Collapsing of sections is disabled when narrow sections are viewed.

Sorting and Subtotaling

To implement sorting and subtotaling for your report, you need to specify the fields on your report to be sorted, and create subtotals sections containing those fields. *Sort fields* define how Omnis subtotals the records or rows of data when printing a report. With no sort fields, Omnis displays records in the order they are listed on your server, or if the data is from an Omnis data file, in the order the data was inserted. When you add sort fields to your report, the report will print subtotals when the values change in the sort fields.

To specify sort fields for your report, click on the Sort Fields button in the report editor toolbar; the Sort fields window opens. In the Sort fields window, you can specify up to nine sort fields by entering each field or variable name in the left-hand column. Note that you can use the Catalog to enter your field or variable names. These sort fields form a nested sequence of sorts on the records that trigger printing of up to nine nested subtotal sections.

The sort fields for a report class are stored in the $sorts group for the class. You can modify the contents of this group at runtime using the notation, to change the sort fields for the report, but if you want the changes to take effect this must be done in the $construct() method of the report before the report instance is created. If you have more than one instance of a report class, each instance will have the sort fields specified in the class, but you can modify the $sorts group for a particular instance if you wish to change its sort fields.

To subtotal a field, place a copy of the field in the Subtotal section and select the appropriate **totalmode** for the field. This is independent of the sort fields, which trigger the printing of the Subtotal sections.

When you enter a field or variable name in the list of sort fields the sorting options are enabled for that field. You can enable any of these options by clicking on the cell and selecting true.

Each sort field has the following options.

- **Descending** sort
  sorts the field in descending (Z to A and 9 to 0) instead of the default ascending order

- **Upper case** conversion
  converts field values to upper case before sorting, so the use of mixed case in your database does not affect subtotaling or sorting
- **Subtotals** when field changes
  tells Omnis to print subtotals using the corresponding subtotal section (1 to 9) when the value of the field changes; that is, if sort field 4 changes, subtotal level 4 will print
- **New page** when field changes
  starts a new page as well as printing a subtotal when the value of the field changes

When you enable the Subtotals or New page options for a sort field, you can specify the number of characters that must change before a subtotal is triggered or new page is printed.

Subtotal Sections

You can specify a *Subtotal heading* section in your report. It prints before the first Record section and successive Record sections following each Subtotals section. The subtotal heading can print column names and anything else you want to apply to each subtotaled Record section.

The *Subtotals* section prints whenever the Record section breaks on the corresponding sort field, with the subtotal printing before the record with the changed value. Since there are up to nine sort fields, you can have up to nine *Subtotal heading* and *Subtotal levels* numbered 1 through 9 corresponding to the sort fields specified in the report. The higher numbered sort fields are nested within the lower ones and hence change more often. That is, sort field 5 changes within sort field 4 which changes within sort field 3, and so on. Correspondingly, the Subtotal heading and Subtotals sections with higher numbers print more often as the sort fields change.

When you have multiple subtotals which print consecutively, the corresponding heading sections print one after another, starting with the one for the last subtotal. Subtotals and totals can be aggregations of several kinds, including sums, averages, counts, minimums, or maximums, depending on the field's **$totalmode** property. Omnis maintains the total for each subtotal printing, then resets the subtotal to zero for the next section.

The *Totals* section prints at the end of the report. As for subtotals, you place the fields to aggregate in this section, and Omnis accumulates the aggregate values across the entire report. You can set the **totalmode** property for a field in the totals section.

Calculated Fields

In addition to totaled fields in subtotal or total sections, you can place calculated fields in report sections which take their value from the result of some calculation. In this case the $dataname property of the field is left blank, the $calculated property is enabled and the calculation is placed in the $text property. For example, this could be the concatenation of a Firstname and Surname variable to display a person's full name. Note that such calculated fields cannot be totaled in subtotal or total sections.

PDF Accessibility

Support for PDF/UA in PDF reports is enabled by allowing you to set the order of objects in a report class in design mode and tag non-text objects such as images. The report object ordering and tags can then be read by an accessibility PDF reader.

The Omnis report engine builds up a page of objects during the record printing process adjusting object positions using sections, $print and the natural order of records during printing. When the print job is complete, Omnis sends the final objects, page by page, to an output device ordered *left to right*, and from *top to bottom*. When printing to PDF, page by page, the objects are sent to the PDF device to be turned into a PDF file using PDF Kit. Until now, the order of objects in the final Omnis print job would determine the order of objects in a PDF file (left to right, top to bottom).

For many reports, this order of objects whilst visually acceptable may not be acceptable if read by an accessibility PDF reader. For example, you may have text that is grouped and makes sense to be read together as a block. With the addition of support for PDF/UA, you can now change the reading order of the objects in a report class and the output PDF if required.

PDF/UA Support

PDF/UA (PDF Universal Accessibility), formally known as ISO 14289-1, specifies that objects in a PDF file should be capable of being read in a specific order, and that visual objects that are non-text based (such as images) should be tagged so they can be read by a PDF reader.

To enable PDF/UA support and have objects tagged and read correctly, an Omnis report printed using the Omnis PDF output device may require developer modifications to control the order output and apply text tags to some objects.

The Omnis PDF device has a subset setting **kDevOmnisSubsetPDFUA** for the Omnis PDF Device.**$setpdfsubset**(*iPdfSubset*) function that when set, tags all objects correctly and sorts objects using the reading order specified in the report class before committing the objects to the PDF file.

PDF/UA support requires a minimum PDF version of 1.7. You can use the Omnis PDF Device.**$setpdfversion**(*iPdfVersion*) function to set the PDF version. The following code can be used to set the PDF subset and version:

```
Do Omnis PDF Device.$setpdfsubset(kDevOmnisSubsetPDFUA)
Do Omnis PDF Device.$setpdfversion(kDevOmnisPDFVersion17)
```

Object Reading Order

To specify the reading order of objects in a PDF report, Omnis design mode allows you to assign a *'reading order'* to an object or set of objects. This reading order is used by the PDF device to sort and order fields in the final PDF file, that is, a PDF reader will read the fields in the specified order.

To set the reading order, report objects have three properties available under the **Accessibility** tab in the Property Manager.

- **$readingordergrp**
  a number representing a field's main grouping, default is 0, meaning it has no group
- **$readingorderindex**
  a number representing a field's order within its reading group, default is 0
- **$alt**
  some non-text based objects (images) support this. The text tag assigned to the object in the final PDF file

All objects are first sorted by **group,** and then by **index** within the group. If a group of objects runs top to bottom, the index within the group does not need to be set, as this is the default order Omnis will use.

To help with setting groups and group indexes in report design mode you can show the group and index (like viewing an object's name or $ident). The **Show Reading Order** option in the report design context menu allows you to see the reading order of objects in the report. By default, there is no reading order, so the order in the report design window is shown in a red box and the value is zero.

To set an object's reading order, select an object or group of objects in design mode and use the Property Manager to set **$readingordergrp** and **$readingorderindex.** Once the order is set, the reading group is shown in a blue box, and if an index in the group is set, this is also shown, e.g. 1:1 is the first object in group 1. The following image shows a report after some fields have been given a group reading order.

If you intend to change the reading order of some objects, then all objects in the report will require the reading order to be set as the default order is 0 and this may conflict with objects with an order. If you only need to change a few objects, do so then assign all other objects a number larger than the order of the objects you want to control.

The $printontop property for report objects will be ignored when the object reading order is specified in your report class.

Previewing Object Order

Viewing the design mode reading order is useful, but due to $print and position sections which can move objects during the print stage, only a final output gives a true representation of the final PDF. To allow you to see the object reading order in the report output, the **Show Reading Order** option is also available in the context menu in the **Print Preview** window. Omnis will show all the assigned reading order stops, and the links between them. There is a small toolbar in the top left corner of the Preview window allowing you to step through the report object order, as it would be read by a PDF reader.

Alternatively, you can use the **Plus** (+) or **Minus** (-) keys to step through the objects in the Preview report. As you navigate through the order, each object is highlighted showing its reading order stop, but you can press Backspace or the left-most toolbar button to show all the reading order stops and links.

The following image shows a report with no object order set, and in this case, Omnis will navigate left to right, top to bottom. The reading order is shown in red, and the link lines show the order of objects a PDF reader will take.

The next image shows a report where the order has been applied. You can see and follow the navigation through the report, and as you press the Plus key the highlight moves through the report objects showing the reading order. The order shows the current object and a link from the previous object and to the next object.

Using this navigation helps you visualize the object reading order and allows you to change the ordering back in the report class to create a final PDF file containing the object ordering you require for the best accessibility.

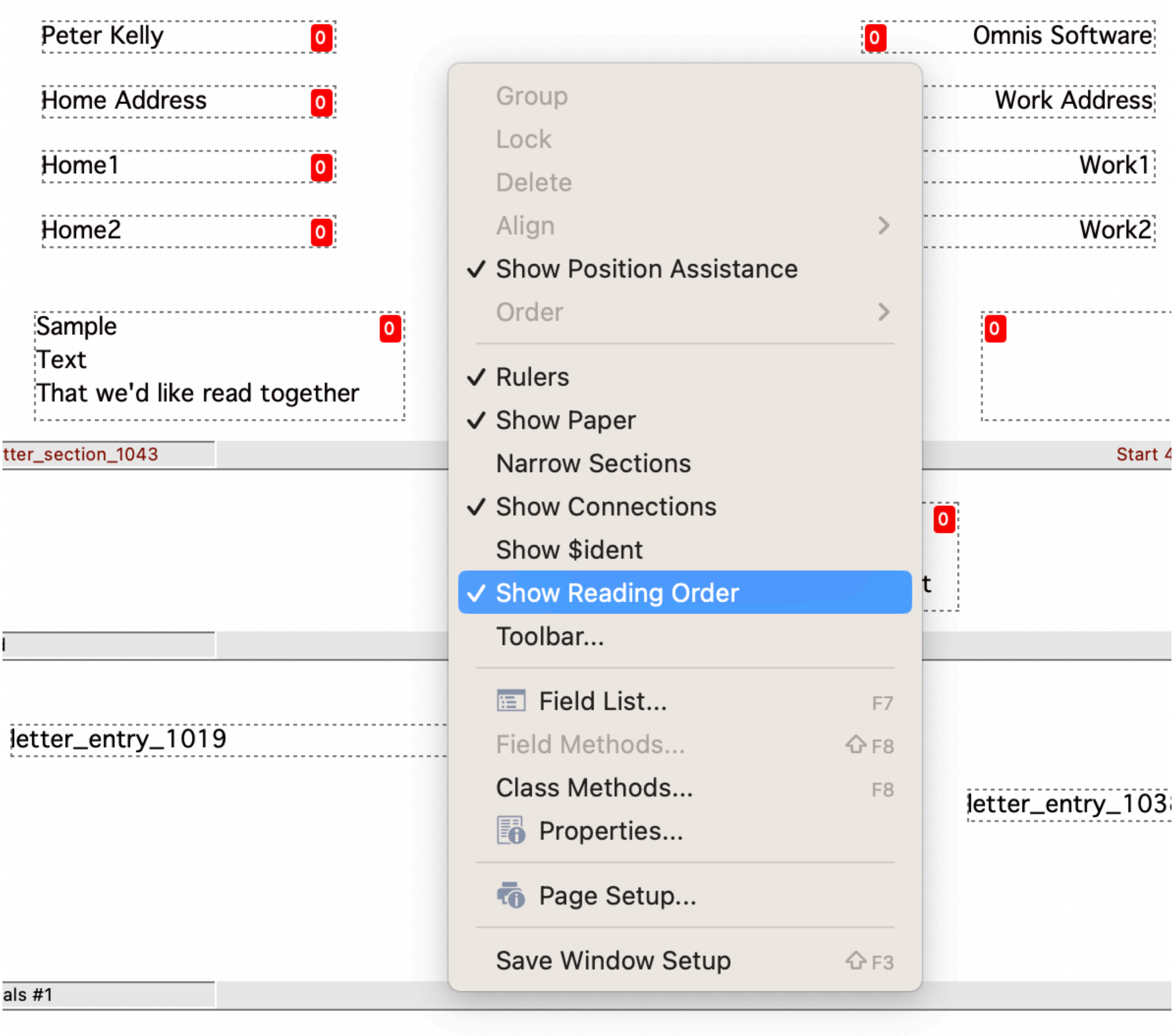
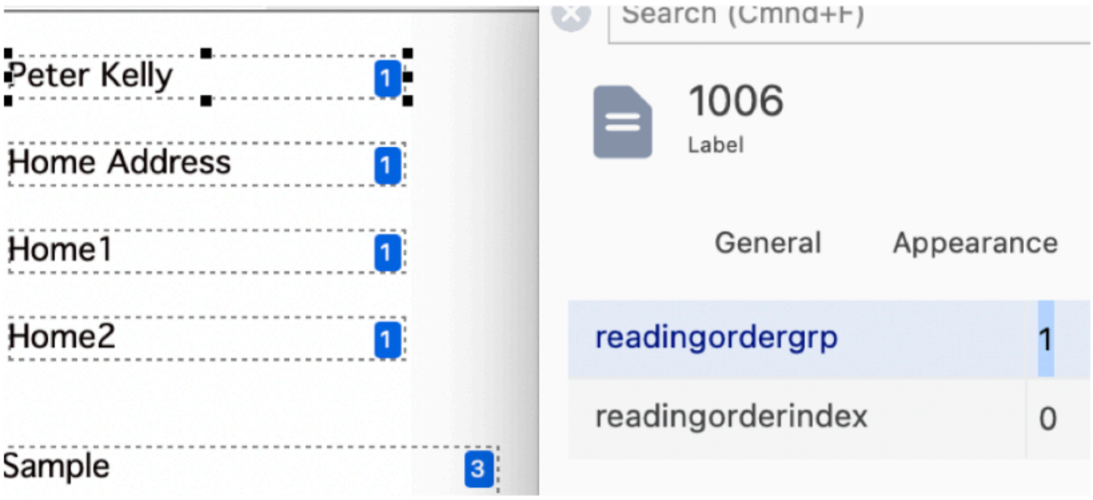


Figure 134:



readingordergrp	1
readingorderindex	0

Figure 135:

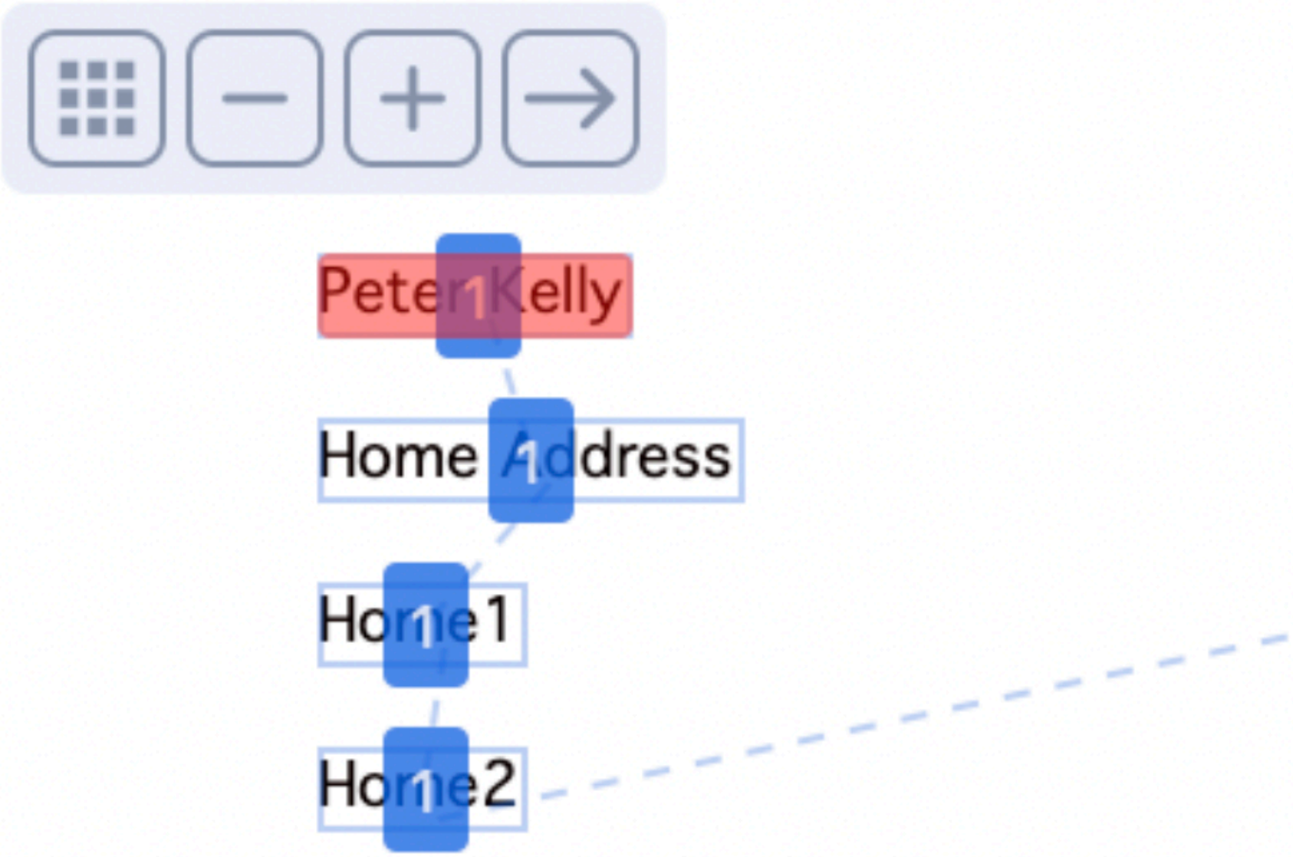


Figure 136:

Custom URLs

You can embed a Custom URL (link) in a PDF report that could, for example, launch Omnis, and call a method inside a specified library, which could perform an action. This is enabled via support for custom URL schemes. Omnis can support one or more custom URL schemes.

On macOS, the URL schemes must be defined in info.plist. The role for the URLs added to info.plist must be 'Editor', otherwise Omnis will not be invoked by the URL. If you add any URL schemes to info.plist you must re-codesign Omnis.

On Windows, the "customURLSchemes" item in the "windows" section of the Omnis configuration file (config.json) is a JSON array of strings. A scheme name can contain alphanumeric characters only and must start with an alpha character. By default, the customURLSchemes item contains the entry "studio%v". When reading the schemes from config.json, Omnis replaces %v in each string with the appropriate value for NNN, where NNN represents the major version of Omnis. For example, the default scheme name will be studio111 in Studio 11.1, or studio120 in Studio 12.0 and so on.

The "customURLUUID" item in the "windows" section of config.json is a UUID that identifies the Omnis server to clients using custom URLs. This value is reserved for internal use and should not be changed.

Creating a custom URL

The custom URL is made up of the scheme name plus any parameters you wish to pass into Omnis. The syntax for a custom URL is:

```
scheme://params/
```

In some cases, the optional trailing / is needed to make the client consider the URL syntax to be acceptable.

The *params* can be either query string parameters or JSON. Here is an example of custom URL protocol using query string parameters:

```
studio111://lib=test&test=10&value=another
```

Here is an example of a custom URL procotol using JSON parameters, before escaping them:

```
studio111://{"lib":"test","test":10,"value":"another"}
```

And after escaping them, for safety:

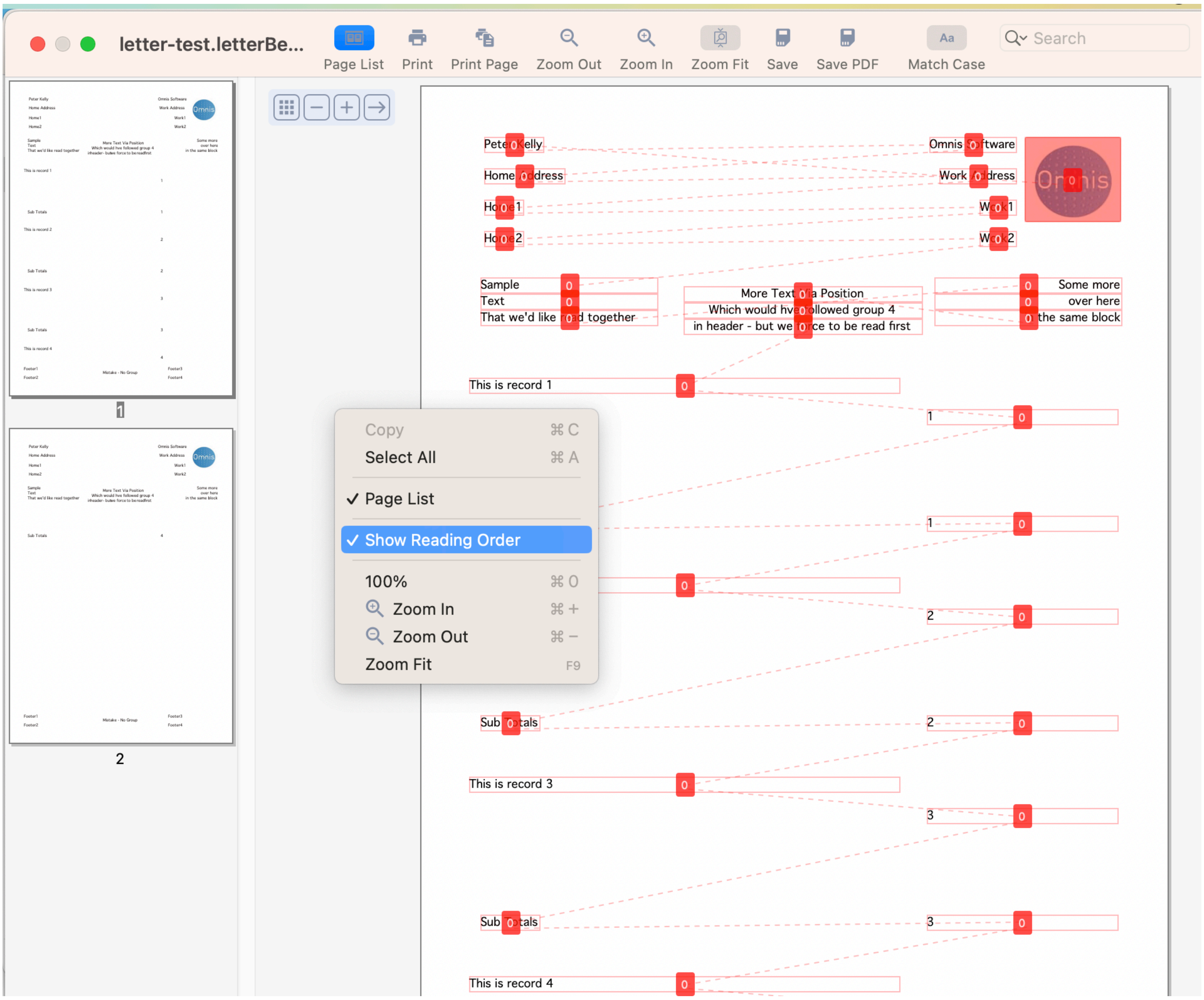

letter-test.letterBe...
Page List
Print
Print Page
Zoom Out
Zoom In
Zoom Fit
Save
Save PDF
Match Case
Search
Copy ⌘ C
Select All ⌘ A
✓ Page List
✓ Show Reading Order
100% ⌘ 0
Zoom In ⌘ +
Zoom Out ⌘ −
Zoom Fit F9
Sample
Text
That we'd like read together
More Text Via Position
Which would hve followed group 4
in header - but we force to be read first
Some more
over here
the same block
This is record 1
This is record 3
This is record 4
Sub Totals
1
2

Figure 137:

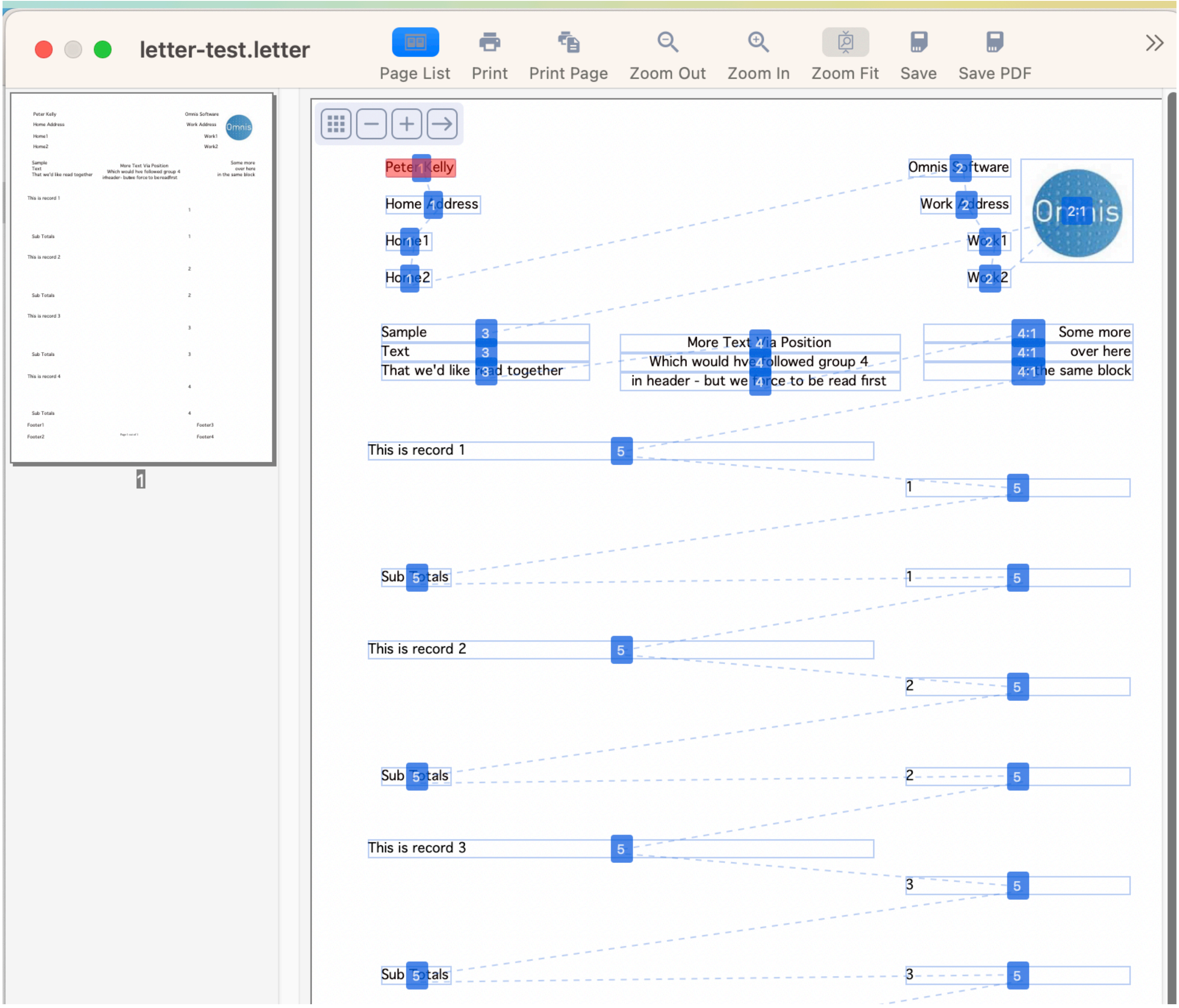

letter-test.letter
Page List
Print
Print Page
Zoom Out
Zoom In
Zoom Fit
Save
Save PDF
Peter Kelly
Home Address
Home1
Home2
Omnis Software
Work Address
Work1
Work2
Omnis
Sample
Text
That we'd like read together
More Text Via Position
Which would hve followed group 4
in header - but we force to be read first
Some more
over here
in the same block
This is record 1
1
Sub Totals
1
This is record 2
2
Sub Totals
2
This is record 3
3
Sub Totals
3
1

Figure 138:

```
studio111://%7B%22lib%22%3A%22test%22%2C%22test%22%3A10%2C%22value%22%3A%22another%22%7D
```

When the custom URL arrives at Omnis, various characters must be escaped, e.g. { becomes %7B, } becomes %7D, : becomes %3A. Some clients, such as Chrome on Windows, automatically escape these characters, whereas on macOS, Safari does not, and you need to do this manually. It is therefore safer to escape all characters. You can use the *OW3.$escapeuritext(cTextToEscape)* function to escape URI/URL characters in order to render them safe.

You can create a custom link in a report by adding the URL to the $linkaddress property of a report Entry field. For example, you could add studio111://lib=testlib/ to create a link to launch Omnis and open a library called 'testlib'. You can also use an Html Text (Link) report object, adding your custom link the $address property. When you print the report to an Omnis Preview window or a PDF, the custom link will be embedded into your report.

Opening a custom link

To open a custom link in a PDF, the end user must right-click the link and open the target in a new tab. Some PDF viewers/browsers may not allow such custom links to run for security reasons, so your custom links should be thoroughly tested for different PDF clients.

When you use a custom URL to run Omnis, Omnis will start up if necessary, and then it runs the **$urlinvoked** method in the Startup_Task of the specified library (identified by the lib column of the parameters). There must always be a column named lib to identify the startup task; if the lib column is missing, or the library is not open, Omnis writes a message to the trace log.

The $urlinvoked method is listed in the Method Editor under the Class methods for the Startup_Task, which you can override and then add your own code. The method has two parameters:

- **$urlinvoked**(cScheme,wParams) called when Omnis is invoked using a custom URL scheme
  cScheme is the custom URL scheme,
  wParams is a row representing the JSON or query string URL parameters.

On Windows, there is an executable called omnisopenurl.exe, in the same directory as omnis.exe, which is used to open URLs, that opens Omnis if necessary, and then sends the request to Omnis. When Omnis starts up on Windows, it checks the registry to see if any of the custom URL schemes in config.json need to be registered – it will only register a scheme if it is not already present in the registry, or if the registry entry of an existing scheme looks like it was created by Omnis. When registering a scheme, the name of the executable used to open a URL, is the currently running core executable name, with openurl appended (the registry key is added to: HKEY_CLASSES_ROOT\studio<version>\shell\open\command).

Printing Reports

In design mode, Omnis provides a wide range of choices for printing or previewing your reports, including sending your report to a Page Preview, the Printer, to a Text or HTML file. For web and mobile apps, you can generate a PDF file from a report and display it in the end users web browser or allow them to download the file. For desktop applications, end users can set the report destination using the **File>>Print Destination** menu option. In your finished library, you can provide a menu, popup menu, or toolbar button to print your report to the required destination. There are a number of commands that let you set the print destination, including *Send to Page Preview, Send to file,* and *Send to clipboard*.

While you are creating your report class you may need to print it to preview or test it. You can use one of the buttons on the report editor toolbar to print the current report class. From these tools you can print to a Page Preview window or the current Printer.

Report Destination Dialog

Note that the Print Destination dialog is only relevant for desktop apps. For web & mobile apps, using the JavaScript Client, the file can be generated on the server and viewed in the browser on the end user's mobile device: see PDF Printing.

You can select the output destination or device for your reports from the Print Destination dialog, available from the **File>>Print Destination** option on the main Omnis menu bar, which is also available to the end user in the Omnis Runtime version. This dialog may also contain any custom devices, such as the HTML device.

The Print Destination dialog includes the following report destinations.

- **Preview** or Page Preview (the default; see below)
  reports are sent to a report *Preview* window which is displayed in the Omnis application window; this is convenient for the end user to view the report, prior to or instead of sending the report to the Printer.

- **Printer**
  the report is sent to the current printer

- **Disk**
  the report is sent to the file specified in the Parameters pane of the Print Destination dialog; the file is stored in a cross-platform proprietary binary format

- **Clipboard**
  reports are sent to the clipboard in text format

- **Port**
  the report is sent to the port specified in the Parameters pane

- **File**
  the report is sent to a text file specified in the Parameters pane

- **RTF**
  the report is sent to an RTF file specified in the Parameters pane

- **HTML**
  the report is sent to an HTML file (UTF-8 encoded)

- **PDF**
  the report is sent to a PDF file and can be opened on the end user's desktop using the Adobe Reader or default PDF viewer.

- **Memory** and **DDE/Publisher**
  are also available but by default are not visible in the Print Destination dialog (you can set their $visible device preference to make them visible; see below)

The *Page Preview* destination is the default report destination; in versions prior to Studio 10.2 the *Screen* report destination was the default, but this has been deprecated and maps to Preview in converted apps.

To set the report destination

- Select File>>Print Destination from the main Omnis menubar

- Select a report destination and click OK, or double-click on a destination

Page Preview Destination

The **Page Preview** (or Preview) option displays a full page on the screen and is the default report destination (from Studio 10.2 onwards). Text is "Greeked" if the screen size is too small, i.e. dots representing the characters so that the whole page fits the available screen area.

Hyperlinks are supported in Page Previews: see HTML Link objects under PDF for details.

You can have more than one report Preview open at a time. Omnis displays a preview as soon as it has prepared the first page of data, that is, normally it does not wait for the report to finish. Therefore, as you scroll or Page down a long report it may take a few seconds to print each page.

The Page preview window can be opened maximized by specifying /MAX in the *Send to page preview* command parameters or in $windowprefs.

Preview toolbars

Preview reports have a toolbar at the top of the window to allow end users to print the report to the current Printer or Save the report to a file. The Omnis root preferences $reporttoolbarscreen and $reporttoolbarpagepreview (in the $root.$prefs devices group) allow you to select which buttons are shown on the Preview toolbar for end user reports.

Zoom Factor

You can specify the initial zoom level for the Page Preview report destination when executing the *Send to page preview* command, or setting the $windowprefs preference. The '/zoom=n' parameter can be included directly after the 'Title' parameter, as follows:

- **Send to page preview** syntax is now
  Send to page preview ([Do not wait for user][,Hide until complete]) title[/zoom=n][/left/top/width/height/cen/max/stk]

- **$prefs.$windowprefs** setting is
  Title[/ZOOM=n][/left/top/width/height/CEN/MAX/STK]) sets the position and initial zoom factor for preview windows

The string /zoom=n is optional, where n is an integer zoom factor, while a zero value means zoom fit. If omitted the Preview window is zoomed to fit the screen.

A value for n of -1 to -7 means use the standard zoom factor indexed using -n (1 to 7); this corresponds to the 7 standard zoom factors for the window, in ascending order.

A positive value means use the standard zoom factor closest to, but not exceeding, n. So you can pass in 175 for example to have an initial zoom factor of 175%.

Copying from Preview reports

You can copy graphics from a report preview window by selecting an area with the mouse and using the Edit>>Copy menu item. You can copy text in the same way, and you can select and copy more of the report than is displayed on screen.

You can use the tab key to tab through the page list, page and search field in the Preview screen. Therefore, to copy content from a preview screen, you can tab to the page if necessary, press Cmd+A to Select all and then Cmd+C to copy content.

When the page has the focus, you can use the Escape key to clear the selection.

The Omnis root preference $disablereportcopy (in the $root.$prefs devices group) allows you to disable the copy via selection feature of screen reports and page previews for end user reports.

Save PDF on Print Preview

The Page Preview window has a Save PDF button. You can control whether this button is present for user reports, using the root device preference $reporttoolbarpagepreview: the constant kRBsavePDF controls whether or not the button is present.

Printer Report Destination

The Printer option sends the report to the current printer. Under Windows, selecting the printer as destination opens a list of installed printers, and changing to a new printer does not affect the default printer setup as defined in Windows Printer Settings.

Under Windows, another application can change the default printer. You can use the Omnis preference **$printernotify** to manage how Omnis responds to the default printer changing. The preference can be set to a constant as follows: **kPrtNoteMsg,** (the default) a notification message is displayed in Omnis allowing the end user to change printer or not; **kPrtNoteAuto,** no notification message is displayed, and the printer changes automatically; **kPrtNoteNoMsg,** no notification message is displayed, and the printer does not change. You can change $printernotify in the Omnis Preferences or Options under the Tools>>Options menu or toolbar option.

Note that you can change the page setup with the File>>Page Setup menu item.

Disk Report Destination

The Disk file report device sends the report output to a file on disk in a cross-platform binary format. If you double-click on the Disk icon in the Report Destination dialog Omnis prompts you for a disk file name.

You can print to the Disk device on one platform, reload the file in Omnis and print it on another platform. Alternatively, you can print the output from the Disk device using the File>>Print Report From Disk menu option, or using the *Print report from disk* command.

Clipboard Report Destination

The Clipboard option sends the current report to the clipboard as an unpaged, text-only report suitable for pasting as text into other applications.

Port Destination

The Port option sends the current report to a Unix or Windows serial or parallel port, or a Mac Modem or Printer port specified in the Parameters pane. This device also uses the settings in the Page sizes pane: see the File device.

Only one program can have a particular port open; if a port is open in Omnis and it is also, for example, the port used by the Spooler, then the Spooler will not be able to function.

Under macOS, there is an option on the Parameters pane to Convert for Imagewriter. When selected, the characters beyond ASCII 127 convert to a combination of a character, backspace, and accent character so that the report can print accented characters and umlauts.

Modern Mac computers do not have serial ports, but you can plug a serial adaptor into a USB port. Once the drivers for your serial adapter are installed correctly, Omnis Studio will automatically recognize any new serial ports when you plug the adapter into your machine. Any new serial ports that are discovered will be displayed in the parameters panel for you to select.

NOTE: Serial adapters and other USB devices which do not support the Comms Toolbox standard will not be recognized by Omnis Studio.

You can save the settings for the Port device to a profile using the Port profile editor: see later in this chapter for a description of the Port Profile editor. You can also setup the port using the *Set port parameters* command.

File Report Destination

The File print destination sends the current report to a file. If you double-click on the File icon in the Report Destination dialog Omnis prompts you for a file name. Omnis does not close the file at the end of the report so you can append multiple reports into a single file. This option enables the Page sizes pane in the Report Destination dialog.

In the Page sizes pane you can specify the number of lines per page to use in reports printed to a file or a port. Omnis stores the setting in the Omnis configuration file.

If you check the Generate paged reports check box, you can also check the Send form feed check box, which tells Omnis to terminate pages with a form feed, or fill in the Line per page field with a number of lines to which to pad out each page. Checking the Restrict page width option lets you enter the number of Characters per line.

RTF Report Destination

The RTF print device sends the current report to an RTF file. If you double-click on the RTF icon in the Report Destination dialog Omnis prompts you for a file name. You can also set the file name under the Parameters pane in the Report Destination dialog, as well as control the how images are embedded or linked. The default behavior is to embed images in the RTF file, but you can link the images in which case any images are stored as separate files and linked to the RTF file; you can also ignore the images altogether.

The following code allows you to set the name of the RTF file:

```
Do $devices.RTF.$setparam(kDevRtfFileName,'/Users/test/Desktop/test.rtf')
```

HTML Report Destination

The HTML report device is a custom device that prints a report to an HTML file on disk. When installed and loaded it appears in the print destination dialog and behaves like any other standard printing device. You can send any Omnis report to the HTML device and access and change the device using the notation. If you double-click on the HTML icon in the Report Destination dialog Omnis prompts you for a file name. The HTML output uses UTF-8: if you use a template file, that must also be UTF-8 encoded.

The HTML printing device uses HTML tables and standard HTML tags to position and structure the output of the Omnis report. The default background color of the HTML file is white. The color of the text in the original report class is retained in the HTML output file. Where possible, the device converts any image or picture data into JPEG images, which are written to disk and linked to the output HTML file.

PDF Report Destination

The PDF report destination sends the report output to a PDF file which can be viewed in a browser in the JS Client or on the end user’s desktop using the default PDF viewer.

The alternative of using the Printer report destination with $macosdesttype set to PDF uses bitmaps to render background objects, and their appearance in a scaled PDF will vary depending on the scaling factor and the algorithm used by the PDF viewer to scale the bitmap.

For PDF (and Page Preview) reports you can embed an HTML link (URL or mail) behind some Text, an Icon, or a Picture using the HTML Link Objects.

Printing Background Images to PDF

Background images on reports need to be shared pictures (PNGs) to print on the Linux headless server. To solve this, go to the library prefs and set $sharedpictures to kSharedPicModeTrueColor. Then open each affected report class in the report class editor; Omnis will ask if pictures are to be converted to shared, so respond with Yes. The reports will now print to PDF in the Linux headless server, and additionally the file size of the report classes will be much smaller.

Omnis will report an error if the Linux headless server cannot print to PDF.

PDF Font Mapping

When using custom fonts for PDF printing there may be a mis-match between the name of a font and its Window registry entry, which results in the font not being found and the report not being rendered correctly. To rectify this, you can add mappings to the "pdf" entry in config.json (that apply to the Windows platform only), to map a font name to its entry in the registry. For example, you can map the font name "Proxima Nova Rg" to its registry entry "Proxima Nova Regular", using the following item in the config.json file.

```
"pdf": {
  "plainSuffixes": "Regular,Standard,Normal,Normale",
  "Proxima Nova Rg": "Proxima Nova Regular",
  "Proxima Nova Rg Bold": "Proxima Nova Bold"
},
```

Memory Report Device

The Memory device lets you send a report to a binary variable or field, which you can hold in memory or save in a database. At a later date you can reload the contents of the binary variable or field and print the report to the printer or any other destination.

You can access the Memory device using the notation only via the $root.$devices notation group. By default this device *is* not shown in the Print Destination dialog, but you can show it using the following method (although in practice you would not use this destination for end users):

```
Do $root.$devices.Memory.$visible.$assign(kTrue)
```

You can print the output from the Memory device using the *Print report from memory* command.

DDE/Publisher Device (Obsolete)

This feature is no longer supported, since it relates to very old Windows and Mac operating systems, but is still available for backwards compatibility. The DDE/Publisher device lets you send a report via DDE under Windows, or to an "Edition" under macOS.

You can access the DDE/Publisher device using the notation only via the $root.$devices notation group. By default this device *is not shown* in the Print Destination dialog, but you can show it using the following method (although in practice you would not use this destination for end users):

```
Do $root.$devices.//DDE/Publisher//.$visible.$assign(kTrue)
```

Screen Report Destination (Obsolete)

The **Screen** print destination was available in versions prior to Studio 10.2 but it is now obsolete; it maps to the Preview report destination by default. The option "useScreenDestination" in the "defaults" section of config.json does allow you to send a report to the screen if required.

Printing Errors

The print manager reports the following error codes and text. You can setup error handlers to manage these errors.

Error	Description
1001650	Non-fatal print manager error
1001670	Fatal print manager error
1001680	Print manager system error; the code is shown in the error text
1001681	Other Omnis error reported by print manager

Tabs in Reports

The "replaceTabsInRTFwithSpacesWhenAddingToReport" item in the "docview" group in config.json sets the default tab width when printing text containing tabs in reports, so they are displayed properly, e.g. in the Page preview. The config item can be a value from zero to 32 inclusive, the default is 2. Zero means leave the text unchanged. 1-32 means replace each tab character found in the text with 1-32 spaces, when adding the text to a print job.

You can use the $settabwidth method to override the default tab width set in config.json.

Using style() in Reports

In versions prior to Studio 10.x, if the style() function was used in a report instance belonging to a remote task incompatible results were generated (but this has been fixed).

The style() function usually generates different results when used in a remote task instance, and the resulting style is suitable for use with the JavaScript client. The fix allows normal, non-JS client results, to be generated by style() when running in a report instance inside a remote task.

However, even with this fix, you should note that the call to print the report from the remote form, which passes the results of style() from some remote form code, will not work: you need to pass the icon id as the parameter, and call style() from within the report instance to make this work with this fix.

Report and Field Methods

You can create a report class, add fields and objects to the report from the Component Store, but to print sophisticated reports you will need to add some programming behind the fields and sections in your report. To do this, you need to write code that uses the Omnis print commands or methods. You can add *class methods* to the report itself to control printing, and you can add *field methods* to each field or section marker on your report to control things like the interval breaks and subtotals.

You can add up to 501 methods to each field or section on your report, and a further 501 methods to your report class. You enter the methods for a report class and its fields using the method editor.

A report class will contain a $construct() and $destruct() method by default. You can add code to these methods to control the opening and closing of the report instance. For example, if your report uses a list you can build the list in the $construct() method in the report. You can use the $open() method or the *Prepare for print* command to open a report instance, you can finish a report using $endprint() or the *End print* command, and you can close a report instance using the $close() method. You can send a list of parameters to the $construct() method when you open the report instance using the $open() method. You can send data to a report instance record by record using the *Print record* command, or print an entire report using the *Print report* command. Alternatively, you can send print messages to a report instance using the notation. For example, you can send a $printrecord() message to print a record to the report instance, or send an $endprint() message to finish the report; there is no equivalent method for the *Print report* command. You can override the default handling for these messages by writing your own custom methods with the same name. You enter these custom methods in the class methods for the report class.

To add a method to a report class

- Open your report class
- Right-click on the report background to open the report context menu
- Select the **Class Methods** option
- Right-click in the method list in the method editor and add your method

To add a method to a report field or section

- Open your report class
- Right-click on the field or section to open its context menu
- Select the **Field methods** option
- Right-click in the method list in the method editor and add your method

Report Data Grid Column Parameters

Column calculation properties (in $::calculation) for the Report Data Grid component are tokenized so that they work with the current function parameter separator.

For compatibility with versions prior to Studio 11, the component still works with the $::calculation and $columnheader properties stored as character strings, provided that the function parameter separators match those currently in use. When you re-enter one of these properties (select the property in the Property Manager and press Return) the property changes so that it is stored as a tokenized calculation, which will then work with any function parameter separator.

An alternative way to convert a library is to export to JSON and re-import, and in this case Omnis tokenizes the calculations on import. For import, Omnis will accept the report list calculations as either character strings or calculations; import always results in tokenized calculations being stored. Accepting both forms means that import is compatible with JSON exported in previous versions.

Print Devices and the Current Device

The following properties under $root handle the group of currently installed print devices or destinations, and the current printing device.

- **$devices**
  group of currently installed printing devices, including Printer, Preview, Screen, Disk, Memory, Clipboard, Port, File, DDE/Publisher, and any custom devices you may have installed. You can set a reference to a device by using

```
Set reference MyRef to $devices.Screen
```

- **$cdevice**
  the current printing device or report destination. You can change $cdevice by assigning a device from the $devices group, for example, to specify the screen as the current device use one of:

```
Calculate $cdevice as kDevPreview
Calculate $cdevice as $devices.Screen
Calculate $cdevice as $devices.$findident(kDevPreview)
```

Print Devices

The $root.$devices group contains the currently installed printing devices plus any custom devices you may have installed. A device has the following properties; $canassign for these properties is true unless specified.

- **$name**
  the name of the device; $canassign is false
- **$title**
  the string used to identify the device in the Print destination dialog
- **$iconid**
  the id of the icon for the device as displayed in the Print destination dialog, zero by default which means the device uses the default icon
- **$ident**
  a unique numeric identifier for the device in the $devices group; $canassign is false
- **$visible**
  if true, the device is shown in the Print destination dialog
- **$isopen**
  returns true if the device is open and in use; $canassign is false
- **$istextbased**
  returns true if the device is text-based, otherwise, the device is image-based, such as Printer, Screen, or Preview; $canassign is false
- **$cangeneratepages**
  this is a read only property. If it returns true, the device can generate pages and all normal page headers and footers will be printed. If it returns false, only the report header and first page header are printed. No footer section is printed.
- **$cankeepopen**
  returns true if the device can be kept open for long periods of time, such as the File device; otherwise, the device should be opened prior to printing and closed immediately after printing has finished, for example you must do this for the Printer; $canassign is false

You can use the following methods for a device; $cando() returns true if a device supports the method.

- **$open()**
  opens the device ready for printing or transmitting text or data. Some devices such as the Screen or Preview can only be opened from a print job when printing a report
- **$close()**
  closes the device, if the device is open

The following example prints two reports in the same print job, and uses the $open() and $close() methods to initialize the Printer.

```
Set reference theDevice to $devices.Printer
If theDevice.$isopen ## check if printer is in use
  If theDevice.$canclose()
    Do theDevice.$close()
  Else
    Quit method kFalse ## if device can't be closed
  End if
End If
Do theDevice.$open() Returns ok ## open the printer
If ok ## print the reports
  Set report name reportOne
  Print report
  Set report name reportTwo
  Print report
  Do theDevice.$close() Returns ok ## close the printer
End If
```

- **$canclose()**
  returns true if the device can be closed. If you opened the device using $open(), this method returns true; if you opened the device via a print job and the job is still in progress, it returns false

- **$sendtext**(cText, bNewLine, bFormFeed)
  sends the text in cText to the current device; all normal character conversion takes place. If bNewLine is true, the device advances to a new line or sends an end of line character; if bFormFeed is true, a new page is started, or a form feed character is sent. Data is sent in parameter order: first text, then the new line, then the form feed.

The following example sends some text to the File device.

```
Set reference theDevice to $devices.File
If theDevice.$sendtext.$cando()
  Do $prefs.$printfile.$assign('HD:MyFile')
  Do theDevice.$open() Returns ok
  If ok
    Do theDevice.$sendtext('Some text',kTrue) Returns ok
    Do theDevice.$sendtext('More text',kTrue) Returns ok
    Do theDevice.$close() Returns ok
  End If
End If
```

- **$senddata**(cData[,cData1]...)
  sends the specified data in a binary format to the device; no character conversion takes place unless the data is of type kCharacter. If more than one parameter is specified the data is sent in individual packets

When using the $senddata() method you must consider type conversion. The method expects binary data, and therefore any data which is not in a binary format is converted to binary. For example, if you pass an integer variable, the data is converted to a 4 byte binary. In some cases, due to cross platform incompatibilities, if you want to be certain of the order in which the data is sent, and of the values which are sent, you should use variables of type Short integer (0 to 255), for example

```
Calculate myShortInt1 as 13
Calculate myShortInt2 as 10
Do myDevice.$senddata( myShortInt1, myShortInt2 )
```

You can send raw data to the Port or File device. The following example prints a report to a binary variable and sends the binary data to the Port device.

```
# print a report to a binary variable
Do $cdevice.$assign($devices.Memory)
Do $prefs.$reportdataname.$assign('myBinaryField')
Set report name myReport
Print report
```

```
# now send the report to the port
Set reference theDevice to $devices.Port
If theDevice.$senddata.$cando()
  Do theDevice.$open() Returns ok
  If ok
    Do theDevice.$sendata(myBinaryField) Returns ok
    Do theDevice.$close() Returns ok
  End If
End If
```

- **$flush()**
  flushes the device. For the File device $flush() will ensure all data is written to disk; you can safely call $flush() for devices which do not support this method; $cando() returns true for all devices that support $senddata() or $sendtext()

Global Printing Preferences

There are a number of Omnis preferences under $root.$prefs that handle the print devices and their parameters. You can set these using the Property Manager or using the notation.

- **$reportfile**
  the full path and file name for the Disk device
- **$printfile**
  the full path and file name for the File device
- **$editionfile**
  the full path and file name for the DDE/Publisher device
- **$pages**
  the page or page numbers to be sent to the device; all devices support this property. You can specify pages as a comma-separated list or range of pages separated by a hyphen, or any combination. Prefixing a range with an “e” will print even pages within the range, or with an “o” will print odd pages within the range. For example
  1,3,7,10-15,25-20,e30-40,o30-40
- **$reportdataname**
  the name of the binary field for the Memory device
- **$reportfield**
  the name of the window field for a Preview or Screen report; if you specify this property the report is redirected to the window field
- **$windowprefs**
  the optional title and screen coordinates for a Screen or Preview window; the syntax is the same as the *Open window command*, such as My Title/50/50/400/300/STK/CEN; the title is also used as the document name when printing to the Printer
- **$waitforuser**
  if true, method execution is halted until the user closes the Screen or Preview window
- **$hideuntilcomplete**
  if true, a Screen or Preview window remains hidden until the report is finished

The following example specifies the Preview as the current device and sets up the preferences for the report window.

```
Do $cdevice.$assign(kDevPreview)
Do $prefs.$windowprefs.$assign('MyTitle/20/20/420/520/CEN')
Do $prefs.$waitforuser.$assign(kFalse)
Do $prefs.$hideuntilcomplete.$assign(kTrue)
```

- **$charsperinch**
  the number of characters per inch when printing to a text-based device
- **$linesperinch**
  the number of lines per inch when printing to a text-based device

- **$generatepages**
  if true, reports generate paged output when printing to text-based devices, that is, page headers and footers are generated as normal; otherwise if false, only one report header and page header is printed at the beginning of the report

- **$linesperpage**
  the number of lines per page when $generatepages is true

- **$restrictpagewidth**
  if true, the width of a page is restricted when printing to text-based devices

- **$charsperline**
  the number of characters per line when $restrictpagewidth is true

- **$sendformfeed**
  if true, form feeds are sent to text-based devices after each page

- **$appendfile**
  if true, data is appended to the current print file specified in $printfile, otherwise if false, the file is overwritten when printing to the File device; note if the device is already open prior to printing a report, the file is appended to regardless

- **$istext**
  if true, forces a non-text device to behave like a text-based device using the same preferences as text-based devices

- **$portname**
  the name of the port when printing to the Port device (Not supported on macOS)

- **$portspeed**
  the port speed setting when printing to the Port device (Not supported on macOS)

- **$porthandshake**
  the handshake when printing to the Port device; this can be kPortNoHandshake, kPortXonXoff, or kPortHardware (Not supported on macOS)

- **$portparity**
  the parity checking when printing to the Port device; this can be kPortNoParity, kPortOddParity, or kPortEvenParity (Not supported on macOS)

- **$portdatabits**
  the number of databits when printing to the Port device; this can be kPort7DataBits, or kPort8DataBits (Not supported on macOS)

- **$portstopbits**
  the number of stop bits to be used when printing to the Port device. This can be kPort1StopBit or kPort2StopBits (Not Supported on macOS)

The following example sets up the preferences for the Port device.

```
Do $prefs.$portspeed.$assign(9600)
Do $prefs.$porthandshake.$assign(kPortNoParity)
Do $prefs.$portdatabits.$assign(kPort8DataBits)
Do $prefs.$portstopbits.$assign(kPort1StopBit)
Do $prefs.$porthandshake.$assign(kPortXonXoff)
Do $prefs.$charsperinch.$assign(10)
Do $prefs.$linesperinch.$assign(6)
# Note $charsperinch and $linesperinch are used for
# all text-based devices
```

There is also a group of Page setup properties under $root.$prefs giving access to the global page settings. These are

- **$orientation**
  the page orientation; this can be kOrientDefault, kOrientPortrait, or kOrientLandscape

- **$paper**
  the paper type or size, a constant; one of 50 or so paper sizes or types including US Letter, European A sizes (from A6 upwards), envelope sizes, custom sizes, and so on

- **$paperlength**
  the length of the paper in cms or inches depending on the $usecms preference

- **$paperwidth**
  the width of the paper in cms or inches depending on the $usecms preference
- **$scale**
  the scaling factor in percent
- **$copies**
  the number of copies

The Omnis root method $getprinterlist() allows you to get a list of printers available to the current user. For example, the following method will return a list of printers:

```
# define local vars printer (Char), mylist (List)
Do mylist.$define(printer)
Do $root.$getprinterlist(mylist)
```

The current line of the list indicates the current printer.

The Omnis preference $disablereportworkingmessage allows you to disable working messages for reports, which you might want to do when printing to a Print Review window.

- **$disablereportworkingmessage**
  If true, the 'Sending report to…' working message is not shown when printing a report.

This property only applies to reports being printed on the main thread (as reports in JavaScript client threads do not show a working message). Note that you cannot cancel the report if you set this property to true. You may also need to use $modes.$fixedcursor and $modes.$ccursor if you want to display a cursor other than the busy cursor while printing the report.

Report Instances

Report instances have the following methods.

- **$printrecord()**
  prints the Record section; same as the *Print record* command
- **$printtotals**(*section*)
  triggers a subtotal or totals section; *section* is the highest level subtotal to be printed, a constant, such as kSubtotal5 or kTotals
- **$printsection**(iSection[,bPosnIsRecord=kFalse])
  This is sent when a section is printed; iSection is one of the constants (kRecord, kTotals, etc.) or a reference to a section field on the report instance. The default handler prints the section positioned according to $sectionstart, $sectionend and the positioning mode for the section. For a Subtotal or Total section the current field values for fields whose $totalmode is not set to kTmNone are temporarily reset to those which were current when $printsection for a detail section was previously called.
  bPosnIsRecord applies to positioning sections only; if bPosnIsRecord is kFalse (the default), this method prints a section based on the position of the previous section; otherwise, when true the section is printed as a record
- **$accumulate**(*section*)
  accumulates the subtotals and totals section, and is sent during the printing of a record section
- **$checkbreak()**
  checks if a subtotal break is required, returns a constant: kSubtotal1 to kSubtotal9 or kNone if subtotal break is not required
- **$skipsection()**
  skips the current section; if you call this during $print() for a field, no further fields will be printed for that section
- **$startpage**(*pagenumber*)
  starts a new page; adds the page header section to the page, and for the first page also adds the report header section
- **$endpage**(*pagenumber*)
  ends a page and adds the footer section to the page; without parameter ends all pages which have been started
- **$ejectpage**(*pagenumber*)
  ejects a page; without parameter ejects all pages which have been ended and not ejected; this method ejects pages which have an active section intercepting their boundary when the section has finished printing

- **$endprint()**
  finishes the report; prints the final subtotals and totals sections and ejects all the remaining pages

- **$openjobsetup()**
  opens the job setup dialog. You can call this method immediately after $open() for a report; if it returns kFalse as the result, the user has selected Cancel, and you should close the report instance. You cannot call $openjobsetup() during $construct() since a print job is not created until $construct() finishes.

- **$cdevice**
  reference to the printing device for the instance; if you wish to change the device, you must do so before returning from $construct() of the report instance, and before you start printing the first record. For example, execute the following at the start of $construct() to specify the page Preview device for the current instance

```
Do $cinst.$cdevice.$assign($devices.Preview)
```

The device preferences are listed under the global printing preferences. Report instances have their own printing preferences which are local to the instance. They take their initial values from the global printing preferences. You can only assign values to these properties in $construct(), and before you start printing the first record.

The $firstpage property always returns 1, and $canassign() is false. The $lastpage property returns the last page. You cannot set $lastpage in the report instance to reduce the number of pages generated, that is, once pages have been generated they cannot be removed from a print job.

Note that the $pageheight property of a report instance returns the height of the printable area excluding the margins, headers, and footer areas of the report.

The following method generates subtotal breaks every fifth record. The $reccount property is incremented and the subtotals accumulated manually.

```
Do $reports.Report2.$open('*') Returns Myreport
For lineno from 1 to Mylist.$linecount step 1
  Do Mylist.[lineno].$loadcols()
  Calculate Myreport.$reccount as Myreport.$reccount+1
  Do Myreport.$printsection(kRecord)
  Do Myreport.$accumulate()
  If mod(lineno,5)=0
    Do Myreport.$printtotals(kSubtotal1)
  End If
End For
Do myreport.$endprint()
```

Page Setup Report instance properties

You can change the page setup information of a report instance without effecting the global settings. The properties which can be set are;

- **$pagesetupdata**
  this can only be calculated prior to printing the first record; the best time is during $construct. When a print job has started, $canassign returns kFalse.

- **$orientation, $paper, $paperlength, $paperwidth, $scale,** and **$copies**
  any of these properties can be changed at any time during a print job, and will effect the next page to be generated. When $startpage for a page has been called, changing these properties will take effect from the next page onwards. A good time to make changes for the next page is during a $endpage for the current page, but it can be done from anywhere prior to the $startpage call for a page to be effected

- **$loadpagesetup**
  When true, the page setup information stored with the report is automatically loaded and used when the report is printed. This will *not* affect the global page setup information. The page setup information is applied to the page setup information of the report instance prior to calling the $construct method or opening the job setup dialog. It is also possible to load the page setup data from the $construct method of the report instance by assigning $loadpagesetup ($cinst.$loadpagesetup.$assign(kTrue)). Assigning kFalse has no effect., For new reports, this property will be set to kTrue by default. For existing reports, this property will be false.

Once a print job is complete and $endprint has been called, $canassign returns kFalse for all these properties.

Printing a report from a list

The following method prints a report from a list and uses a *For* loop to print the report record by record.

```
Do $reports.Report1.$open('*') Returns Myreport
For lineno from 1 to Mylist.$linecount step 1
  Do Mylist.[lineno].$loadcols()
  Do Myreport.$printrecord()
End For
Do Myreport.$endprint()
```

Note that you should not use hash variables to define the columns in list if it is going to be used as the basis of a report.

Screen Report Fields

You can send the output of a report to a window field, a Screen Report Field, which is similar to a Preview report but contained in the window field. The current page count is reported in the $pagecount property (read only), while $currentpage is the currently displayed page and is assignable at runtime. When more than one page is visible, the value indicates the page that is most visible.

Zoom

You can use the $zoom() method to scale the report from 25% to 200%. The method $zoom(*iZoom*) zooms the screen report field, where iZoom can be positive (indicating a percentage between 25 and 200% inclusive), or 0 meaning zoom to fit, or negative (-1 to -7) where -iZoom indexes the 7 standard zoom factors from smallest to largest.

Search

The method $searchreport(*cText*[,*bIgnoreCase*=kTrue,*bNext*=kTrue]) searches the report for cText. Further calls with the same cText and bIgnoreCase search for the next (bNext kTrue) or previous (bNext kFalse) match. Empty cText clears the search.

There is an event which works in conjunction with $searchreport (needed because the search occurs in a background thread). The event enables you to manage next and previous buttons, and status text. The next and previous buttons are assumed to start in disabled state. The event evReportSearchStatus is sent to the report field when the report search status changes: this has one event parameter pReportSearchStatus which is a row with 4 columns, as follows:

Column	Description
next	If true, search next can be enabled as there is another search result later in the report
prev	If true, search previous can be enabled as there is another search result earlier in the report
count	The count of search results
index	The 1-based index of the current search result

Report Field and Section Methods

Report fields and sections contain a **$print()** method that controls that particular field or section when it is printed. Every time a field or section is encountered during printing its $print() method is called, so for fields in the report Record section $print() is called for every row of data. You must end your own custom $print() methods with a *Do default* command to carry out the default processing for that line after your code has executed.

For example, the following $print() method for a report field prints the field in bold if its value is greater than 1000.

```
If parm_value>1000
  Do $crecipient.$fontstyle.$assign(kBold)
Else
  Do $crecipient.$fontstyle.$assign(kPlain)
End If

Do default
```

Report Object Positioning

When a report field prints, its position and data are passed to its $print() method; when a report section prints its position only is passed. You can set up parameter variables of type Field reference in the $print() method for a report section or field to receive its position and data. You can manipulate the position variable using the report object positioning notation. If you change the position of a section all objects in that section are affected together with all subsequent sections in the report. Making changes to the position of an object does not affect other objects.

A report position variable has the following properties.

- **$inst**
  the report instance to which the position belongs
- **$posmode**
  the mode of the report position, which is one of the following constants; assigning $posmode does not change the physical position of the object, but it does change its coordinates to the new coordinate system.

Constant	Description
kPosGlobal	the position is global to the print job, relative to the top-left of the local area of the first page
kPosPaper	the position is relative to the top-left of the paper edge of the page specified by $posvertpage and $poshorzp
kPosPrintable	the position is relative to the top-left of the printable area of the page specified by $posvertpage and $posho
kPosLocal	the position is relative to the top-left of the local area (excluding the header and footer sections, and the mar specified by $posvertpage and $poshorzpage
kPosHeader	the position is relative to the top-left of the header area (union of report and page header sections) of the pag $posvertpage and $poshorzpage
kPosFooter	the position is relative to the top-left of the footer area of the page specified by $posvertpage and $poshorzp
kPosSection	the position is relative to the top-left of the section specified by $possectident

In addition, you can set $posmode to one of the following values to return the coordinates of an area on the page specified by $posvertpage and $poshorzpage.

Constant	Description
kBndsGlobal	returns kPosGlobal coordinates. The top, left, width, and height are calculated to global coordinates of the loc
kBndsPaper	returns kPosPaper coordinates. The top and left are zero, and the height and width are calculated to the heig the page
kBndsPrintable	returns kPosPrintable coordinates. The top and left are zero, and the height and width are calculated to the h printable area of the page
kBndsLocal	returns kPosLocal coordinates. The top and left are zero, and the height and width are calculated to the heig of the page
kBndsHeader	returns kPosHeader coordinates. The top and left are zero, and the height and width are calculated to the he area of the page
kBndsFooter	returns kPosFooter coordinates. The top and left are zero, and the height and width are calculated to the hei area of the page

- **$possectident**
  the $ident of the section when $posmode is kPosSection
- **$posvertpage**
  the vertical page number when $posmode is *not* kPosGlobal or kPosSection
- **$poshorzpage**
  the horizontal page number when $posmode is *not* kPosGlobal or kPosSection. This will usually be set to 1. Horizontal page numbers apply when horizontal pages are enabled
- **$top**
  the top of the position in cms or inches local to its $posmode
- **$left**
  the left of the position in cms or inches local to its $posmode
- **$height**
  the height of the position in cms or inches

- **$width**
  the width of the position in cms or inches

Measurements are in either cms or inches depending on the setting of the **usecms** Omnis preference which you can change in the Property Manager using the Tools>>Options/Preferences menu option.

Page layout

To understand the positioning notation it helps to look at the layout of the report on paper or screen. The area available for printing is limited to the printable area on the paper as determined by the printer or device. Within this space Omnis reports print to the header, footer, and local or global areas, that is, the space remaining after subtracting the header, footer, and margins specified in the class. Note that Omnis subtracts the margins specified in the class from the paper edge, rather than the boundary of the printable area.

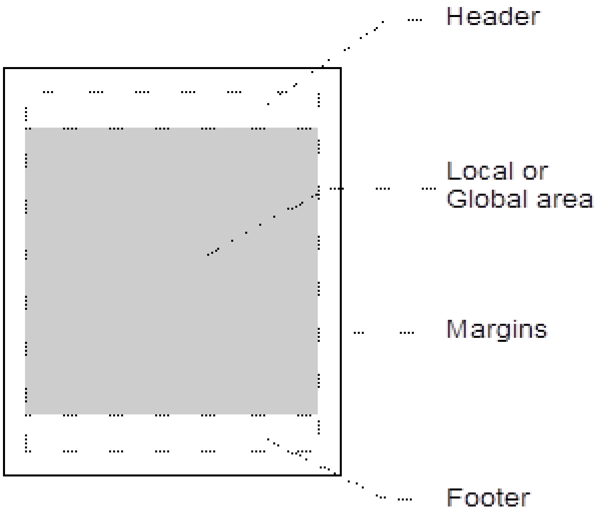
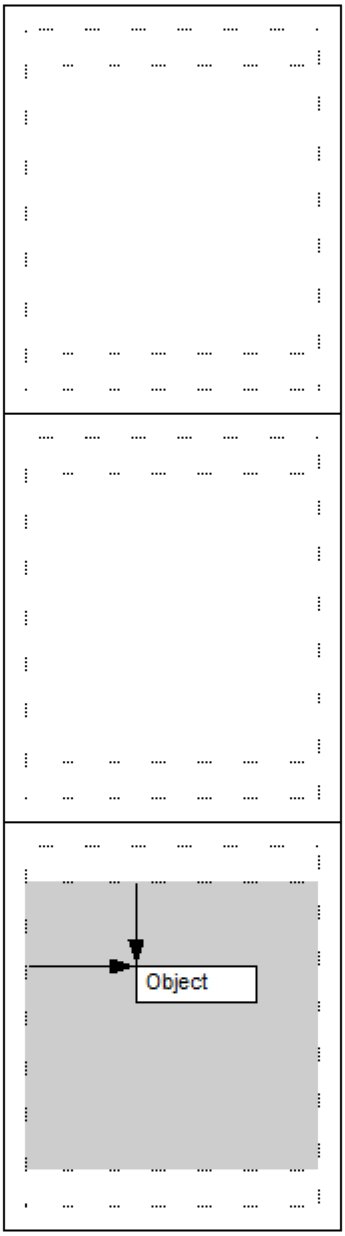
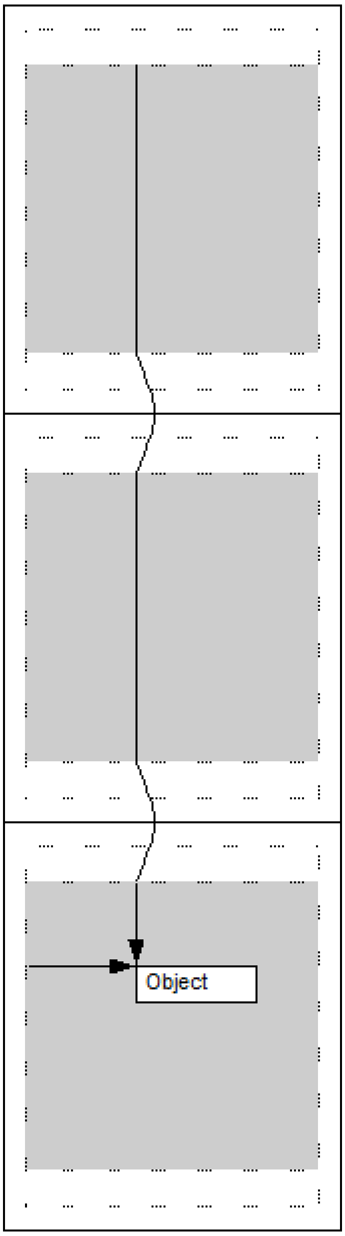


Figure 139:

The position of a report object, either a section or report field, is relative to the local area on the current page, or the global area for the entire report.



Local coordinates are relative to the local area on the current page



Global coordinates are relative to the global area for the entire repor

The following example method produces a report with multiple columns by configuring itself according to the current paper size and orientation. The report class contains various instance variables including iCurColumn, iMaxColumns, iLeftAdjust to handle columns and a global left adjustment. The data is taken from a list, but your data can be from any source. The Record section contains one field that gets its data from the list. Note the code for this method does the positioning and the *Do default* command prints the section.

```
# $print() method for Record section in a column report
# Declare Parameter var pThePos of type Field reference
# and Local var posBnds of Row type
# pThePos is in global coordinates and does not contain the page
# number, so make a copy and convert it to page-based coordinates
Calculate pos as pThePos
Calculate pos.$posmode as kPosLocal
# Fetch the global boundaries for the page: we can do this now since
# setting $posmode to kPosLocal set $poshorzpage and $posvertpage
Calculate pos.$posmode as kBndsGlobal
# Check if the bottom of the section will fit on the page
If (pThePos.$top+pThePos.$height)>(pos.$top+pos.$height)
  # if it doesn't fit is there room for a new col on current page
  If (iCurColumn<iMaxColumns)
    Calculate iCurColumn as iCurColumn+1
  Else
```

```
    # put the section at the top of the next page for column one
    Do pos.$offset(0,pos.$height)
    Calculate iCurColumn as 1
  End If
  # now calculate the section's top position based on posBnds.$top
  Calculate pThePos.$top as pos.$top
  # calculate the section's left pos based on current column number
  Calculate pThePos.$left as (iCurColumn-1)*$cinst.$labelwidth+iLeftAdjust
Else If not(pThePos.$left)
  Calculate pThePos.$left as iLeftAdjust
End If

Do default ## this prints the Record section
```

You can use the $offset(x,y) method to move the object horizontally by x units (unless $horzpages is enabled) and vertically by y units. The units are defined by $prefs.$UseCMS and maybe either positive or negative.

When a report is based on a report superclass several items are inherited including report header and footer sections (e.g. page header, subtotototals, footer, etc), the objects within these sections, as well as the sort fields in the report superclass. The objects within the record section of the report superclass are not inherited. You can, therefore, create a report template containing your company identity and base all other reports on the template to ensure a uniform design and layout is maintained across all your reports.

Inherited sections cannot be manipulated in any way. You cannot resize an inherited section by moving the section object immediately below the inherited section, or change the properties of the section or its objects. When a section is inherited, the height of the section is determined from the superclass, and all other sections below are moved up or down accordingly.

For inherited sections, the text of the section bar and the text of the section connection lines are shown in blue. The background is shown in gray to indicate that this section cannot be manipulated.

Inheriting Sections

Modifying a Superclass

When modifying the section of a superclass, e.g. adding, removing or changing objects within a section, these changes will be reflected in any report class which inherits the section.

Removing an entire section from a superclass will also affect all subclasses. However, adding a new section to a superclass will not affect any of its subclasses. To inherit the new section, subclasses need to be modified to inherit this new section.

Any changes made to sort fields will be reflected in all subclasses.

Inheriting/Overload a section

To inherit or overload a section, right click on the section in the Property Manager. A context menu opens allowing you to inherit or overload the property. You cannot manipulate inherited sections, or inherited section objects, or to add more objects to an inherited section.

Subtotal Sections and Sort Fields

A subclass inheriting a subtotal section will also inherit the sort field from the super class, unless the subclass has already specified a sort field for the subtotal section in question. Sort fields can be overloaded or inherited via the context menu of the sort field. The properties of an inherited sort field cannot be changed. If you want to change the properties of an inherited sort field you must first overload the sort field.

When overloading a sort field, the properties of the sort field originally inherited are maintained. If a sort field is inherited which has not been specified by its super class, the name will be displayed as #???.

Positioning Sections

If an inherited section contains positioning sections, these are also inherited. It is *not* possible to add additional positioning sections to an inherited section. It is not possible to inherit position sections without inheriting their “main section”.

Notation

You can inherit or overload sections using notation. To do this you assign $isinherited of the section notation. For example:

```
Do $clib.$reports.myreport.$pageheader.$isinherited.$assign(kTrue)
```

To inherit sort fields using notation:

```
Do $clib.$reports.myreport.$sorts.1.$isinherited.$assign(kTrue)
```

The objects of inherited sections are not part of the subclass and as such will not appear in the list of objects of the report class, but will appear in the list of objects of a report instance.

The following section properties can be inherited: $reportheader, $pageheader, $subtotalhead, $subtotalhead1 to …9, $subtotal9 to …1, $pagefooter, $totals.

Report Fonts

If you are developing an application for a cross platform environment (for desktop apps), you may want to set up the system font tables to allow the fonts used in your application to map correctly across the different platforms. There is a system font table in a library for report classes and window classes for each platform supported in Omnis.

The fonts in the report font table will appear in the $font property for report objects. So even if you are developing an application for a single platform, you may still want to edit the report font table(s) to add fonts to those already available for report objects.

Font table	Description
#WIRFONTS	Report font table for Windows OS
#MARFONTS	Report font table for Mac (9 & earlier)
#UXRFONTS	Report font table for Unix
#MXRFONTS	Report font table for OSX/macOS (10 onwards)

To view the report fonts system table

- Use the Browser Options dialog (press F7/Cmnd-7 while the Studio Browser is on top) to make sure the system tables are visible
- Double-click on the **System Tables** folder
- Double-click on **#WIRFONTS** or the report font table for your platform

The #WIRFONTS system table contains a list of fonts that are available in Omnis by default. Each row in the font list displays the corresponding font for each platform supported in Omnis. To change the font mapping, replace the name of a font either by typing its name or selecting it from the list of fonts. To add a font, click in the next available line in the font list and add the name of the font. Add a font name for each platform.

The font table editor loads or creates a font table for each platform (corresponding to each column in the editor) and allows you to edit them all simultaneously. Therefore, when you edit the report font table for the first time, and click OK to finish editing it, a new system table is added to your library for each platform supported in Omnis, other than your current platform.

Windows Fonts

Under Windows, all fonts used in reports must be installed and registered. Reports can use fonts that are installed for the *current user* located in:

```
C:\Users\USER\AppData\Local\Microsoft\Windows\Fonts
```

and registered here in HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Fonts. Such fonts can be installed by right-clicking on the .ttf font file and selecting the **Install** font option.

Omnis will also look for fonts installed for *all users* and registered in HKEY_LOCAL_MACHINE (HKEY_LOCAL_MACHINE\SOFTWARE\Microso NT\CurrentVersion\Fonts), but this requires administrator privileges to install and write to the registry, so this may not be appropriate for some end users, in which case they can install a font for the current user.

Monaco font (macOS)

Apple has replaced Monaco font with Menlo, therefore you should use Menlo in your reports to be compatible with current and future versions of macOS.

Unknown macOS Fonts

You can map unknown macOS fonts in reports, such as the New York legacy Mac font, to alternative fonts. You can add the “unknownMacOSFonts” item to the “pdf” section of config.json to specify the font mapping. For example:

```
"pdf": {
  "unknownMacOSFonts": {
    "New York": "Times New Roman",
    "default": "Lucida Grande"
  }
},
```

The members of the unknownMacOSFonts object are the names of the unknown fonts to be mapped, and the name of the replacement font. A “default” member can be included to map all other fonts not listed in unknownMacOSFonts to the specified font.

Port Profiles

A port profile is a named collection of information sufficient to completely describe the operating system configuration of a port. Under Windows, the port profile information corresponds to the information required to set the fields in a DCB for a serial port (under Win16 the DCB is different to Win32). There is no port profile for a parallel port, since it requires no operating system configuration information.

Under macOS, the port profile information corresponds to the information required to make the SerReset and SerHShake API calls. (Note: Port profiles are not currently supported on macOS.)

The information required to completely configure a port therefore comprises:

1. The port profile (not required for parallel ports)

2. Characters per inch.

3. Lines per inch.

Port Profile Management

Each port profile is stored in a file. The Ports folder in the Omnis tree contains the port profiles. The port profile file contains:

- An indicator of the platform to which the profile corresponds - Win16, Win32 or macOS. This allows for runtime checking.
- The name of the profile, to be used in the report destination dialog, and as an argument to the *Set port parameters* command. Note that profile names are not case sensitive.
- The profile data.

Note that this means that each profile file only contains the data for a single platform.

You can create and edit profiles using the **Port Profile Editor** on the **Add-ons** submenu in the **Tools** menu (it is not available on some older operating systems). The profile editor must run on the platform for which the profiles are to be created.

Port Profiles at Runtime

Report Destination Dialog

When the report destination is set to Port, the parameters tab of the dialog displays the port configuration information. There is a dropdown list on this tab, which contains a list of profile names, and one additional entry, ‘Use options below’. When this is selected, you can specify the port parameters in the dialog. When a profile is selected, Omnis configures the port using the information in the profile. Note that the dropdown list and configuration fields on this tab are disabled if you select a parallel port.

Set port parameters command

You can use a port profile name in place of the port parameter list:

```
Set port parameters {Profile name}
```

When this command executes, it first checks the entire parameter string against the list of profiles. If it matches an entry in the list, the command uses the profile to configure the port; otherwise, the command treats the parameter string as a parameter list.

Notation

The relevant $port… notation such as $portparity is unassignable when a port profile is selected. The $portprofile property is the port profile for the current port on the macOS, or just the single port profile under Windows.

Printer Escapes

When sending to printer, escape characters are not interpreted correctly, because the printer driver attempts to draw them as data. You must send reports containing printer escapes to port to ensure escape sequences are interpreted correctly.

The Omnis root preference $exportencoding ($root.$prefs) determines how the data is converted before Omnis sends it to the port. Set this to kUniTypeAnsiLatin1 for printer escapes to be interpreted correctly.

The escapes generated by the style() function cannot be mixed (in a report field's data) with escapes specific to a printer.

Labels

To print labels in Omnis you need to create a report class and set up its properties for label printing. You can create the report class using the standard SQL or Omnis report wizards, or you can create an entirely New Report and add the fields yourself. This section uses the Omnis Report wizard as the basis for a Customer address label, but the process is the same for any label report.

To create the basis of your label report

- Create a new report class using the Omnis Report wizard and include the fields you want in your label
- Open the report class to modify it
- Delete the header section and any labels the report wizard places on the report class, but leave the data fields; your report class should look something like the following

To change this report class into a label report you need to change some of its properties, set the properties of the Record section to position your labels on the printed page, and as a further enhancement you can set the properties of some of the fields on the report to exclude empty lines. Note that all measurements use the current units set in the **usecms** Omnis preference.

To set the label properties of a report class

- Click on the background of your report class to view its properties
- Set the **islabel** property to kTrue
- Set the **labelcount** property to the number of labels across the page; for example, for standard 3 x 8 laser labels you set **labelcount** to 3
- Specify the width of a single label in the **labelwidth** property; if there are spaces between your labels, include the space in the label width, that is, the labelwidth is the distance between one record and next across the sheet of labels
- If you want to print more than one label for each row or record of data set the **repeatfactor** property, otherwise leave it set to 1 for a single copy of each label

To specify the distance between each row of labels down the page, you change the properties of the Record section in your report class.

- Click on the Record section to view its properties
- Set the **startmode** property to kFromTop, and in the **startspacing** property enter the distance between the top of one row of labels and the next going down the label sheet

Excluding Empty Lines

When you print your labels some of the fields may be empty and a blank line is printed. However you can stop a field from printing and move up all subsequent lines by setting the **nolineifempty** property for the field. For example, if your label includes two lines for the address you can set the **nolineifempty** property to kTrue for the second address field. In this case if the second address line is empty for a particular record, the line is not printed and subsequent fields move up one line. If any address field on your label is likely to be empty you ought to set its **nolineifmpty** property.

Using a Calculated Field

Rather than putting two separate fields on your label report for the Firstname and Lastname data, you can use a single calculated field and the *con()* function.

To create a calculated field

- Create a field on your report and view its properties
- Leave the **$dataname** property empty, and set the **$calculated** property to kTrue
- Enter the calculation in the **text** property, something like the following

```
con(CU_FNAME,’ ‘,CU_LNAME) ## note space char is in quotes
```

The *con()* function concatenates the current values in the CU_FNAME and CU_LNAME fields and separates them with a single space character.

Using all the features described in this section, your label report should look something like the following when printed to the screen.

HTML Report Device

The Omnis Studio Print Manager API has been made public, allowing you to create your own custom printing devices as external components and place them in the XCOMP folder. You can show your own custom printing devices in the Print Destination dialog, and use the printing preferences and notation to control your own devices. The HTML printing device is an external component and shows what you can do with custom devices. You can use the HTML report device in the same way as the standard report destinations; there is no difference between internal and external output devices.

When the HTML external component is loaded in Omnis, it registers an external output device with the Omnis Studio Print Manager and shows the HTML icon in the Report Destination dialog. To print to the HTML output device, you can set it up via the Report Destination dialog, or access it via the notation using the print device methods.

You can specify the HTML device using the notation as follows.

```
Calculate $cdevice as kDevHtml
Calculate $cdevice as $devices.Html
```

You can also set an item reference to the HTML device:

```
Set reference myDevice to $devices.Html
```

The constant kDevHtml is supplied by the HTML component at registration together with some other constants.

Setting the HTML Device Parameters

The HTML output device has several parameters which affect the overall appearance of the HTML document generated by the device. You can change some of these parameters in the Report Destination dialog and the notation, while some can be manipulated by the notation only. The HTML output uses UTF-8: if you use a template file, that must also be UTF-8 encoded.

The HTML device parameters are represented by constants which you can use in the notation. Some of them correspond to parameters in the Report Destination dialog.

Constant	Description
kDevHtmlFileName	the pathname of the destination HTML file

Constant	Description
kDevHtmlFont1	largest point size which maps to HTML font size 1
kDevHtmlFont2	largest point size which maps to HTML font size 2
kDevHtmlFont3	largest point size which maps to HTML font size 3
kDevHtmlFont4	largest point size which maps to HTML font size 4
kDevHtmlFont5	largest point size which maps to HTML font size 5
kDevHtmlFont6	largest point size which maps to HTML font size 6
kDevHtmlFont7	largest point size which maps to HTML font size 7
kDevHtmlImageBorder	whether JPEG images have a single pixel border
kDevHtmlUseRects	whether background rectangles are to be used to determine the background color of the HTML intersects the background rectangle
kDevHtmlBackcolor	background color of the HTML document
kDevHtmlTextcolor	default text color; any black text received from the print manager will be changed to this color
kDevHtmlLinkcolor	color for HTML text or pictures which are HTML links
kDevHtmlVLinkColor	color for links which have been visited
kDevHtmlALinkColor	color for links which are currently active
kDevHtmlTemplate	full path and file name of a template HTML file which must be UTF-8 encoded; it must already c framework for an HTML file, that is <html><head> <meta http-equiv="content-type" content="te charset=UTF-8"></head> <BODY bgcolor= ...etc...> </BODY> </HTML>
kDevHtmlTemplateChars	the place holder contained within the template file which marks the point at which the report o inserted into the template, that is, "$$$$" the template file must contain this text, that is <html>< http-equiv="content-type" content="text/html; charset=UTF-8"></head> <BODY bgcolor= ...etc...> </BODY> </HTML>
kDevHtmlScaleFont...	an additional single font scale factor; the following constants are available kDevHtmScaleFontN kDevHtmScaleFontVSmall: reduce HTML size by 2 kDevHtmScaleFontSmall: reduce HTML size b kDevHtmScaleFontLarge: increase HTML size by 1 kDevHtmScaleFontVLarge: increase HTML siz

You can get and set the value of the device parameters using the following methods.

- $getparam(param constant)
  returns the value of the specified parameter
- $setparam(param constant, value [,param constant, value, ...])
  sets the value(s) of the specified parameter(s)

For example

```
Do $devices.Html.$setparam(KDevHtmlFont1,6,KDevHtmlFont2,8)
```

```
Do $devices.Html.$getparam(kDevHtmlFileName) Returns MyPath
```

The following example, sets up a template file (which must be UTF-8 encoded):

```
Calculate lPath as 'C:\<path>\Template.htm'
Do $devices.HTML.$setparam(kDevHtmlTemplate,lPath)
Do $devices.HTML.$setparam(kDevHtmlTemplateChars,'$$$$')
```

The value of all device parameters is stored in the Omnis configuration file.

Sending Text or Data

It is possible to send text or data to some internal and external devices. The HTML device supports both. You can use the methods $sendtext() and $senddata() to send text and data, respectively.

When sending text, the HTML output device surrounds the given text with the correct HTML syntax, that is, it places begin paragraph and end paragraph statements around the text. You can send text with more than one call to $sendtext(), but still have the text appear in one paragraph. To do this, specify kFalse for the line feed parameter of the $sendtext() method. The device buffers the text separately, before adding a single paragraph to the document when you call $sendtext() with the line feed parameter set to kTrue.

When sending data, the device writes the data directly to the current position in the HTML file without any modification.

You can send text or data between reports, but not during printing, that is, while a report is being printed calls to $sendtext() and $senddata() are ignored.

The following method uses $senddata() to send data to an HTML file.

```
Set reference myDevice to $devices.Html
Calculate $cdevice as myDevice
Do myDevice.$setparam(kDevHtmlFileName,"C:\Omnis\REPORT.HTM")
Do myDevice.$open() Returns ok
If ok
  Do myDevice.$senddata(myData1)
  Set report name Report1
  Print report
  Do myDevice.$senddata(myData2)
  Set report name Report2
  Print report
  Do myDevice.$senddata(myData3)
  Do myDevice.$close()
End If
```

HTML Report Objects

HTML report objects are special objects that you can use to insert objects, such as other HTML documents, pictures, DLLs, or web site addresses, into your HTML reports. The HTML report objects are part of the HTML printing device and appear in the **Media** group in the Component Store when editing a report class.

The HTML report objects have an $address property which you can set to the address or location of an HTML document, picture, DLL, or web site, for example, "results/result1.htm" or "http://www.omnis.net/". The text can contain square bracket calculations, such as "www.[lvWebName]".

If the HTML objects are printed to any other device other than the HTML device, they behave like their equivalent Omnis field types, a standard picture or text field.

The **omnisPreviewURLPrefix** item in the 'defaults' section of config.json allows you to set the report preview URL prefix for the $address property for HTML Link objects. The item defaults to 'omnis:' if empty.

Chapter 11—Window Components

Window classes and Window components are required for developing desktop or thick client applications only, and are therefore hidden in some editions of Omnis Studio, including the Community Edition. To create web or mobile apps, you need to create Remote forms using JavaScript components.

In addition to the Window class components, Menu classes and Toolbar classes are described in this chapter since they relate to desktop apps only. General techniques for managing windows and window instances, used for creating desktop applications, are described in the Window Programming chapter.

Example Apps and Code

Many of the Window Components are included in example apps under the **Samples** section in the **Hub** in the Studio Browser; use the filter to display the 'Windows' examples, or use the Search to find a specific component. You can examine the window classes and components in these example apps, and look at the code behind each component: you can double-click on a window component in design mode to see its code methods in the method editor.

Window Class Components

There are over 60 components or controls that you can use in Window Classes, for desktop or thick client applications, including standard Entry fields, Buttons, Lists, Grids, and so on. The following table provides a list of all the window class components in Omnis listed in their respective groups. (Background Objects are described at the end of this chapter.) Xcomp indicates an External component that is loaded automatically.

Group	Icon	Name (type)	Description
Buttons	⬚	Button Area	Invisable area that responds to u
	☒	Check Box	Displays On or Off choices with a mark (also Yes or No, and 1 or 0 v
		Multibutton Control (Xcomp)	A round, animated popout butto opens to show a number of add options
	OK	Push Button	Button that responds to user cli
	◉	Radio Button	Round button that can be either
	◉ Opt1 ◉ Opt2	Radio Button Group	Round buttons that can be eithe in mutually exclusive group
		Round Button (Xcomp)	Round Button showing progress individual values
	OK▾	Split Button	Standard button with a dropdov
		Switch Control (Xcomp)	iOS style switch with animated s
		Trans Button (Xcomp)	A 'rollover' type button
Containers	XY	Group Box	Groups other fields on your winc
		Paged Pane	Multiple pages or panes contain and other controls
		Scroll Box	Group other fields in a scrollable
		Tab Pane	Multiple pages or panes with tal
Entry Fields	###	Masked Entry Field	Entry field with 'mask' to format
	Abc Xyz	Multi Line Entry Field	Entry field allowing multiple line scroll bars
	Ab|	Single Line Entry Field	Field into which users can insert view existing data
	Ab ☒	Token Entry Field	Field which tokenizes entered te
Graphs	◔	Graph2	Graph component with multiple types
Labeled Fields	Ab|	Labeled Fields	Fields and label combined into c
Labels	T	Labels and text objects	See Background Objects
Lists	☑	Check List	List with check box line selectior
		Combo Box	Combined dropdown list and er
		Complex Grid	Grid that can contain other field controls
		Data Grid	Grid to display text and numeric
		Droplist	Single column dropdown list
		Headed List Box	List with button style headers
		Icon Array	Displays list of items as clickable
		List Box	Displays list variable contents
		Popup List	Single-column list allowing easy selection
		String Grid	Grid to display character data or

Group	Icon	Name (type)	Description
		Tree List	Displays list data in an expandab hierarchy
Media	JPG	JPEG Control (Xcomp)	Displays JPEG images
	🌐	OBrowser (Xcomp)	Embed a web page into a windo enables HTML controls
	ICON	OmnisIcn Control (Xcomp)	Displays an icon from an Omnis file
		Picture Control	Displays image data from a pictu variable
	▶	Video Player (Xcomp)	Plays video file from disk or rem
		WAV Player (Xcomp)	Plays a WAV sound file
Menus		Popup Menu	Displays a menu class on a wind
Navigation		Accordion Control (Xcomp)	List of expandable options
		Breadcrumb Control	Displays “location” within the hie an application
		FishEye Control (Xcomp)	Displays a row or column of click icons
	LINK LINK LINK	Hyperlink Control (Xcomp)	List of hyperlink options
		Navigation Menu (Xcomp)	Cascading menu with images an options
		Sidebar Control (Xcomp)	Displays a list of options with gro
		Tab Strip	Set of tabs only that can be linke paged pane
Other	▦	Calendar Control (Xcomp)	Presents dates in a standard cale format
	🕒	Clock Control (Xcomp)	Clock face showing the current t
		Color Palette	Color picker allowing color selec
	TEXT	Marquee Control (Xcomp)	Displays scrolling text
		Progress Bar (Xcomp)	Indicates progress of a counter
		Slider Control (Xcomp)	Draggable button to set a value
		Transform Control (Xcomp)	Adds animation or effects to wir objects
Reports	🖨	Modify Report Field	Embeds a report class in a wind
		Screen Report Field	Displays the output of a report
Shapes	Abc	Shape Field	Shape with some field propertie other shapes see Background O
Subwindows		Subwindow	Embeds another window in the window

Some external components have been deprecated and have been moved the ‘Deprecated Components’ group in the Component Store. See Deprecated Components.

There are also a number of ‘Internal’ window components that should not be used for new applications, including HelpMethods and Icon Edit; they can be shown using the **Exclude Group** option, available by right-clicking on the Component Store.

Loading External Components

Most external components are pre-loaded while other may need to be loaded manually to be visible in the Component Store (in the standard or deprecated groups). To load an external component, Right-click/Ctrl-click on the Component Store and select the **External Components…** option (or double-click the #EXTCOMPLIBS system class in the Studio Browser) to open the External Components dialog. Expand the External Components group (not the JavaScript group), find the component (library) you want to enable and select it.

Change the 'Preload status' for the component to 'Opening [lib-name]' to load the component for the current library, or select 'Starting Omnis' if you want the component to load for all libraries. You only need to load or enable the components you intend to use: loading any components that are not used places an unnecessary load on Omnis.

When loaded manually, the external component is included in the relevant group in the Component Store, e.g. the Accordion component is an external component and is added to the 'Navigation' group, or the external component may be added to the 'Other' group.

Window Class Object Limit

You cannot place an unlimited number of objects on a Window class. The object limit is **8191** for a Window class, including objects on subforms, although in practice the limit is likely to be less due to platform limitations.

Object Properties

All fields and components have General properties that control the overall behavior and appearance of the field which you can view and change in the Property Manager. In addition, each type of field or component has its own set of properties and methods that provide its unique functionality. All fields have a particular size and position, stored in the **left, top, width,** and **height** ($left, $top, $width, $height) properties, which are displayed at the top of the Property Manager.

External components have many of the standard properties, together with its own specific properties which are generally shown under the Custom tab in the Property Manager.

Object Names

All window fields have a **name** ($name) property which is displayed at the top of the Property Manager. When you add a component to a window, Omnis assigns the component a default name with the format 'classname_comptype_N' where N is a four digit integer unique to the class. For example, an entry field in a window class might be named 'wMyWindow_entry_1008'. You can accept the default name assigned to a component or change it to a more descriptive name for your application.

To change the name of an object, click on the name or the pencil icon and change the name. There are no restrictions on the name of an object, although you are advised to use only alphanumeric characters *and avoid using spaces*. Do not use the same name for multiple objects since this will cause confusion or errors when you refer to the object in the notation. For this reason, you should use unique names for fields and objects within the same class.

The **object type** is shown below the object name in the Property Manager, e.g. Entry or Pushbutton. You cannot change the type of an object. The $objtype property is displayed in Advanced mode in the Property Manager, e.g. kEntry or kPushbutton, but it cannot be changed.

Object datanames

All data-bound fields and components have a Dataname ($dataname) property, which is displayed in the top panel of the Property Manager. For some list objects, $listname is shown which is the name of the data variable associated with the list-based window object. When you create a class using an Omnis wizard, the $dataname of each field created automatically in the class is assigned a variable of the appropriate type. When you create a window field from the Component Store you need to assign the $dataname property manually.

You can create variables in the Variable panel of the Method Editor (click on Methods in the Design bar at the top of a window to open the Method editor), or you can type the name of a variable into the Dataname field in the Property Manager, press the Return key and define the variable in the 'Create Variable' dialog. See Variables for more information about declaring and naming variables.

Component Icons

Several of the window class components can use icons to enhance their visual appearance, such as the Pushbutton, Sidebar, and Icon array. These icons are added to a component by specifying the icon name or ID in the **$iconid** property for the control. The icon size can be specified using a size constant, such as k16x16, k32x32, or k48x48, or you can append +wxh to the icon name or ID to specify a custom size, e.g. help+64x64.

Window component icons can be SVG image files selected from the dialog that opens when you click on the $iconid property in the Property Manager (or for older apps you can use PNG files). The 'material' iconset is selected by default when the Select Icon dialog opens, which contains over 100 icons that you can use in your own apps. The icons in the material iconset are sourced from the Google material design set, and they have been themed using the Omnis SVG Themer tool (so they support JS and system themes). The Google icons are issued under the Apache License Version 2.0 (https://fonts.google.com/icons), and you are free to use these in your Omnis applications with the proper attribution in your product licensing.

If you use your own SVG icons, they should be placed in a named **iconset** folder inside the 'iconsets' folder in the Omnis tree. The name of the iconset needs to be assigned to the **$iconsets** property of the current library. See Selecting an Icon for more details about specifying icons, SVG icons and iconsets.

For legacy apps only, PNG icons can be located in the #ICONS system table in the current library, the Userpic icon datafile, or the Omnispic datafile. You will need to use the Icon Editor to add or edit the icons in #ICONS or an icon data file.

Themed SVG Icons

From Studio 11 onwards, you can use themed SVG icons that have been "themed" using the Omnis SVG Themer tool (available in the Tools>>Add Ons menu). The 'material' icon set in Omnis contains themed SVG icons and is available automatically when you edit a Window class.

A themed SVG icon will use the color set in the **$textcolor** property of the window class control, so it matches the color of the text for the control. For Styled text, a themed SVG is drawn using the current text color for the text run.

Some external components support themed SVG icons. The Multibutton, Round button and Tile external components have the $textcolor property, plus the HTML icon link control has the $::textcolor property. The color specified in these properties will be applied to a themed SVG icon.

Drag Icon background

As a consequence of support for themed SVG icons for window classes, drag icons have a background by default on macOS, to prevent themed SVGs from becoming invisible, and make drag icons more cross-platform. You can turn off this behavior using the **dragIconBackground** item in the 'macOS' section of config.json (default is true, to show the icon background).

Dark and Light Modes

You can specify different SVG icons for Dark and Light modes (this is only intended for running desktop apps on Windows or macOS, since different system color modes do not apply to web and mobile apps running in a web browser).

Each icon set folder can have two sub-folders, named **dark** and **light,** into which SVG icon files can be placed to support Dark and Light system color modes.

When you assign an icon to a control you only need to assign a single icon name or ID and the icon for Dark or Light mode will be chosen automatically from the appropriate sub-folder.

Vertically Centered Text

The $vertcentertext property controls whether or not text is centered vertically in the field area. The property makes it easier to produce well aligned text and fields across all the platforms supported in Omnis. Using this property in your windows will allow you to line up the base-line of text labels and the text data contained in fields.

- **$vertcentertext**
  If true, single line text is vertically centered in the height of the field. If false, the text is vertically positioned according to the default positioning for the field. For existing fields the property is set to kFalse.

The $vertcentertext property is available for several window field types, including single line edit fields, combo boxes, droplists, background labels, background text objects, string labels, shape fields (the text part), checkboxes (no border), radio buttons (no border), masked entry fields. The new property also applies to several remote form fields, including single line edit fields, combo boxes, droplists, background labels, string labels, checkboxes (no border), and masked entry fields. Note the property is not available for multi-line fields on windows.

Font Scaling for Fields

You can increase or decrease the font size of the Multi-line Entry field, String grid and Data grid using the key press Ctrl + or Ctrl -.

The $disablefontsizekeys property lets you control font scaling for these controls, together with the standard List, Checkbox list, Headed list and Tree list which already respond to the Ctrl +/- key press to scale the font. The default value of $disablefontsizekeys is kFalse, which means the control will respond to the Ctrl +/- key press to adjust its font size; set the property to kTrue to disable font scaling.

Event & Control Methods

$sendevent method

Window objects and window instances have the $sendevent method which allows you to test your $event/$control methods for window fields (or window instances).

- **$sendevent**(iEvent[,eventParameters…])
  Sends event iEvent (an ev… constant value) to the object with eventParameters passed as name,value pairs, for example $sendevent(evClick,‘pLineNumber’,2). Returns kFalse if the event is discarded; generates a debug error if there is a problem with the parameters.

You can also pass #SHIFT, #CTRL/#COMMAND, #ALT/#OPTION as “event parameter” names, to set the value of these variables when the event is being executed.

When entering a $sendevent method, typing Ctrl+Space after the quote lists the event parameter names.

Note that the invoked $event/$control will execute, but if there are requirements of the data associated with the control, you need to separately code for that - $sendevent simply sends the event causing $event or $control to execute appropriately.

Alpha Colors & Transparency

Some of the Window class controls support **alpha colors,** meaning that you can set the transparency for the color of the control. The color picker in the Property Manager displays an alpha selection slider if a selected control supports alpha colors (the alpha slider is hidden for controls that do not support alpha).

The controls that support alpha colors include the Line, Oval, Rect and RoundRect background objects for windows.

The ***rgba()*** function can be used to set the RGB color and alpha setting for controls. The syntax is rgba(red,green,blue,alpha) with each parameter being an integer value in the range 0-255, where an alpha value of 255 means completely transparent. For example, to set the color of a window background object, in this case 50% transparent red:

```
Calculate $cwind.$bobjs.1016.$forecolor as rgba(255,0,0,127)
```

In addition, the color selection palette for the controls that support the use of a color palette or a popup color palette, including the colorpalette control, push buttons, and toolbars, include the alpha selection slider.

When assigned a color with no alpha the palette will automatically hide the alpha slider. If the control is assigned an alpha value, the palette will display the alpha slider.

For example, where **colorbutton** is a push button with $buttonmode set as kBMcolorpicker:

```
Do $cwind.$objs.colorbutton.$contents.$assign(rgb(255,0,0))  ## will cause the color palette not not show a
Do $cwind.$objs.colorbutton.$contents.$assign(rgb(255,0,0,127))  ## will cause the color palette to show an
```

Omnis external components can support alpha colors. The Export & Import Library to JSON options also support alpha color values for controls.

Container Fields

Some of the complex field types are described as *container* fields. A container field is simply a window field that contains other fields. These include tab and paged panes, complex grids, group boxes, scroll boxes, and subwindows. All container fields except subwindows have the $objs and $bobjs object groups containing the fields and background objects within the container field. Therefore, in the notation you access the objects within a container field via these object groups. For example, the notation for a field called MyField inside a paged pane is

```
$iwindows.WindowName.$objs.PagedPane.$objs.MyField
```

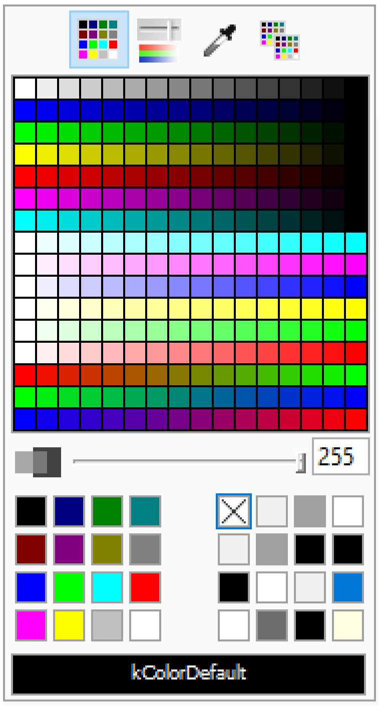


Figure 140:

$container

Every field within a container field has the $container property which is an item reference to the container field to which the object belongs, for example

```
# Code in $event for MyField
Set reference iItem to $cobj.$container
# sets iItem to $root.$iwindows.WindowName.$objs.PagedPane
```

You can nest container fields 999 levels deep. Beyond this level, the most deeply nested field or object is not set up when the window is opened and becomes a display field showing an error message.

$container returns the window instance for controls (including subwindows) at the top level: in versions prior to Studio 10 this was not available for window class controls, only JavaScript controls. For example, you can use $cinst.$container() to refer to the outer window instance when executed in a subform window.

If you have a loop in your code that steps through from a window class control up the container hierarchy the final container will be the window, so you will need to test if the container is a window, e.g. If itemref.$container().$ref.$classtype=kWindow, then Break to end of loop.

$objlink

The $objlink property is an integer that contains information about the container of the object. You can assign $objlink in your code using class notation, provided that *the design window is not open*. So for example, you can move an existing control in a window class into a Group Box in the same window class using code.

Object Animation

There is a library and window class control property, $animateui, that allows you to animate some window controls, including the **Tree List,** plus the **Tab Strip** has some display types to highlight and animate the tabs when they are selected. The property is defined as:

- **$animateui**
  If the library property $animateui is true, all objects that support $animateui will animate aspects of their interface. Therefore, the object property only applies when the library property is false.

If the $animateui library property is false (shown on the Appearance tab in the Property Manager), the $animateui property for the individual object is used. Therefore, if you only want *some of the controls* in your library to animate, set the $animateui library property to false, and override at the object level by setting $animateui for the object to kTrue.

Moving Objects

Window instances have the methods $beginanimations() and $commitanimations() which allow you to animate changes to certain properties of some window components including the $alpha property: other properties supported are $left, $top, $width and $height.

- **$beginanimations**(iDuration[,iCurve=kAnimationCurveEaseInOut])
  after calling this, assignments to some properties are animated by $commitanimations() for iDuration (in milliseconds) and using the specified animation curve (kAnimationCurveEaseInOut is the default)
- **$commitanimations()**
  animates the relevant property changes that have occurred after the matching call to $beginanimations()

For example, you could move a component into view by animating a change to its position via its $left and/or $top properties.

```
Do $cinst.$beginanimations(1000, kAnimationCurveLinear)
Calculate $cinst.$objs.button.$left as currentposition ## var set to required position
Do $cinst.$commitanimations()
```

If you set the same property for an object more than once, the first property change is animated, and then the last property change is animated when the first completes, while property changes between the first and last are ignored.

The iCurve can be one of the following animation “easing” curves:

- **kAnimationCurveEaseIn**
  The animation begins slowly and then speeds up as it progresses.
- **kAnimationCurveEaseInBack**
  The animation is similar to kAnimationCurveEaseIn but first moves in the opposite direction before easing begins.
- **kAnimationCurveEaseInOut**
  The animation begins slowly, accelerates through the middle of its duration, and then slows again before completing.
- **kAnimationCurveEaseOut**
  The animation begins quickly and then slows down as it progresses.
- **kAnimationCurveEaseOutBack**
  The animation is similar to kAnimationCurveEaseOut but moves beyond the final point before easing back to the final location.
- **kAnimationCurveEaseOutBounce**
  The animation starts slowly and then bounces on its final location.
- **kAnimationCurveEaseOutElastic**
  The animation starts fast and springs to a stop around its final location.
- **kAnimationCurveLinear**
  The animation occurs evenly over its duration.

The **evAnimationsComplete** event is generated after the last property change has completed, which allows you initiate a further animation or another action, or reverse the changes you have made.

Rounded Borders

You can apply rounded borders to most Window class UI controls by setting the $borderradius property.

In general, $borderradius only applies when $effect is kBorderPlain, kBorderCtrlEdit or kBorderCtrlList. For Pushbuttons, Radio buttons, and Check boxes, $borderradius only applies when $buttonstyle is set to kUserButton.

Object Transparency

Most window components have the $alpha property which means you can set the transparency of the component (an integer from 0 to 255, with 0 being completely transparent and 255 opaque). The $alpha property for a component can be manipulated using the $beginanimations() and $commitanimations() methods so you could “fade in” and “fade out” objects by setting the alpha from 0 to 255 using animation.

Tooltips

Tooltips are short messages that pop up when the end user passes the pointer over a field or control to provide some help or information about the object. The tooltip text is entered in the **$tooltip** property for the object. Entry fields can also have Content tips.

You can control the background and text colors, the justification of text and the position of the tooltip relative to the component. The toolip properties are in the 'tooltip' section of 'appearance.json', as follows:

- systemstyle
  If true (the default), tooltips in the system style are drawn using colorinfobk and colorinfotext; if false, Omnis style tooltips are drawn using tooltipbackgroundcolor and tooltiptextcolor
  Omnis style tooltips have rounded corners and (unless it is not relevant for the particular tooltip) a small pointer
- tooltipbackgroundcolor
  The background color used for Omnis style tooltips
- tooltiptextcolor
  The text color used for Omnis style tooltips
- defaultjustification
  The default justification of the tooltip rectangle relative to the active area of the component: 0 left (the default), 1 right, 2 center; ignored by certain controls if applying the default does not make sense

The **$tooltippos** property controls the position of the tooltip relative to the control:

- $tooltippos
  A kTooltipPos… constant that specifies where $tooltip appears relative to the window control

Constant	Description
kTooltipPosMouse	$tooltip appears relative to the mouse pointer (this is the default and corresponds to 10.1 behavior)
kTooltipPosRight	$tooltip appears to the right of the window object
kTooltipPosBottom	$tooltip appears below the window object
kTooltipPosLeft	$tooltip appears to the left of the window object
kTooltipPosTop	$tooltip appears above the window object

HTML Components for Desktop Applications

You can add your own custom HTML components to your window classes, which are used in the thick client, that is, for desktop based applications. (*Note this feature does not relate to the JavaScript components for creating web and mobile apps* – this feature refers to using HTML based controls on *window classes.*)

By adding HTML controls to your window classes you can enhance the UI in your desktop applications and accelerate your development projects – you can obtain many different types of ready-made HTML based components from third-party sources, from simple data controls, to date selectors, to full gantt charts, with the richness and interactivity you would expect to see in web-based applications.

Omnis HTML controls can be thought of as thick client external components implemented using HTML, JavaScript and CSS – in effect, you can use any browser based technology to implement this type of HTML controls. To add an HTML based control to a window class you need to use the **OBrowser** window control, which can, in this case, be used to display a single HTML based control in a window class (the OBrowser object can also act as a regular browser for displaying web pages, which is described under OBrowser). The OBrowser object is in the Media group in the Component Store.

The Omnis HTML controls themselves are located in the **'htmlcontrols'** folder in the main Omnis Studio program folder. Each control has its own sub-folder in the htmlcontrols folder, and the name of this folder is used as the name of the control, e.g. the files for a control named List would be placed in a folder called List. The Omnis tree contains some example HTML controls which you can use for testing, or as a basis for creating your own custom controls. These examples, such as the Quill component which provides a basic text editor, are not supported controls in their own right, so we don't recommend using them as-is in your applications.

To add an HTML control to a window class, you need to add the OBrowser object to the window, which is available in the **Media** group in the Component Store, and set its **$urlorcontrolname** property to the name of the control – this property displays a droplist containing the names of all the available controls installed into the htmlcontrols folder in your Omnis development tree, including any you have added. Having added the control to your window you can set its properties in the same as way as any other Omnis component. See the Adding HTML controls to your window later in this section.

Using HTTP for controls

The $htmlcontrolsusehttp property controls whether html controls are served using a file:// URL (when set to kFalse), or using the built-in HTTP server (kTrue). File URLs are not deemed secure, so some web APIs in HTML controls may not be available. Therefore, it is recommended that you set $htmlcontrolsusehttp to true. When true, html controls are loaded from the 'html/htmlcontrols' folder in Omnis.

The config options "htmlcontrolsFolder" and "defaultHtmlcontrolsFolderInDataFolder" only apply when $htmlcontrolsusehttp is false.

Third-party HTML controls

As well as creating your own controls using HTML, JavaScript, and CSS, you can obtain many ready-made controls from third-party sources, either on an open source or paid-for basis. The HTML code for a control needs to be embedded into the Omnis compatible HTML template which is required to load a control into the OBrowser object in Omnis. You can find many very useful HTML or JavaScript based controls on GitHub (https://github.com/), plus there are a number of example HTML controls in the Omnis Studio GitHub repository: https://github.com/OmnisStudio/

JavaScript Client Bridge

The **JavaScript Client Bridge** (JSCBridge) is an HTML control that allows you to run the Omnis JavaScript Client within OBrowser in a standard Window Class, which means you can open a Remote Form in the desktop (fat client) version of Omnis Studio, passing data between the form and Omnis.

The source code and documentation for the JSCBridge control are available on the Omnis Studio GitHub repository: https://github.com/OmnisStudio/Omnis-JSCBridge

Creating Omnis HTML Controls

Each Omnis HTML control can be comprised of a number of files which are placed in a folder in the 'htmlcontrols' folder in the Omnis program folder. The main file for each control is an HTML file which is named <control name>.htm. For example, if you want to create a control called "quill" you need to create an HTML file called 'quill.htm' which is placed in a folder named 'quill' within the 'htmlcontrols' folder. The .htm file is the file loaded into the browser control (set using $urlorcontrolname) when the control is used on your window.

In addition, there may be a JSON file named <control name>.json in the control's folder which defines the htmlcontroloptions row. Plus, the control folder can contain other resources needed as part of the control implementation, such as JavaScript files, CSS files, image files, and so on. The control .htm file typically has links to these other resources.

When you have added the correct files to the relevant folder the control will be ready to use and add to the window classes in your application. To deploy your application, you will need to add the same files and folder structure to the Omnis runtime tree.

<control name>.htm

The .htm file for a control defines a jOmnis object, its various callbacks, and the HTML content for the control itself, embedded at the place marked "...control-specific contents..." in the HTML code. The file has the following structure:

```
<html>
  <head>
    <title></title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <script type="text/javascript" src="../omnishtmlcontrol.js"></script>
    <script>
      jOmnis.callbackObject = {
        omnisOnLoad: function () {},
        omnisSetOptions: function(options){},
        omnisCssChanged:function(){},
        omnisSetData: function (value) {},
        omnisGetData: function () {},
        omnisGetCurrentLine: function(){},
        omnisGetSelection: function(){},
        omnisSetFocus: function () {},
        omnisTab: function() {},
        omnisGetDraggedData: function (event) {},
        omnisDropHilite: function (hiliteLine, destinationId) {},
        omnisDropUnhilite: function () {},
        omnisGetDropLine: function (mouseX, mouseY) {},
        omnisDoScroll: function(scrollDirection, scrollAmount){},
        omnisOnWebSocketOpened: function() {},
        omnisOnWebSocketClosed:function() {}
      };
    </script>
  </head>
  <body style="margin:0;" class="omnishtml">
    …control-specific contents…
  </body>
</html>
```

Note that the omnishtml class could be applied to another element rather than the body.

The **omnishtmlcontrol.js** JavaScript file is used at the start of the page, and provides an interface to various objects in Omnis, including:

- Provides the interface between JavaScript and the OBrowser component.
- Provides an object that allows you to manipulate an Omnis list represented in JSON.
- Ensures that HTML controls can use Date variables as their $dataname.

Existing users should note that pre-Studio 10.2 versions had 3 JS files that are now combined into 1 file: the omn_list_base.js and omn_date.js files have been bundled with omnishtmlcontrol.js so you only need to reference that one file. Existing HTML controls should work as the bundle is named omnishtmlcontrol.js, which is also now minified.

<control name>.json

The JSON <control name>.json file defines the options and properties of your HTML control. The control's JSON document has the following structure:

```
{
  "data": "multi",
  "options": {
    "selectedcolor": 255,
    "margin": 0,
    "mainiconid": 1710,
    "title": "The title",
  },
  "optionsDescriptions": {
    "selectedcolor": "Description for selectedcolor option",
    "margin": "Description for margin option",
    "mainiconid": "Description for main icon id",
    "title": "Description for title"
  }
}
```

The top-level object has members as follows:

Member	Description
data	This member indicates how data is exchanged between OBrowser and the JavaScript running in the HTML control. It can have three possible values: all: This means that when OBrowser wants to get the current value of the control from JavaScript, it will retrieve all of the data. NOTE that all is the default data handling mechanism if the control does not have a .json file. single: Applicable to controls for which $dataname is a list. When OBrowser wants to get the current value of the control from JavaScript, it will retrieve just the current line. This is useful for single select lists that do not modify the data. multi: Applicable to controls for which $dataname is a list. When OBrowser wants to get the current value of the control from JavaScript, it will retrieve the current selection state and the current line. This is useful for multiple select lists that do not modify the data. single and multi provide optimized data handling for lists that do not modify the data.
option	Each member of options is an option that can be used to modify the behaviour of the control. The options object defines the members and their default values. Each member must be a simple type (number, boolean or string). The initial value of $htmlcontroloptions for a new control is the value of this object. If you edit the json file to include new options, OBrowser should detect them and update $htmlcontroloptions. When OBrowser sends the options to JavaScript it sends the current value of $htmlcontroloptions. If the option name ends in “color” then the value stored in the options object is an integer RGB value. When OBrowser sends this to JavaScript it converts it to a CSS color value. If the option name ends in “iconid” then the value stored in the options object is a valid icon id from an iconset. When OBrowser sends this to JavaScript it converts it to a file URL. When using an icon from #ICONS, the file URL uses the same PNG file as that used for the JavaScript client. You will need to manually delete the PNG in the html/icons folder if you edit the icon, to allow Omnis to re-generate the file.
optionsDescriptions	There should be a member in this object for each member of options. The values are used as tooltips when editing $htmlcontroloptions using the property manager.

The Callback Object

omnishtmlcontrol.js creates an instance of an omnishtmlcontrol object called jOmnis. Your page sets the callbackObject member of jOmnis, to allow the omnishtmlcontrol object to communicate with your control implementation.

Your callback object can contain its own members, but do not use ‘omnis’ as the prefix of a member name, since we use omnis to identify methods provided by the callback object that may be called by omnishtmlcontrol.

Methods called using $callmethod are members of the callback object.

The following table describes the omnis-prefixed methods you need to provide in a callback object. Each member is marked as mandatory or optional to indicate if you must provide an implementation.

Method	Description
omnisOnLoad	Mandatory. Called when the browser onLoad event occurs. Use this to perform initialization.

Method	Description
omnisSetOptions	Optional. Called to set the options for the control. It has a single argument, which is a JavaScript object containing the members defined in the control .json file. When first loading the control, omnisSetOptions is called before CSS is applied and before set data. Once the control is loaded, omnisSetOptions can be called again if the application assigns $htmlcontroloptions.
omnisCssChanged	Optional. Called after the CSS in the omnishtml class has been added to the page, or updated. When first loading the control, omnisCssChanged occurs after omnisSetOptions, but before set data. Once the control is loaded, omnisCssChanged can be called whenever a property that contributes to the omnishtml class is changed.
omnisSetData	Mandatory. Called to set the data for the control. One parameter, the data.If the data is a row variable, then the parameter is a JavaScript object with a member for each row column; the data types of the members must be simple types (character, boolean, integer, number). If the data is a list, then the parameter is an instance of omnis_list.Otherwise, the data is a value of a simple type. Note that for character data, OBrowser converts Omnis line endings (\r) to suitable line endings for the HTML control (\n) before calling omnisSetData.
omnisGetData	Mandatory for controls which have a data mode of “all”. Called to get the data from the control. Returns the data for the control.If the data is a row variable, returns a JavaScript object. It must have the same definition as the originally set row. If the data is a list, the return value must be an instance of omnis_list.Otherwise the return value must be a simple type. Note that for character data, OBrowser converts browser line endings (\n) to Omnis line endings (\r) in the returned data.
omnisGetCurrentLine	Mandatory for controls which have a data mode of “single” or “multi”. Returns the current list line.
omnisGetSelection	Mandatory for controls which have a data mode of “multi”.Called to get the list selection from the control. Returns an array of integers, with a member for each list line. A member is zero if the line is not selected, one if the line is selected. The array entry for line 1 is at array index zero.
omnisSetFocus	Optional. Called when the control receives the focus. You may need to focus an element when this method is called e.g. this.elem.focus().

Method	Description
omnisTab	Optional. Passed a single parameter, the JavaScript keydown event. Called when a tab occurs. This gives the control the opportunity to tell Omnis to tab out of the control.omnishtmlcontrol provides default behaviour based on HTML tabindexes - you can override the default by providing omnisTab. To tell Omnis to tab out of the control, omnisTab calls jOmnis.tabOutOfControl with a single Boolean argument which is true to perform a shift tab.
omnisGetDraggedData	Optional. If your control supports drag data, you provide this callback to let Omnis obtain the data being dragged. The return value is the dragged data, or null if nothing can be dragged.You can either return text, or a list, or a row.There are helper methods in jOmnis: makeDraggedDataList and makeDraggedDataRow to assist with the latter two return types.
omnisDropHilite	Optional. Called to highlight the control when it is a possible drop destination during drag and drop.Two parameters: hiliteLine, destinationIdhiliteLine is Boolean, true if the $hiliteline property is set for the control.destinationId is the drop destination id, typically a line number returned by omnisGetDropLine if highlighting lines is supported.There is a helper method in jOmnis, that can be used to highlight the entire control. For example:jOmnis.appendDefaultHiliteDiv(document.body);appendDefaultHiliteD returns the appended div, so you can remove it from the DOM when unhighlighting.
omnisDropUnhilite	Optional. Called to remove drop highlighting from the control. No parameters.
omnisGetDropLine	Optional. Called when $hiliteline is true, to determine the line over which the pointer is positioned. The return value is the line number (destination id).It takes 2 arguments: mouseX, mouseY These are the current pixel coordinates of the pointer.
omnisDoScroll	Optional. Called to scroll the control while the pointer is over its edges during drag and drop of data.It takes 2 arguments: scrollDirection, scrollAmount scrollDirection is an eScrollDirections value (see omnishtmlcontrol.js) that identifies the direction to scroll. scrollAmount is the maximum number of pixels by which to scroll. Scroll by this amount, if scrolling is desired.
omnisOnWebSocketOpened	Optional, and not normally needed. Called when the socket between OBrowser and the HTML control opens. From Studio 8.1.6 the method receives the Web Socket port as a parameter.
omnisOnWebSocketClosed	Optional, and not normally needed. Called when the socket between OBrowser and the HTML control closes.

Sending Events

jOmnis contains APIs that allow you to send events to $event:

API	Description
sendClickEvent	jOmnis.sendClickEvent(lineNumber)Generates evClick with pLineNumber set to lineNumber.
sendDoubleClickEvent	jOmnis.sendDoubleClickEvent(lineNumber)Generates evDoubleClick with pLineNumber set to lineNumber.
sendControlEvent	jOmnis.sendControlEvent(infoObject)Generates evControlEvent with pInfo set to the row corresponding to infoObject.

HTML controls can pass dates when calling sendControlEvent(), either directly, or as a column in a row/JS object. However, due to issues sending messages to Omnis including Omnis dates inside lists/rows, any object members whose names begin "__" (double underscore) are stripped out before sending to Omnis.

In addition, HTML controls can pass nested rows/JS objects with sendControlEvent().

Development Mode

If you need to alter the behavior of your HTML control in some way in development mode, you can check the value of jOmnis.mDesign which will be true in development mode.

Debugging

You can edit config.json (see the later configuration section) to enable debugging of your control.

On Windows, once you have set a remote debugging port, open Chrome and navigate to http://127.0.0.1:nnnn where nnnn is the remote debugging port.

On macOS, right click on the control and select inspect element. Note that the web inspector window that opens does not work that well with our window ordering.

Drag Object Support

Due to the way pointer events work with the control, when you enable drag object or drag duplicate, Omnis displays a small bar at the top of the control to enable it to be dragged. There is a property, $dragobjectbarcolor that you can use to set the color of this bar.

Reloading HTML Controls

You can use the $reload() method with an Omnis HTML control to reset it to its initial state. $reload() also automatically redraws the control, so its data will also be set. Using $reload like this is useful when debugging your JavaScript.

Tooltips

The tooltip property for an Omnis HTML control is applied once when the control is created, and it is not re-evaluated. If you want to change the tooltip after creation, you must use $callmethod to provide an interface to change a title attribute in the HTML.

Adding HTML controls to your window

Having created or obtained an HTML control and placed it in the htmlcontrols folder, you can use it in a window class in your library. There are also a number of example controls for you to use as well, such a simple List and Quill, a basic text editing control. Once the control is placed on your window it can be used and updated in the same way as any other control.

Locate the **OBrowser* object in the** Media** group in the Components Store and add one to your window class. Open the Property Manager and set the $urlorcontrolname property to the name of the control – the droplist for this property contains the names of all the available controls installed into the htmlcontrols folder.

For HTML controls located in the 'html/htmlcontrols' folder in Omnis, you will need to set the $htmlcontrolsusehttp OBrowser property to kTrue, after which the HTML controls will be listed in the $urlorcontrolname property.

There is a version of the JS Markdown Object that is implemented as an HTML control which you can use in a window class; see Markdown Object.

Html Control Properties

When you select a control its properties will be displayed in the Property Manager, including the following properties.

$dataname

An Omnis HTML control can be data bound. The dataname can be a list, a row, a date, or any other simple non-binary type. This makes the control behave like any other data bound control, with one small exception that improves performance. Omnis does not redraw the control when it gets the focus. The only real consequence of this is that you need to explicitly call $redraw in order to update the control.

$disabledefaultcontextmenu

The underlying browser has its own context menus e.g. a TEXTAREA with spell checking enabled has clipboard menu items, as well as spelling suggestions etc. The underlying browser menu is considered the default context menu, and you can disable this using the $disabledefaultcontextmenu property.

$htmlcontroloptions

An Omnis HTML control may have a row of options that can be used to configure its behavior. You can consider these to be custom properties. The Property Manager has a droplist button for this property, which opens the editor for these options. Options with names ending in color or iconid are edited using the color picker or select icon dialog respectively. The fixed column at the left of the editor has tooltips that display descriptions for the members of the options row.

If the control does not have any options, this property is read-only.

$applycss and $cssextra

You can optionally apply a CSS class named omnishtml to the Omnis HTML control (note that the control needs to explicitly use omnishtml - if it does not, then apply CSS will not have any affect). Set $applycss to kTrue if you want to use the omnishtml class.

The omnishtml class contains entries for various other properties of the browser object (OBrowser): $backcolor, $backalpha, $textcolor, $align, $fontsize, $fontstyle, $font. $font uses the JavaScript client font table entry corresponding to the window font.

In addition, OBrowser also concatenates the value of $cssextra to the end of the omnishtml class e.g. you could set $cssextra to “text-decoration:line-through;text-transform: uppercase;”.

$dragmode

You can set $dragmode to kDragData to enable drag from the control. This only works if the particular control has been designed to support drag data. Drag and drop uses the standard Omnis drag and drop messages.

$hiliteline

List controls that accept dropped data can be configured to highlight individual lines during drag and drop. You can set this property to true, to indicate that you want highlight line behavior, but you will only get that behavior if the Omnis HTML control currently being used supports it.

In addition, when evDrop occurs for a control with $hiliteline set to true, the pDropId event parameter identifies the area of the control over which the drop is to occur, either a line number or ident (when $hiliteline is true), or zero if the control is not list-based (or $hiliteline is false).

$callmethod()

Omnis HTML controls can have methods. You can use the $callmethod() method to call a method within a control or one of the standard callbacks.

- $callmethod(cName,vParam)
  Calls method cName in the control object, passing parameter vParam; returns a unique id for this call. The method runs asynchronously and sends the evCallMethodDone event to the control on completion (see Events below)

```
# method gets the data from the example Quill control
Do $cinst.$objs.quill.$callmethod('omnisGetData') Returns iID

# event method for Quill control assigns the data returned to a var
```

```
On evCallMethodDone
  If pUniqueId=iID
    Calculate iData as pReturn
    # Do something
  End If
```

Markdown Object

There is a version of the JS Markdown Object that you can use in a window class as an HTML control (the source files for the control are located in the 'html\htmlcontrols' folder); see the Markdown Object JS component for more details about creating markdown content and setting its properties.

To use the Markdown Object in a window class, add an **OBrowser** control to your window, set its **$htmlcontrolsusehttp** property to kTrue, then select 'markdown' in the **$urlorcontrolname** property.

The markdown content can be built up in the same way as for the JS control and assigned to an instance variable specified in $dataname of the Markdown HTML control.

The color settings and so on for the markdown can be set in the **$htmlcontroloptions** property, for example, textcolor and imagemaxwidth, which are equivalent to the properties for the JS control.

Ports

The OBrowser component operates on the same port as the Omnis Server which is either assigned dynamically or via the Omnisserver property. For debugging HTML controls, OBrowser opens another port for WebSocket communications between htmlcontrols and Omnis. This is $serverport + 1, or 6912 if $serverport not set. This can be overridden by setting "obrowser > htmlControlPort" in config.json.

If debugging in OBrowser is enabled (if the canDebug item in the 'obrowser' section of config.json is true), OBrowser opens another port to allow remote debugging of the web content. By default this is port 5989, but can be overridden by setting the remoteDebuggingPort item in the 'obrowser' section of config.json.

Events

Omnis HTML controls have some basic events, such as single and double click, but they can have custom events.

evCallMethodDone

The evCallMethodDone event is triggered when a $callmethod() is completed: it has three parameters in addition to pEventCode:

Parameter	Description
pUniqueId	The unique id that was returned by $callmethod(). This associates this event with the original call.
pReturn	The return value of the control method. NULL if an error occurred - see pErrorText for details.
pErrorText	Text describing the error.

evClick and evDoubleClick

Omnis HTML controls can generate standard click and double click events, with the pLineNumber event parameter.

evControlEvent

Omnis HTML controls can generate custom events. Each custom event sends evControlEvent. This has one parameter in addition to pEventCode:

Parameter	Description
pInfo	A row containing information about the event. If the control generates more than one type of control event, a column in this row can identify the event type

Menu Classes

A **Menu class** defines a pulldown menu that can be installed on the main Omnis menu in desktop apps (not web or mobile apps). A menu class can also define a Popup menu control that can be displayed on a window class. You can create a new Menu class from the Studio Browser using the New Class>>Menu or Class Wizard>>Menu… option.

Menus let end users perform standard operations in your desktop application, such as open a window for data entry or print a report. The definition for a standard menu is stored in a *menu class*. You can create your own custom menus and install them on the main application menu bar using the *Install menu* command, typically in the Startup_Task. You can install a menu on the menu bar of a window, or open it as a popup or context menu on a window class. You can create hierarchical menus that drop down off another menu, and you can incorporate standard Omnis menus such as File and Edit into your application.

The types of menu classes you can create are:

- **Standard dropdown menus**
  you can install any menu class on the main menu bar; you can add shortcut keys and control access to menus using user levels, you can check and uncheck individual menu lines and enable/disable them.
- **Hierarchical menus**
  you create a hierarchical menu as a separate menu class and add it to another menu line; when the user selects the line a menu drops down
- **Popup menus**
  this type of window field pops up a standard menu when you click on it
- **Window menus**
  you can install any standard or custom menu on the menu bar in a window class
- **Context menus**
  you can define a context menu that pops up when you Right-click on a field or window

You can add up to 500 lines or menu items to a menu class, but in practice you will only need the first twenty-or-so for most types of menus. You can add a keyboard alternative, or shortcut key, to each menu line when you create the class.

Methods do the real work behind the menu. You can add methods to the class itself and each menu line. The class methods can initialize the menu when it is installed, and the line methods could do anything from open a window, print a report or series of labels, or insert a row into your database. When you select a line in the installed menu, Omnis runs the method behind that menu line.

Menu wizards

You can create a menu using one of the wizards or templates available in the Studio Browser.

To create a new menu using a wizard

- Select your library in the Studio Browser
- Click on the Class Wizard option, then click on the Menu option
- Under the *Wizards* option, select **Menu Wizard** and click on the Create button, or under the *Templates* option, select **File Menu** to create a standard File menu
- Follow the instructions on screen

The following wizards and templates are available:

- **Menu Wizard**
  creates a menu containing menu lines to open window classes and print report classes; can also contain hierarchical menus
- **File Menu Template**
  creates a menu that you can use to replace the standard File menu; you can edit this menu class and add your own menu lines

You can also create a new menu class using the New Class>>Menu option in the Studio Browser. Having created the menu class, add the title or name of the menu in the $title property. To add a line, right-click on the menu title and select **Add Line.** Add the text for the menu line in the $text property for the line. To add a second line, right-click on the first menu line option and select **Add Line.**

Menu lines and methods

You can create a menu class and add each menu line, but to make your menu properly function you need to add some programming behind your menu. If you build a menu class from scratch you will have to add methods to the menu yourself, but if you built your menu using the menu wizard then Omnis will have added the appropriate methods to your menu automatically.

You can add *class methods* to the menu itself to control the menu when it is installed. And you can add *line methods* to each line in your menu by double-clicking on the menu line to open the method editor: a line method is executed when the corresponding menu line is selected in the installed menu.

Many of the standard functions of a menu, such as enabling or disabling a menu line, adding shortcut keys, or setting passwords for each menu option, are properties of each menu line. You can edit the properties of a menu or menu line using the Property Manager.

Menu Icons

Your own custom menus can have icons for each menu line. Menu lines have the $iconid property in which you can specify the id of an SVG file or a 16x16 PNG icon for the menu line; larger icons are not available for menu lines. If the property is empty (the default) the menu line does not have an icon.

The icons can be SVG files from an iconset (or PNGs from an iconset, the #ICONS system table, or the OmnisPIC or USERPIC icon data file in legacy apps only). See Selecting an Icon for more details about specifying icons, SVG icons and iconsets.

The **$iconcolor** property for a menu line (or toolbar button) sets the icon color when using a themed SVG icon. The **$defaulticoncolor** property for a menu class (or toolbar) sets the icon color when using themed SVG icons *and the $iconcolor property of the item is kColorDefault*. If $defaulticoncolor is also kColorDefault, then themed icons use the text color.

Menu Shortcut Keys

You can specify a shortcut key or keyboard alternative for each line in your menu. When the end-user presses the specified key combination the menu line is activated. Under Windows, you can add Ctrl and Alt key combinations to menu lines. Under macOS you can add Cmnd and Option key alternatives. You can further modify keyboard alternatives with the Shift key under any OS. You enter these keys in the Property Manager for the menu line, or by pressing the required key combination when the appropriate menu line is selected in the menu editor.

The menu editor context menu has the option **Accept All Key Strokes.** When checked (the default) the menu editor accepts all keystrokes, including shortcut keys, and enters them into the current menu line. When this option is unchecked, you cannot enter menu lines directly from the keyboard, in this case you have to enter the text and shortcut key for each menu line in the Property Manager.

To add a shortcut key in the menu editor

- Open your menu class in design mode
- Select the menu line and press the key combination you want to assign to it

For example, select the menu line and press Ctrl or Cmnd and the number key "5" to add the Ctrl/Cmnd-5 shortcut key, or press Ctrl or Cmnd and the letter "A" to add the Ctrl/Cmnd-A shortcut key to the current menu line. Whichever platform you are using, the appropriate shortcut key is entered for all platforms automatically.

Certain shortcut keys cannot be inserted in this way, because they have functionality that is detected and intercepted by Omnis or the operating system. They are Ctrl/Cmnd-T and Ctrl/Cmnd-S on all platforms, plus Cmnd-Q and Cmnd-W under macOS.

You can specify menu shortcut keys for a menu line in the Property Manager by assigning a value to the $winshortcutkey or $macshortcutkey.

You should avoid using standard key combinations that appear in Omnis or the operating system. macOS function keys on extended keyboards activate the menu option with the corresponding Cmnd-number combination. Thus, F1 is the same as Cmnd-1. You cannot use the Shift-Cmnd-n options, where n is a digit from 0-9, because macOS uses these options.

Furthermore, you should make sure that no two menu items in a menu have the same shortcut key and that no two menus installed at the same time have the same shortcut key. Duplicates will be unpredictable depending on which menus are installed at the time or the order in which they appear on the main menu bar.

Menu Shortcuts (macOS)

The "useFnInMenuShortcuts" item in the "macOS" section of the config.json file controls how Function keys on macOS are interpreted. When set to true (the default), the Function+number menu short cuts display as Fn, or if false they display as +n. (ST/MC/264)

Alt Shortcuts Keys under Windows

Under Windows you can add Alt key equivalents to menu lines and to the menu title itself. You specify the key by including an ampersand (“&”) before the character in the menu line or menu title. For example, if you want your users to open a menu called Travel with the Alt-T key combination, add an ampersand before the T in the menu title. In this case, the text for the menu title should be “&Travel”. Likewise, you can add an Alt shortcut key to any letter in a menu line. For example, to add the Alt-S shortcut key to a “Customers” menu line, the text for the menu line should be “Cu&stomers”. To display an ampersand (“&”) as literal text in a menu title or menu line, you need to insert two ampersands, for example, “Profit && Loss” to display “Profit & Loss”.

You can include the ampersand in the appropriate menu line or title when you enter the item in the menu class editor, or you can add it to the text or title property for the item in the Property Manager. Usually, you add shortcut keys as an afterthought, in which case it is easier to do it in the Property Manager. Note you cannot select the menu title or line and press the required Alt-key combination to assign this type of shortcut key: you must enter it directly into the menu editor when you enter the line or using the Property Manager.

Hierarchical menus

A *hierarchical menu* or submenu is a menu that drops down from another menu line when you select the option. You can create a hierarchical menu using any previously defined menu class. To add any previously defined menu to a menu line, to create a submenu, you add the name of the menu class to the $cascade property for the menu line in your main menu.

When you add a hierarchical menu to a menu line, Omnis places an arrow against the menu line in the menu editor. This also appears in the installed menu indicating there is a hierarchical menu attached to this menu line. When you select the option in the installed menu the hierarchical menu drops down.

You can cascade menus up to five levels deep under macOS, and up to eight levels under Windows. You should avoid cascading a menu off itself, or creating a chain of menus that cascade off each other recursively.

Rebuilding submenu instances

Each time you add a line to a menu instance, delete a line from a menu instance, or change the $cascade property at runtime, Omnis rebuilds the menu, and as part of the rebuild process it destroys and recreates instances for cascading menus. You can manage the rebuilding of a menu instance by assigning kTrue to $disablerebuild, and then setting it back to kFalse after updating the menu, so that the menu rebuild only occurs once.

Menu Instances

$menuinst for a Cascading menu object (and a Popup menu) will appear in the Notation Inspector. This allows you to select $menuinst and see the properties of the instance, and also drill down further to see $objs, etc. Note that $menuinst only appears in the Notation Inspector when the parent object has an associated instance.

Window menus

A *window menu* is a menu installed on the menu bar of a window. A window menu bar is a property of the window class itself. To show the menu bar for a window you must enable the $hasmenus property. Having enabled the menu bar for a window, you can drag menus from the Studio Browser and drop them onto a window menu bar.

When you enable the window menu bar all the objects on your window including fields and background objects will move down. You can add any type of menu to a window menu bar, including your own custom menus or the standard Omnis menus, such as File or Edit: you add a standard menu by right-clicking on the window menu bar and selecting the menu you require.

Popup menus

A *popup menu* is a type of window field that opens a menu when you click on the field. You can create a popup menu using any previously defined menu class and you can add any of the standard Omnis menus such as File and Edit to your window as popup menus. When you create a popup menu field you enter the name of the menu class in the field’s $menuname property.

You can use the constant kDefaultBorder for the $effect property to ensure the menu has the default border style for the current operating system.

Context menus

A *context menu* is a menu that pops up when you Right-click on the background of an open window or a field; under macOS you Ctrl-click to popup a context menu. Context menus appear throughout the Omnis design environment to help you access methods and so on, but you can add context menus to any of the windows in your application. To create a context menu, you enter the name of a previously defined menu class in the $contextmenu property for the window class or field.

Security and menu access

You can restrict access to certain parts of your library, including menu items, by setting up a secure system of passwords. You can define varying degrees of access for up to eight groups of users (each type of user or group is assigned a separate password), plus a master password. Several different users can use the same password: you are not limited to literally eight users. You can use the passwords set up in your library to control access to the menus in your library.

Access to menus or menu lines is set in the $users property for the menu item and utilizes the user numbers and passwords set up in your library. The default is to allow access to all the menu items in your library for all users or passwords, that is, passwords 1 to 8. The master password has access to all menus at all times. The users property contains the string “12345678”. To restrict access to a menu or menu line, you delete the user number from the users property. For example, to restrict access for user 4, delete the number 4, which leaves the string “1235678”. To allow access to a menu item, you include the user number in the users property for the menu or menu line. For example, to allow access for user 4 only, delete the default string and enter the number 4 only.

Status bar help for menus

You can add short help messages to menus and menu items that display on the window status bar or the main Omnis help bar. You enter the message in the $helptext property for the menu title or menu line. The $hasstatusbar window property enables the status bar for a window class, and the $helpbaron Omnis preference enables the main Omnis help bar. You can change the font and point size for the main Omnis help bar with the $helpfont property. Under macOS Classic you can display the menu help text in Help balloons by enabling the $balloonson property.

Toolbar Classes

A **Toolbar class** defines a custom toolbar that can be installed onto the main Omnis toolbar in desktop apps only (not web or mobile apps). A toolbar can float inside the application window, or can be added to a window class. You can create a new Toolbar class from the Studio Browser using the New Class>>Toolbar or Class Wizard>>Toolbar… option.

You can create your own toolbars that contain buttons and other controls that lets the user access common options and functions in your application. You can install your own custom toolbars in the top, left, right, or bottom docking area of the main Omnis application window using the *Install toolbar* command, typically in the Startup_Task or from a custom menu. You can add toolbars to any window class too, but you must create your toolbar class first before you can add it to a window. The toolbar commands, such as *Install toolbar,* refer to toolbars in the main docking areas, not window toolbars.

Toolbar classes can contain methods and variables. As with menu classes, the methods you place behind each control do the real work in a toolbar. You can add methods to the toolbar class itself and to each control or button. When you click on a control in your toolbar, Omnis runs the $event() method behind the tool.

Toolbar Wizards

Under the Class Wizards option in the Studio Browser there are a number of templates that you can use as the basis for your own toolbars. They include a **File, Standard,** and **View** template which you can use in a window toolbar: the methods behind the controls in these toolbars call methods in the current window.

To create a new toolbar using a wizard

- Select your library in the Studio Browser
- Click on the Class Wizard option, then click on the Toolbar option
- Select the toolbar template you require and click on the Create button
- Follow the instructions on screen

The following toolbar templates are available:

- **File**
  toolbar contains buttons to Open and Close
- **Standard**
  toolbar contains Save, Revert, Print, Page Setup, and Help buttons

- **View**
  toolbar contains buttons to switch between icons and details view

NOTE: all these toolbars are designed to be installed on a window menu bar and not the main Omnis toolbar since each button in the toolbar contains an event method that calls a method which you have to add to the parent window.

You can also create a new toolbar class using the **New Class>>Toolbar** option in the Studio Browser. Having created the toolbar class, you can drag components from the Component Store and drop them into the toolbar editor.

Toolbar Controls

You can add as many buttons and controls as you like, and you can use the separator control to put space between groups of tools. The following toolbar controls are available:

- Check Box
- Color Picker
- Combo Box
- Dropdown list
- Font List
- Font Size List
- Line Style Picker
- Pattern Picker
- Popup List
- Popup Menu
- Push Button
- Radio Button
- Spacer

You can drop a control anywhere within the active area or between existing controls. Once you've placed controls in a particular order, you can drag them to the left or to the right to reposition them on your toolbar. Dragging a control from one toolbar class to another copies the control to the destination class.

If you place several radio buttons together, without separators, they behave as a group. That is, when you select one radio button in a group the currently selected one will be deselected. Popup lists and menus look like buttons, but when you click on them in the installed toolbar a list or menu drops down.

Combo boxes

Combo boxes can be used in unified toolbars on macOS. They have the rounded edit field appearance, like the finder toolbar search box.

The $iconid property is available for combo boxes in toolbars, but applies to macOS only. It specifies the icon drawn to the left of the control, where a click opens the menu containing the combo box list.

- When set to zero, the icon is the standard macOS search icon
- When set to a valid value, the 16x16 icon selected replaces the search icon
- When set to an invalid value, the control just shows a down arrow like a combo box

To use a combo box for searching (for example), check for evAfter with the next event of evOK; this is generated when you hit return with the focus in the combo box.

Radio Button Groups

On macOS, if **$splitbuttonbars** is set to kTrue, a group of radio buttons (of kToolRadioButton type) will be displayed as separate buttons with their respective labels; the default is kFalse where radio buttons are displayed in a compact group. The property only applies to toolbars on macOS, including main toolbars, floating toolbars and for unified window toolbars where kTBOptionmacOSOmnisTopToolbar is true.

Toolbar Separators

Toolbar separators or "spacers" have a property $flexible that can be set to kTrue when $blank is kTrue, as follows:

- If true, and $blank is also true (that is, no dividing line is shown), and the toolbar is on a window, the separator expands in width to use docking area space not occupied by other toolbar objects. If more than one separator is flexible, the unused space is shared equally between them.

A typical use of $flexible is to place a single separator in the toolbar, and set $blank and $flexible to kTrue. The result is that any controls to the right of the separator become right justified.

Toolbar Icons

You can add an icon to most types of controls, all except the list and combo types. You can specify an icon for a control in its $iconid property. The icons can be SVG files from an iconset (or PNGs from an iconset, the #ICONS system table, or the OmnisPIC or USERPIC icon data file in legacy apps only). See Selecting an Icon for more details about specifying icons, SVG icons and iconsets.

Controls that you can check or uncheck, such as radio buttons and check boxes, display different icons for the different checked and unchecked states. You can create icons for all these states using the standard naming for icons in an Icon set (in legacy apps, all possible states for these controls are stored in the OmnisPIC icon data file).

The **$iconcolor** property for a toolbar button (or menu line) sets the icon color when using a themed SVG icon. The **$defaulticoncolor** property for a toolbar class (or menu) sets the icon color when using themed SVG icons *and the $iconcolor property of the item is kColorDefault*. If $defaulticoncolor is also kColorDefault, then themed icons use the text color.

Tooltips

You can add tooltips to individual toolbar controls in the $tooltip property for the control. To hide and show tooltips for the toolbars in Omnis and your libraries you can set the $showtoolbartips Omnis preference.

Combo boxes can have a content tip when there is no text label drawn in the toolbar; the content tip uses the value of the text property.

Toolbar Methods

You can add *Class methods* to the toolbar class itself to control the toolbar when it is opened, and you can add *Tool methods* to the controls in your toolbar: a tool method is executed when the corresponding tool or control is clicked on in the installed toolbar.

You can add up to 501 methods to each tool or control in your toolbar, and a further 501 methods to your toolbar class. You enter the methods for tools (buttons) and toolbar classes using the method editor, by double-clicking on the control or the background of the toolbar editor.

Toolbar classes contain a $construct() and a $destruct() method by default. You can add code to these methods that control the installing and closing of the toolbar. In addition, all controls except spacers have an $event() method in which you add the code you want to run when the tool or control is clicked on. For example, you could use the *Open window instance* command in a tool method to open a window, or you could use the *Print report* command to print a report to the current destination.

Installing Toolbars

You can install a toolbar class at any time from the toolbar editor itself; this is useful if you want to see how the toolbar looks while you're designing it. However, in your finished application you can use the *Install toolbar* command or the notation to install a toolbar. You can also add any toolbar class to the docking area of a window using the $toolbarpos property.

The $initialdockingarea property of a toolbar class determines which docking area the toolbar is installed into. Toolbars install into the top docking area of the main Omnis application window by default.

If you have enabled the $allowdrag property for toolbar class, you can drag the installed toolbar out of the docking area; the toolbar is now *floating*. If you have enabled the $allowresize property in the toolbar class, you can resize the floating toolbar.

You can Right-click on a docking area and select the Show Text option from the context menu to show the text for each toolbar control.

Docking Areas

You can install a toolbar into the top, bottom, left, or right docking area in the main Omnis application window. Note that most list-type controls, such as the Dropdown list and Combo box types, are not displayed in a toolbar if it is installed into the left or right docking area. They display as expected when the toolbar is at the top or bottom, or is floating.

You can right-click on a docking area to open its context menu which lets you show and hide the text labels for any installed toolbars. You can show text for the IDE toolbars as well as your own custom menus.

You can view and change the properties of the main docking areas using the Notation Inspector and the Property Manager.

To view the docking areas in the Notation Inspector

- Press F4/Cmnd-4 to open the Notation Inspector
- In the Notation Inspector, expand $root and the $prefs group
- Expand the $dockingareas group

Toolbar Docking Area Height

The library preference $windowsizeexcludesdockingarea allows you to ignore the height of the toolbar docking area when specifying the position of a window. If true (the default in new libraries), the width and height of a window exclude the relevant dimension of the toolbar docking area (this does not affect windows under macOS with a standard macOS top toolbar since they are excluded from the docking area automatically).

Window Toolbars

A Toolbar class can be added to a window class, rather than the main application window. On Windows it is added to the window docking area (top. Left, right, or bottom), whereas on macOS the toolbar is added to the title bar of the window. There are a number of window class properties that control toolbars:

Property	Description
$enablemenuandtoolbars	If true, all main toolbars and menus are enabled when this window is on top
$toolbarnames	The names of the toolbar classes used in the window docking area
$toolbaroptions	Set of toolbar options; see below. Note that you can only assign this property when $t is, not kDockingAreaNone. Also note that kTBOptionmacOSOmnisTopToolbar can onl designing a window
$toolbarpos	The position of the toolbar in the window; this can be any of the kDockingArea... cons kDockingAreaBottom, kDockingAreaLeft, kDockingAreaNone, kDockingAreaRight, kD is, except kDockingAreaFloating)

Toolbar Options

The $toolbaroptions property affects the appearance and style of the window toolbars. The following constants are available, which can be selected in the Property Manager in design mode:

Constant	Description
kTBOptionDefault	Use settings stored in Omnis config file
kTBOptionLargeIcons	Show large icons
kTBOptionmacOSCompressed	Any macOS title bar toolbar which is using the unified style will automatically min between toolbar items; see below
kTBOptionmacOSExpanded	Any macOS title bar toolbar will appear below the window title and the window ti see below
kTBOptionmacOSOmnisTopToolbar	On macOS, for window styles that support macOS style top toolbars, use Omnis s macOS style for the top toolbar; this option is only applied when opening the win
kTBOptionNone	Small Icons, no text

Constant	Description
kTBOptionShowText	Show text
kTBOptionVarTextWidth	Button width depends on text width

Expanded Toolbars (macOS)

The Omnis configuration item **useToolbarStyleExpanded** in the ‘macOS’ section of the cofig.json file enables the legacy expanded toolbar style instead of the default toolbar style (typically unified). This only applies to macOS 11 and later, but when set to true the window toolbar style will use the legacy expanded style, i.e. toolbars sit under the window title. By default, this is false and toolbars will use the new automatic style on macOS 11 and later, i.e. toolbars are unified and to the right of the window title.

kTBOptionmacOSExpanded

When selected **kTBOptionmacOSExpanded** will place any toolbar displayed in the title bar of a window on macOS Big Sur or later below the title with the title centered as with previous versions of macOS. This setting is overriden by the global configuration file ‘useToolbarStyleExpanded item in the ‘macOS’ group in config.json (added in Studio 10.2). When set to true all windows on macOS will use the expanded style. By default on macOS Big Sur and later a toolbar in the title bar will appear unified and be positioned next to the title.

kTBOptionmacOSCompressed

When selected **kTBOptionmacOSCompressed** will minimize the space between toolbar items for a macOS unified toolbar, i.e. where the toolbar title appears to the left of the toolbar items. The text label for items are also hidden. This option only has effect where the unified toolbar is supported (macOS Big Sur and later). For this option to be active the kTBOptionmacOSExpanded or kTBOptionmacOSOmnisTopToolbar option cannot be set. Space can be added between toolbar items by using a blank kToolSpacer toolbar component.

Toolbar Style for macOS

When the window style supports it, window toolbars default to being standard operating system toolbars that are part of the window title bar. All toolbar controls are supported under Mac OS X 10.5 apart from the combo box toolbar controls (which cannot be supported due to focus issues).

You can access the old toolbar styles using the kTBOptionOSXOmnisTopToolbar option in the $toolbaroptions property of the window class. If true, the $toolbarbutton property adds a button in the title bar of the window to hide and show the toolbar (Mac OS X only, this only applies when the window has a Mac OS X style top toolbar).

The method $toolbarbuttonclick() sends a click to the toolbar button on the window instance (Mac OS X only, this only applies when the window has a Mac OS X style top toolbar).

Accordion Control

Group	Icon	Name (type)	Description
Navigation	☰	Accordion Control (Xcomp)	List of expandable options

The **Accordion** control presents a list of hyperlink options, each of which has a header link that, when clicked, expands to show more information to the end user. In addition, each option in the list can include a further link option or an icon which users can click on. You can control how the options in the list expand and collapse, creating a very interactive selection tool for end users.

Populating the Accordion

The contents for the Accordion control is provided by a list variable specified in its $dataname property. The list should have at least two columns to specify the Heading text and the expanded content or text, while you can add a third column for the additional Link text or icon. You can set the $headingcolumn, $contentcolumn, and $headinglinkcolumn properties, to specify which list columns are used for the Heading text, Text content, and Link text, respectively.

For example, the following method defines a list with three columns to contain the Heading text, the content text, and the link text. You would typically add multiple lines in the list, which could be static data or loaded from a database.

```
# define iAccList (List) iAccHeading, iAccText, iAccLink (Chars)
Do iAccList.$define(iAccHeading,iAccText,iAccLink)
Do iAccList.$add(\
   "Heading 1",con("Some Text",chr(13),"over multiple lines ",\
   style(kEscColor,kRed),"with word wrap",chr(13),\
   "and some more lines",chr(13),"and some more"),"Action 1")
# and so on, to populate the Accordion list of options
```

The column specified in $headinglinkcolumn can be a Character or Long Integer to either represent some hyperlink text or an icon id. In addition, when $headinglinkcolumn specifies an icon id, you can include another column in the list to add a tooltip for each icon. The column containing the tooltips is specified in $icontooltipcolumn.

For example, the following method is very similar to the previous example, but this time the link column is defined as a Long integer and icon IDs are added to the list contents.

```
# define iAccList2 (List), iAccHeading, iAccText (Chars), and iAccIconId (Long Int)
Do iAccList2.$define(iAccHeading,iAccText,iAccIconId)
For lNum from 1 to 100 step 1
Do iAccList2.$add("Heading A",con("Text ",lNum,chr(13),"over multiple lines ",style(kEscColor,kRed),"with w
```

The icons can be added to the #ICONS system table in your library, or stored in an icon set.

Note in the example above, that your content column can contain styled text, assuming the $styledtext property for the control is set to kTrue. The content column can also contain multiple lines of wrapping text, using a carriage return (chr(13)) as the line delimiter.

Expand mode and Animation

The $expandmode property specifies what happens when the user expands or closes an entry in the Accordion control. The property is a constant as follows:

Constant	Description
kACCexpandModeZeroOrOne	Zero entries or one entry must be open; when you open an option, any previous option will collapse automatically
kACCexpandModeOne	One entry must be open; the first option will open by default, then when you open another option, the previous option will collapse automatically
kACCexpandModeZeroOrMore	Zero or more entries can be open; options must be opened and closed manually
kACCexpandModeOneOrMore	One or more entries must be open; the first option will open by default, then further options can be opened and closed manually

If kTrue, the property $fadetext forces the text to fade in or out when the entry is opened or closed, while the $animationsteps property specifies the number of steps used to animate the opening or closing of an entry in the list (in the range 1-64).

Accordion Events

When the user clicks on the hyperlink (present if hyperlink column is present) the control generates evClick, with pLineNumber set to the clicked line number. Like many other list types, you could trap the line selected in the $event() method for the control, and return the value of the heading text, the option text or some other value in another column in the list to make a selection in subsequent code.

Breadcrumb Control

Group	Icon	Name (type)	Description
Navigation	≡H	Breadcrumb Control	Displays “location” within the hie an application

The **Breadcrumb** control can be used to display the end user's "location" within the hierarchy of an application, as well as allowing the end user to navigate back up the system by clicking on one of the "crumbs" or the "Home" crumb. A breadcrumb control is often seen in the context of a website, as secondary navigation, but it can used to enhance the UI for many types of desktop applications, such as consoles and dashboards.

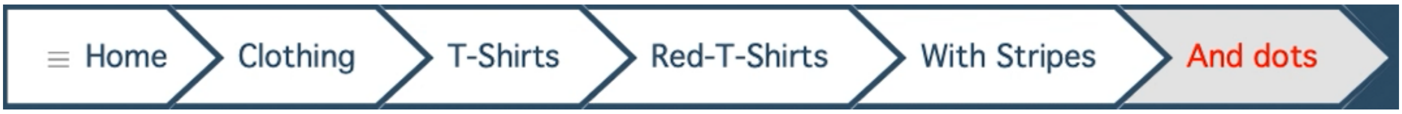


Figure 141:

The Breadcrumb control can be displayed in different styles: Arrow (as above), Rounded rectangle, or plain Text; this is set in the $crumbstyle property. The following is the rounded style, showing the third crumb highlighted when the pointer is over it.

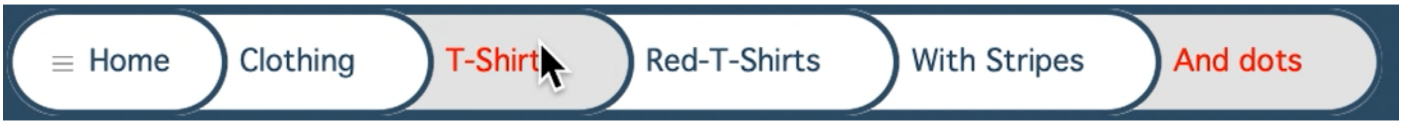


Figure 142:

The following shows the plain text style, showing the fourth crumb highlighted when the pointer is over it.

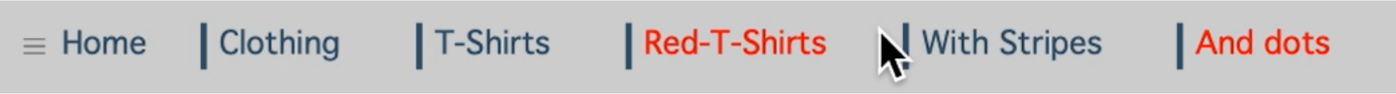


Figure 143:

Properties

The content for the other crumbs in the control are taken from a list assigned to its $dataname property. The list has three columns: crumb_text (Character), crumb_id (integer), crumb_icon_id (Character) – the icon ID is the name of an icon in an icon set assigned to the library. The first crumb in the control is referred to as the "Home" crumb and is always visible; its text and icon are defined in the $homecrumbtext and $homecrumbiconid properties. The first line in the data list is the first animated crumb to pop out from the Home crumb, with subsequent list lines being successive crumbs in the control.

The Breadcrumb control has the following properties:

Property	Description
$homecrumbiconid	The icons id used for the home crumb
$homecrumbtext	The text shown on the home crumb.
$crumbstyle	The drawing style of the breadcrumb, a constant: kCrumbStyleText kCrumbStyleRoundRect kCrumbStyleArrow
$textcolor	The crumb text color
$crumboutlinecolor	The outline color of breadcrumbs
$crumbcolor	The color of a breadcrumb
$activecrumbcolor	The color of the active breadcrumb
$activecrumbtextcolor	The text color of the active breadcrumb
$showactivecrumb	If kTrue the active crumb (the final crumb on the path) is shown in the active color

Events

The Breadcrumb control reports the evBreadCrumbPathChanging event which is sent when the user selected a crumb in the path, with the event parameters pBreadCrumbID, which is the id of the crumb (column 2 in the crumb list definition), and pNewBreadCrumbList the proposed new path list.

This event can be discarded with Quit Event (Discard Event) which will prevent the default action which is to shrink the path based on the crumb selected.

Example

The following code will create a breadcrumb as shown in the examples above.

```
# where bread_crumb is a list assigned to $dataname of the Breadcrumb control
Do bread_crumb.$define(crumb_text,crumb_id,crumb_icon_id)
Do bread_crumb.$add("Clothing",1,"")
Do bread_crumb.$add("T-Shirts",2,"")
Do bread_crumb.$add("Red-T-Shirts",3,"")
Redraw bread_crumb
```

There is an example app to demonstrate the Breadcrumb window control in the **Samples** section in the **Hub** in the Studio Browser, and on the Omnis GitHub repo at: https://github.com/OmnisStudio. Search for Omnis-Breadcrumb.

Button Area

Group	Icon	Name (type)	Description
Buttons	⬚	Button Area	Invisable area that responds to u

A **Button Area** is in essence an invisible button and behaves exactly like a pushbutton. See Pushbutton for details about button areas.

Calendar Control

Group	Icon	Name (type)	Description
Other	▦	Calendar Control (Xcomp)	Presents dates in a standard cal format

The **Calendar** control presents the current month and today's date in a standard calendar format. The component has many useful properties that let you control the overall appearance of the calendar, such as fonts, colors, and the display style for days and headers. To display today's date in runtime, you need to set the $currday property. For example, to set the current day of the calendar component to today's date you could use:

```
# method contains instance var iCurrentDate (DateTime) with default value of #D
Do $cinst.$objs.calendar.$currday.$assign(iCurrentDate)
```

Modern UI Style

From Studio 11 onwards, the UI in the Calendar external component has been enhanced to display dates using a modern interface. The **$uistyle** property must be set to **kCalUIstyleModern** to enable the new UI; the **kCalUIstyleClassic** setting maintains the existing drawing style (which is the default). The following properties are only available when $uistyle is set to kCalUIstyleModern:

Property	Description
$navcolor	The color used for the navigation bar
$navfont	The font used for the navigation bar
$navfontsize	The fontsize used for the navigation bar
$navtextcolor	The color of the text in the navigation section
$showmonthnav	If true, if the navigator bar is shown
$weeknumbercolor	The color for the background of week numbers if shown
$weeknumbertextcolor	The text color for week numbers if shown
$showweeknumber	If true, the week number column is shown
$setdayicon	Sets the day icon

The following existing properties are ignored if $uistyle is set to kCalUIstyleModern: $daymode, $currdaymode, $headingmode, $otherdaymode.

Navigation bar

When the **$showmonthnav** property is set to kTrue the navigation bar is shown. Clicking on the right or left arrow in the navigation bar will change the month view.

If you click on the month displayed in the navigation bar the month selector is shown in the main calendar, and likewise selecting the year in the navigation bar the year selector is shown in the main calendar. Selecting a cell from any selector returns the calendar to the previous selector and eventually to the default mode.

If **$allowchange** is set to false, the left and right navigation buttons are removed and various navigation selectors unavailable.

If **$showweeknumbers** is set to kTrue, week numbers are displayed down the left side of the calendar; the color of the week number text and background is controlled using the $weeknumbertextcolor and $weeknumbercolor properties, respectively.

Calendar Events

The Calendar component reports two events: evDateChange, sent when the date changes, and evDateDClick, sent when the user double-clicks on a date cell. Both these events pass the pCurrentDate event parameter. You can detect these events in the $event() method of the component; assuming you're using a variable iCurrentDate as above you could do:

```
On evDateChange
  Calculate iCurrentDate as pCurrentDate
  # then Do something else
```

Calendar Example

The following example uses the calendar component, and provides lists to set the month and year, and buttons to display the next or previous months.

The following methods initialize the calendar window and format the display of the calendar component.

```
# $construct() method of the calendar window
# window contains iCurrentDate (DateTime), iMonth (Char), iMonthList (List), iYear (Number), iYearlist (Lis
Calculate iCurrentDate as #D
Do method $setdroplists
# $setdroplists method in the window
Do iMonthList.$define(iMonth)
Calculate assignloop as dpart(kMonth,iCurrentDate)
For loop from 1 to 12 step 1
  Calculate tmp as con("1 ",loop," 90")
  Do iMonthList.$add(dname(kMonth,tmp)) Returns line
  If assignloop=loop
    Do iMonthList.$line.$assign(iMonthList.$linecount)
    Calculate tmp as dadd(kMonth,-1,tmp)
    Do $cinst.$objs.back.$text.$assign(

      con("< ",dname(kMonth,tmp))) ## sets the Previous butt
    Calculate tmp as dadd(kMonth,2,tmp)
    Do $cinst.$objs.next.$text.$assign(

      con(dname(kMonth,tmp)," >")) ## sets the Next butt
  End If
End For
Do iYearList.$define(iYear)
Calculate assignloop as dpart(kYear,iCurrentDate)
For loop from 1970 to 2030 step 1
  Do iYearList.$add(loop) Returns line
  If assignloop=loop
    Do iYearList.$line.$assign(iYearList.$linecount)
  End If
End For
Do $cinst.$objs.calendar.$currday.$assign(iCurrentDate)
```

The calendar example window has a list for selecting a month and one for the year. The methods for these two lists are as follows:

```
# $event() method for the Month selection list
On evClick
  Do iMonthList.$line.$assign(pLineNumber)
  Do iMonthList.$loadcols()
  Calculate iCurrentDate as con(dpart(kDay,iCurrentDate),

      ' ',iMonth,dpart(kYear,iCurrentDate))
  Do method $setdroplists
  Do $cinst.$objs.month.$redraw() ## redraw the lists
  Do $cinst.$objs.year.$redraw()
# $event() method for the Year selection list
On evClick
  Do iYearList.$line.$assign(pLineNumber)
  Do iYearList.$loadcols()
  Calculate iCurrentDate as con(dpart(kDay,iCurrentDate),

      ' ',dpart(kMonth,iCurrentDate),' ',iYear)
  Do method $setdroplists
  Do $cinst.$objs.month.$redraw() ## redraw the lists
  Do $cinst.$objs.year.$redraw()
```

In addition, the calendar example window has pushbuttons for showing the Next and Previous months. The $event() method for the Previous button is as follows:

```
On evClick
  Calculate iCurrentDate as dadd(kMonth,-1,iCurrentDate)
  # or Calculate iCurrentDate as dadd(kMonth,1,iCurrentDate)

# to advance the month by one
  Do method $setdroplists
  Do $cinst.$objs.month.$redraw() ## redraw the lists
  Do $cinst.$objs.year.$redraw()
```

Check Box

Group	Icon	Name (type)	Description
Buttons	☒	Check Box	Displays On or Off choices with a mark (also Yes or No, and 1 or 0 v

The **Check box** can represent boolean data, that is, they can display On or Off choices, Yes or No, and 1 or 0 values. The field you associate with the check box by setting its $dataname should be a Number or Boolean field. Checking a check box sets the value of the field to one; 'unchecking' or clearing the check box sets the value to zero. You enter the text to display to the right of the check box in the $text property for the object. Omnis calls the field method for a check box when the user clicks on the field.

The images for the check box on and off states (i.e. the box ticked and not ticked) are stored in the Omnispic icon data file in the 'Multistate 2' page (Multistate 3 for macOS Classic).

Check boxes report the evClick event, but no parameters are passed. You can detect the evClick event in the $event() method for the object.

For example, in a login screen the user can enter a username and password, with the option to store their details by checking a check box. The check box is associated with the iRemember variable which is a Boolean. When the user checks the box, the variable is set to true and this value is used in the methods that set up the user account; specifically, if the check box is set, the user information could be stored on the client machine using a cookie component. Here is the method that could handle this:

```
# part of the $checkuser method
If iRemember ## if check box checked then store user info
  Calculate $cinst.$objs.cookie.$userdata as con(iUserName,"@:@",iPassword)
End If
```

Horizontal Mode

The Check Box has a "horizontal" mode which makes the check box look and behave like a slider switch, which animates between the on/off state. The $buttonmode setting kCheckBoxHorizonal enables the horizontal behavior.

The existing appearance properties $textcolor, $forecolor and $iconid specify the text color, forecolor, and icon for the selected state, and $backcolor is the background of the switch, while the properties $secondaryforecolor, $secondarytextcolor, and $secondaryiconid specify the appearance for the off state of the control.

The horizontal check box animates by default but this can be disabled by setting $animateui to kFalse. The animation is disabled when the horizontal check box is used in a complex grid.

When button mode is kCheckBoxHorizonal the $text property is a comma separated list allowing you to specify the text for the off and on states of the control. For example, $text = "PERSONAL,BUSINESS", or $text = "OFF,ON" will display the check box as follows:

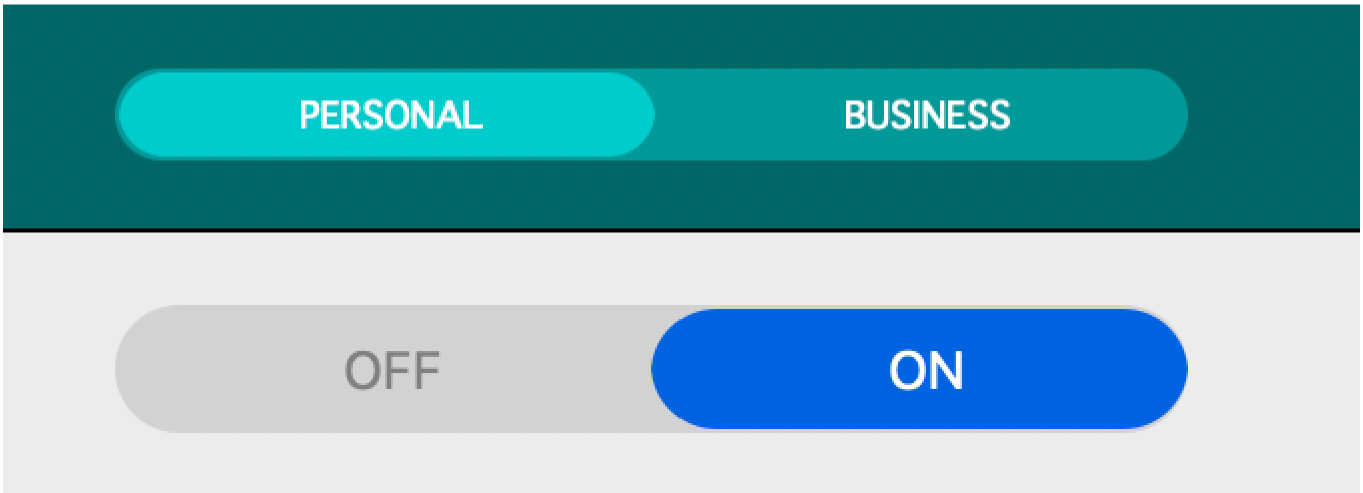


Figure 144:

There are a number of new theme colors to control the appearance of the horizontal check box; by default the theme colors are set to kColorDefault.

Color	Description
horizontalcheckbox	the appearance group name for this control.
background	background color of the control.
switchon	background color of the switch in the 'on' state
switchontextcolor	text color of the switch in the 'on' state
switchoff	background color of the switch in the 'off' state
switchofftextcolor	text color of the switch in the 'off' state

There is an example app to demonstrate the Check Box window control, including the horizontal mode, in the **Samples** section of the **Hub** in the Studio Browser, and on the Omnis GitHub repo at: https://github.com/OmnisStudio. Search for Omnis-CheckRadio.

Round Check Boxes

The $buttonstyle property can be set to **kCheckBoxRound** to create a round check box. The style will create the maximum possible circle within the control's rectangle taking into account its height and width. The style animates the fore color of the circle when switching between true and false values (this can be turned off via $animateui).

The kCheckBoxRound button style supports a fill pattern or a transparent pattern set using $backpattern. If filled, the circle takes the colors from $forecolor and $secondaryforecolor for the checked (true) or unchecked (false) values, respectively. If transparent, the circle is not drawn and only the icon for the checked or unchecked state is displayed.

The round style uses $iconid and $secondaryiconid to control the icon displayed within the circle for the checked (true) or unchecked (false) values, respectively. Furthermore, when using themed SVG icons, you can use the $textcolor and $secondarytextcolor (for the checked and unchecked values, respectively) to set the color of the icons, which is white by default. For example, you could specify a green check icon when the checkbox is true and a red cross icon when the checkbox is false.

The default values for fore colors and text colors are inherited from the OS, for example, on macOS a round circle with nuanced blue is shown for true and a gray circle for false values alongside white icons.

Check List

Group	Icon	Name (type)	Description
Lists	☑	Check List	List with check box line selectio

The **Check list** displays a check box against each row in the list field. The user can select a line by checking the check box for a line. They are most suited to single columns of data or values in which the user can check multiple values or choices. If a check box is checked the corresponding line in the underlying list variable is selected. To create a check list, enter the name of your list variable in $dataname, and enter the $calculation for the list, or a variable name for a single column list.

Clock Control

Group	Icon	Name (type)	Description
Other	🕒	Clock Control (Xcomp)	Clock face showing the current t

The **Clock** control lets you show the current time in the current time zone on the client. There are many properties, most of them self-explanatory, that let you control the appearance. For example, you can specify whether the clock face is analog or digital using $digital and display it in 24 hour mode using $24hour, you can change the colors of the hour, minute, and second hands and the background color, you can set the time zone to be displayed using $timezone and $timezoneadj, and you can add an icon to the face using $iconface, $iconid, and $scaleicon.

The clock component passes 3 events detected in the $event() method for the object: evHoursChange, evMinutesChange, and evSecondChange which in turn pass the parameters pHours, pMinutes, and pSeconds respectively.

The analog and digital clock face let you show or hide the hours, minutes, and seconds. The $construct() method of the window is as follows:

```
# window contains instance vars iShowHours,iShowMins,iShowSecs (all Boolean type)
Calculate iShowHours as kTrue ## set the default settings
Calculate iShowMins as kTrue
Calculate iShowSecs as kFalse
Do method $setprops
# $setprops method sets the 2 clocks according to
# current check box settings
Calculate $cinst.$objs.clock.$showhours as iShowHours
Calculate $cinst.$objs.clock.$showminutes as iShowMins
Calculate $cinst.$objs.clock.$showseconds as iShowSecs
Calculate $cinst.$objs.digiclock.$showhours as iShowHours
Calculate $cinst.$objs.digiclock.$showminutes as iShowMins
Calculate $cinst.$objs.digiclock.$showseconds as iShowSecs
```

Each of the check box options on the window has the following $event() method:

```
On evClick
  Do method $setprops
```

Color Palette

Group	Icon	Name (type)	Description
Other	▭	Color Palette	Color picker allowing color selec

The **Color Palette** control, or Color Picker, allows the user to select a color. The type of palettes or color picker is specified in design mode using the $palusermodes property. The control has two methods:

- **$setpalmode()**
  $setpalmode(iMode) sets the mode of the palette; iMode can be an integer in the range 1 – 5, or an Omnis constants: kStandardPal=1, kRGBPal=2, kBrightnessPal=3, kDropperPa=4l, kUserPal=5

- **$colorind()**
  $colorind(iIndex,iMode[,iColor,bRedraw=kFalse]) sets or returns color selected in the palette; parameter iIndex is an integer 0-255, or -1 to get current color (you must use -1 if color not in 256 colors); iMode is 0 to get color, 1 to set color, or 2 to get current color index; iColor is color to set at index

The different values for iMode for $colorind() are:

iMode value	Description & parameters
0	(iIndex, iMode) gets the color (from the current color if iIndex==-1 or the color at index position iIndex)
1	(iIndex, iMode, iColor, bRedraw) sets the color (from the current color if iIndex==-1 or at index position iIndex) to iColor (and RGB) and redraws the control if (bRedraw)
2	(iIndex, iMode, iCnt, StartColor, EndColor, bRedraw) sets a gradient color range starting from iIndex, for iCnt iterations, using the colors StartColor ending at EndColor with optional redraw bRedraw
3	(iIndex, iMode) iIndex ignored, simply returns the currently selected cell index
4	(iIndex, iMode, iAppRef) if iIndex==0 it copies the control color table into the app iAppRef, else it copies the iAppref color table into the control
5	(iIndex, iMode, bIsAlpha) copies the default Omnis color table into the control, if bIsAlpha is set the color table will support alpha
6	(iIndex, iMode, bIsAlpha) copies the default Omnis color table into the control, if bIsAlpha is set the color table will support alpha: the color table will show the substitute transparent color option (small green T)
7	deprecated: embedded system colors are no longer supported
8	deprecated: embedded system colors are no longer supported
9	(iIndex, iMode, iNewIndex) sets the controls current index to iNewIndex
10	(iIndex, iMode, findColor) given the RGB color findColor, this search the colour table and sets the current table index if the color can be found
11	(iIndex, iMode, iColor) sets the initial color of the palette to iColor, rather than using kDefaultcolor; parameter 1 is unused and should be passed as 0

When the palette opens in RGB mode, the entry field has the focus, and selects all of the text. As you type, the color value updates. You can press return in one of the entry fields to set the color property, or Escape to close the color picker. In addition, when the focus is on one of the normal selection arrays, pressing an arrow key removes the mouse capture, so you can navigate using the arrow keys.

Combo Box

Group	Icon	Name (type)	Description
Lists	▤	Combo Box	Combined dropdown list and er

A **Combo Box** is a combination of dropdown list and an entry field. When viewed on an open window, you can choose an item from the list or type anything you want into the entry box. You might therefore put the salutations *Mr, Mrs, Ms, Miss* in the list, and leave the end user free to enter *Dr* or any other entry when needed. You can open a combo by clicking on the drop arrow, or tabbing to the field and pressing Alt plus the down arrow key; pressing just the down arrow key cycles through the choices in the dropdown list. Selecting a line with the mouse or pressing the Tab or Esc key closes the list.

When you create a combo box you must specify the name of the data field in the **$dataname** property and the name of your list variable in the **$listname** property. Enter a $calculation to format the list for display. The $contenttip property lets you add text to the field to help the end user understand what content should be entered into the entry part of the field. Combo boxes support $keyevents which means you can detect what character or function keys are typed into the entry part.

When using data from a list variable you should leave the $defaultlines property empty. Some of the properties of a combo box refer to the entry field part or the list part. For example, on the Appearance tab you can specify the number of lines displayed when your combo box drops down by setting the $listheight property.

The $bordericonstyle property allows you to add an icon to the left side of the field part of a Combo box; note that border icons cannot be shown on right side due to the drop arrow icon in the list (the $studioide property must be set to kTrue to display border icons).

The $contentpadding property allows you to add padding around the content inside the control.

Disabling the Automatic Search

By default, the content of the Combo box **list** is used to populate the edit field based on the content of the edit field when the dropdown list is opened. However, if the $disablesearchonopen property is set to true, the automatic search is disabled.

This property also applies to toolbar Combo boxes. For Data Grids, this property is used for columns with $columntype kDataGrid-ComboPicker.

Setting the focus

You can use $ctarget.$assign() to set the focus on a Combo box in a window toolbar. In versions prior to Studio 11, you could not programmatically set the focus on a toolbar combo box, but you can using $assign() and the following code to set the focus:

```
Do $ctarget.$assign($cinst.$toolbars.tCombo.$objs.combo) ## or
Do $ctarget.$assign($itoolbars.tCombo.$objs.combo)
```

In addition, $ctarget.$assign() works when the window has no current field, but this does not work during $construct for any field. Therefore, you can use *Queue set current field* for toolbar combo boxes (and on Windows, toolbar droplists). To do this, use the code

```
Queue set current field $cinst.$toolbars.t1.$objs.combo
```

which does work when executed during the window $construct.

Background theme on macOS

When the $backgroundtheme property is set to kBGThemeControl Combo boxes look and behave as they should on macOS.

Complex Grid

Group	Icon	Name (type)	Description
Lists	▦	Complex Grid	Grid that can contain other field controls

A **Complex Grid** can display multiple rows and columns of data taken from a list variable, but also allows data entry. To create a complex grid, you place a Complex Grid control on your window and drag other fields, including standard Entry fields, Droplists, or Check boxes, into the *row* and *header sections* of the complex grid field. A Complex grid is a *container field* having its own $objs and $bobjs object groups which contain the foreground and background objects inside the grid field. In addition, the $dividers group contains the dividers for the grid field.

Significant properties of complex grid fields, excluding the various header and border style/color properties, include:

- **$canclickvertheader**
  If true, the vertical header of the grid will accept clicks and highlight selected lines, provided that the $enterable property of the grid is kFalse
- **$canresizecolumns**
  If true, the user can use the mouse to resize the columns of the object
- **$canresizeheader**
  If true, the header on the complex grid can be resized
- **$canresizerows**
  One of the kResize... constants indicating how the user can use the mouse to resize the rows of the object
- **$enterable**
  If true, the grid is enterable
- **$extendable**
  If true, the grid automatically extends to allow the user to enter more lines
- **$firstsel**
  The number of the first selected character or line in the current contents; only when the grid is not enterable
- **$firstvis**
  The number of the first visible character or line in the contents
- **$lastsel**
  The number of the final selected character or line in the current contents; only when the grid is not enterable
- **$lastvis**
  The number of the final visible character or line in the contents
- **$multipleselect**
  If true, the complex grid allows the user to select more than one line
- **$showheader**
  If true, the grid has a header
- **$showhorzheader**
  If true, the grid has a scrollable horizontal header
- **$showvertheader**
  If true, the grid has a scrollable vertical header
- **$hiliteline**
  the grid lines highlight during drag and drop
- **$dropbetweenlines**
  the complex grid highlights between lines during drag and drop
- **$extendedgridlines**
  If kTrue, the grid lines of the final row extend to the base of the grid

Grid row focus alpha

When Complex grid rows receive the focus they are highlighted with an alpha version of the selection color. The amount of alpha used when hilighting the row is controlled using the appearance properties $gridfocusedrowalpha and $gridunfocusedrowalpha.

The highlight method used in versions prior to Studio 10.22 can be reinstated by setting the item **newSelectedRowDrawing** in the 'complexgrid' section of the config.json file to false.

Objects in a Complex Grid

Every field or object in a complex grid has the $gridsection property which tells you the section the object is in, and $gridcolumn which tells you its column. The fixed left-most column is column zero: the other columns are numbered from one and are separated by the dividers. Every field in the header is in column zero. The $top and $left properties of an object is relative to the top left-hand corner of its grid section.

Every field or object contained in a complex grid has the $container property which returns, in this case, the name of the grid field the object belongs to.

To insert a field in a complex grid from a method you use the notation

```
Do MyGrid.$objs.$add(section,column,type,top,left,height,width) Returns NewFieldRef
```

The “mingridpos” item in the “complexgrid” section of the config.json file specifies how close grid lines in a complex grid can be to one another. Zero allows them to overlay each other, giving the appearance of hidden columns. A positive value allows them to be further apart. A negative value is treated as zero.

Resizing Rows

The end user can change the height of the rows in a complex grid by dragging the row dividers. By default, dragging a row divider without pressing the Shift key resizes the single row above the row divider. To make all rows have the new row height, press the Shift key while dragging a row divider. This is the default behavior when the “shiftRequiredToResizeAllRows” item in the “complexgrid” section of config.json is set to true. If the item is set to false, the behavior is reversed; press the Shift key while dragging a row divider in order to resize the single row above the row divider. Dragging a row divider without pressing the Shift key gives all rows the new row height.

The **$getrowheight**([irow]) method returns the height of the specified row. If iRow is not specified, the height of the current row is returned.

Events for Complex Grids

Each contained field receives its normal event messages such as evClick, evBefore, evAfter, and the field event handler can pass these events to the $control() method contained in the complex grid.

Complex grids receive the events **evRowChanged** and **evExtend** which you can handle in the $event() method for the grid field. The evRowChanged event is sent whenever the user clicks in a different row and when the window is opened. The evExtend event is sent whenever a row is added to a grid with the **$extendable** property set. These events return the **pRow** event parameter which holds a reference to the row changed or the new row. Thus pRow.$line gives you the row number and pRow.ListColName returns the value of the cell. Note the events evCellChanging and evCellChanged are available for string and data grids only, not complex grids. The event **evClipChangedData** is sent to the field when the clipboard has changed the data in the grid.

The **evColumnDividerMoved** event is sent to a Complex Grid when a column divider has been dragged. The event has two parameters: pDivider, the divider column number (same ID as in $dividers), and pDividerMoveBy, the number of pixels the divider was moved by; note this can be negative. You can discard the event to stop the grid divider being moved.

Grid Field Exceptions

Generally, the properties of a complex grid apply to the whole grid or to a single row or column. However, you can set the properties of a single cell in the window instance by setting an *exception* for the grid cell. To do this you use the notation

```
Do MyGrid.$objs.fieldname.row.property.$assign(value)
```

For example, if you wanted to show cells in Grid1 column cBal in red if the value is negative, you could use the following code which runs when the user tabs out of the cell

```
On evAfter
  Calculate row as pRow.$line
  If cList.[row].cBal < 0
    Do $cinst.$objs.Grid.$objs.fBal.[row].$backcolor.$assign(kRed)
    Redraw {Grid}
```

You could use this event handler for the cBal field in the grid, but the interior fields could pass the events up to the $control() method in the complex grid field.

```
On evAfter
  Calculate row as pRow.$line
  If $cobj.$name = 'fBal' & cList.[row].cBal < 0
    Do $cobj.[row].$backcolor.$assign(kRed)
    Redraw {Grid}
```

You can clear exceptions using the $clearexceptions() method which acts on a cell, row, column or the whole grid, as follows

```
# clear all exceptions
Do $cinst.$objs.GridName.$clearexceptions()
# clear exceptions for a row
Do $cinst.$objs.GridName.$clearexceptions(RowNum)
# clear exceptions for a column
Do $cinst.$objs.GridName.$objs.FieldName.$clearexceptions()
# clear exceptions for a cell
Do $cinst.$objs.GridName.$objs.FieldName.RowNum.$clearexceptions()
```

You can attempt to set an exception for any property, although in practice this may not be satisfactory for some properties. Appearance properties, and button text for example should however all work as expected. For example, exceptions can be imposed for $width and $height for grid objects referenced via $obj and $bobj, however in this case, support for objects in $obj is limited to non-enterable tables only.

Sliding Columns

Complex grids can have Left or Right sliding columns. The $hasslideoutcolumn property controls the sliding columns on the complex grid, taking one of the following constants: kTableColumnsNormal, kTableColumnsLeftSlide, kTableColumnsRightSlide, or kTableColumnsLeftRightSlide.

The method $slideoutcolumn(kTableSlideDirection…, iRow=-1, kTableColumn…) sets iRow: if iRow=-1 the current row is hidden/shown/toggled, while kTableColumn… controls if the row’s left or right column slides out. The kTableSlideDirection… constants are: kTableSlideDirectionToggle, kTableSlideDirectionHide, or kTableSlideDirectionShow.

Complex grids support mouseover(kMHorzCell) returning the column clicked allowing you to detect clicks in slide out columns.

There is an example app to demonstrate Sliding Columns in complex grids in the **Samples** section of the **Hub** in the Studio Browser, and on the Omnis GitHub repo at: https://github.com/OmnisStudio. Search for Omnis-ComplexGridSlide.

Data Grid

Group	Icon	Name (type)	Description
Lists	▦	Data Grid	Grid to display text and numeric

The **Data Grid** component allows you to display text and numerical data and is very similar to the String grid: see String grid.

Droplist

Group	Icon	Name (type)	Description
Lists	☰	Droplist	Single column dropdown list

A **Droplist** (or Dropdown list) can contain a single column of data only. Most types of list can contain a single column, but some types such as list boxes, dropdown lists, combo boxes, or check box lists are best suited for displaying short single columns of data or are restricted to single column lists. The data inside a short list can come from your database loaded into a list variable, or you can include a number of lines that always appear in the list by default.

To create a Droplist list with default lines, add a *Droplist* control to your window and specify a list of values in the $defaultlines property. When you click on the droplist the default lines will appear. You can add default lines for dropdown, combo box, and tree list fields. Alternatively, you can assign a list variable to the $dataname property of the list field.

Droplists report the **evClick** event which you can detect in the $event() method for the control. This event returns the parameter pLineNumber containing the line number of the list row clicked.

The following example method is from a banking example; a droplist lets you select the account on the Transaction pane. The method behind the droplist loads the Account Id according to the selected line, assigns the stored values to the variables in the window instance, and redraws the fields on the client.

```
# $event() method for droplist
# iAccountList2 is the inst var for the droplist
On evClick
  Single file find on fAccounts.id (Exact match) {iAccountList2.[pLineNumber].2}
  Calculate iAccountName as fAccounts.type
  Calculate iTransactions as fAccounts.transactions
  Calculate iFinalBalance as fAccounts.balance
  Do $cinst.$senddata(iTransactions,iAccountName,iFinalBalance)
  Do $cinst.$objs.transaction.$redraw()
  Do $cinst.$objs.accounts.$redraw()
  Do $cinst.$objs.view_accname.$redraw()
  Do $cinst.$objs.final_balance.$redraw()
```

Background Theme

MacOS

The $backgroundtheme property controls how dropdown lists look and behave on macOS, as follows:

- If the theme is kBGThemeControl, and provided the list has <= 1000 lines, the dropdown list looks and behaves as a popup menu (the standard type of dropdown list on macOS).
- If the theme is kBGThemeWindow, the dropdown list behaves like a normal dropdown list, and has a standard themed macOS appearance. In the case where the list has 1001 lines or more, and theme kBGThemeControl, the look and behavior becomes that of kBGThemeWindow.

Windows

The color and fill pattern of a Droplist on Windows respects the $backgroundtheme property and only defaults to Windows system colors when the theme is set to kBGThemeControl. For other background themes it uses the appropriate colors and fill pattern, and for kBGThemeNone, it uses the foreground and background colors and fill pattern.

FishEye Control

Group	Icon	Name (type)	Description
Navigation	🖼	FishEye Control (Xcomp)	Displays a row or column of click icons

The **FishEye** control presents a row or column of icons to allow the end user to select an option by moving the mouse over the control and clicking on an icon. When the end user's mouse moves over the control, individual icons are enlarged and a text label is displayed for each icon.

Populating the FishEye

The contents for the FishEye control is provided by a list variable specified in its $dataname property. The list should have at least two columns, one for the ID of the icons to be displayed in the control, the other column for the text labels for the icons. You can set the $iconcolumn and $textcolumn properties to specify which columns are used for the icons and text, respectively. $iconcolumn is set to column 1 by default, so your text labels could be in column 2.

The icons can be added to the #ICONS system table in your library. To achieve the best magnification effect in the FishEye control, you should include the 48x48 version of each of the icons you wish to use.

The following method defines the list for a FishEye control and adds some lines; you would normally add multiple lines, which could be static data or loaded from a database.

```
# define iFishList (List), iFishIcon (Long int), iFishText (Char)
Do iFishList.$define(iFishIcon,iFishText)
Do iFishList.$add(1704+k48x48,"Window")
Do iFishList.$add(1719+k48x48,"Form")
Do iFishList.$add(1712+k48x48,"Object")
# and so on to build the list of options
```

Position and Expand direction

The $position property determines how the icons and text in the control expand when the mouse hovers over the control, and together with the $edgefloat property you can position the FishEye and set the direction in which it expands. $position can be set to one of the kFisheyePos... constants, and $edgefloat can be set to a kEFposn... constant. For example, with $position set to kFisheyePosBottom, and $edgefloat set to kEFposnLeftBottom, the FishEye control is positioned at the bottom of the window and the icons and text will expand upwards (as shown). You should experiment with the different $position and $edgefloat settings to achieve the effect and positioning you want.

Magnification and Stepping

The $magnification property specifies the factor by which the FishEye control increases the size of the icon under the mouse; it should be set to a real number between 1.0 and 16.0. The greater the number, the greater the magnification.

The number of icons that expand to the left and to the right of the icon under the mouse is specified by $steps; it should be an integer between 0 and 16. Therefore a low number, such as 0 or 1, will appear to make individual icons "popup" and you pass the mouse over the control, whereas a larger number will display a more gradual stepping effect. Again, you need to experiment with the $magnification and $steps properties.

The $magnifyall property controls whether or not all the icons are expanded; it is set to kFalse by default which produces its characteristic gradual stepping effect as the mouse moves over the control. If true, the items are only displayed when the mouse is over the control, and they all have a fixed square size ($magnification*$height for horizontal positions or $magnification*$width for vertical positions) and $steps is ignored.

Text Labels

You can specify the color of the text in $textforecolor. You can set the color and pattern for the background of the labels in $textbackcolor and $textpattern. In addition, you can specify the transparency of the label background by setting the $textalpha property; this is a numeric value in the range 0-255, where 0 is completely transparent and 255 is opaque.

FishEye Events

Clicking on an icon in the FishEye control generates an evClick event with pLineNumber set to the line number of the clicked entry. You could return the value of the label or some other value in another column in the list based on the line selected.

Graph2 Control

Group	Icon	Name (type)	Description
Graphs	◕	Graph2	Graph component with multiple types

The **Graph2** component is described in detail in Omnis Graphs chapter in the *Extending Omnis* manual.

Group Box and Scroll box

Group	Icon	Name (type)	Description
Containers	XY	Group Box	Groups other fields on your wir

A **Group box** and **Scroll box** let you group other fields on your window. You can create a Group box or Scroll box from the Component Store and drag other fields within their borders; so they do not contain or display data themselves, but they can contain other data fields and controls.

You can edit the label for a Group box in its $text property. Group and Scroll boxes can contain methods including a $event() method to detect events. Group and Scroll boxes are container fields so you can access the fields inside the box in your code using notation: see Container Fields.

You can apply a background or border color, or add a gradient pattern to scroll box fields. If you want to make entry fields within the scroll box appear to be transparent, you can apply a gradient to the scroll box, place the fields within the scroll box and set their background theme to parent.

A Group box or Scroll box can act as a *side panel:* see Side Panels under Page pane.

Headed List Box

Group	Icon	Name (type)	Description
Lists	▤	Headed List Box	List with button style headers

A **Headed List Box** displays data from a list variable in a table format. You can add button style headers to each column of the list which the user can click on to sort the data. You can also make the columns sizeable, and set other properties that control its appearance. A headed list can have a non-scrollable footer row, which could contain column totals, for example.

In addition to the general list box properties, such as $multipleselect, the headed list has the following properties

- **$dataname**
  the name of the list variable
- **$calculation**
  the calculation to format the columns for the list; you can use the con() function to concatenate multiple columns separated by the kTab character
- **$maxeditchars**
  the maximum size of the edit field for editing a column, or 0 if columns cannot be edited
- **$enableheader**
  if true the column headings act like buttons
- **$canresizeheader**
  if true the columns can be sized at runtime
- **$boldheader**
  if true the headings are bold
- **$showcolumnlines**
  if true the list draws lines between the columns at runtime
- **$designcols**
  the number of columns, maximum of 30 columns; you must set this property to the required number of columns before being able to set the $calculation and $columnnames with the appropriate column
- **$columnnames**
  a comma-separated list of heading text for the columns; you can set the column names in the dialog that opens when you click the property dropdown
- **$align**
  allows you to determine column alignment
- **$columnalignmode**
  provides runtime alignment control
- **$headerfillcolor**
  the color of the header; defaults to kColor3DFace
- **$headertextcolor**
  the color of the text in the header; defaults to kColorDefault, that is, the text color of the list
- **$colcount**
  a runtime only property, which is the number of columns in the headed list

- **$columnwidths**
  a runtime only property, allowing the column width to be set in a quoted comma-separated list of pixels values
- **$showheaderlines**
  If true (the default), header separator lines are drawn in the header
- **$hidefooter**
  If true, the footer row is hidden; set this to false to show the footer and set the contents using the $footer runtime property

The property $headerfontsize specifies the font size for the text in the header; if this is zero, the header text font size is the same as $fontsize. This means that Omnis can use a small font size for the header, like the Mac Finder.

You can use the style() function to style the text in the header button for each column specified in $columnnames. Note the styled text in $columnnames has to be assigned in $columnnames at runtime.

Entering the Calculation

The $calculation property for headed lists can be edited via a droplist button, which opens a dialog containing two tabs. The first tab has a grid allowing you to enter the variable name or calculation for each column. The second tab allows you to enter the long form of the calculation in the form *con(var1,sep,var2,sep,…),* where *sep* is either kTab or chr(9). You can use the Notation Helper or Catalog to enter variable names into this dialog.

Setting Column Widths

You can set the column widths for Headed lists (and string and data grids) using the runtime only property $columnwidths. This property returns a comma separated list of column widths in pixels. When you use the $assign() method to assign to this property, you must put the comma separated list in quotes. For example

```
# Item reference HeadedListRef set to headed list instance
Do HeadedListRef.$columnwidths returns HeadedCols
# returns something like ‘30,40,55’
Do HeadedListRef.$columnwidths.$assign('20,30,40')
# assigns the column widths to the headed list instance
```

In order to display a 16x16 icon in the header in a Headed List, you need to set a minimum column width of 21 pixels, to allow for the width of the icon and the padding Omnis adds to a column.

The **$resizecolumn** property specifies the column that is resized appropriately when the width of the control changes, such as when using $edgefloat properties to resize the list when the window size changes. A value of zero means no column is resized, but the last column extends if necessary.

The **$autosizecolumn(iColumn)** method can be used to resize the specified column in the headed list, based on the maximum width of the data in the column.

Text Alignment

The $align text property lets you set the alignment of all the columns in the list. At runtime, you can override the alignment for individual columns using the method

- $setcolumnalign()
  $setcolumnalign(columnNumber, alignment) sets the alignment to kLeftJst, kRightJst, or kCenterJst, and returns kTrue for success

and you can return the current alignment for a column using

- $getcolumnalign()
  $getcolumnalign(columnNumber) returns the alignment of the specified column

Note that headed list boxes do not support the style() function with type parameters kEscLTab, kEscCTab and kEscRTab.

The property $columnalignmode provides additional runtime control over $setcolumnalign(), and can have the following values: kAlignModeHeading, kAlignModeBody, kAlignModeAll and kAlignModeNone. These determine whether the heading, body, both or neither are affected by calls to $setcolumnalign(). Note that the call to $setcolumnalign() always stores the new alignment value in the list; $columnalignmode determines if the stored value is used. When the stored value is not used, $align determines the alignment.

When you create a headed list, you must set the $dataname property to the name of a list variable, and if necessary, enter the formatting expression in $calculation. You setup the column header text in the $columnnames property. You can specify the number of columns in the design object in $designcols. You can set the column widths by dragging columns with your mouse; shift-drag resizes the column to the right of the mouse pointer so you can use this for the last column. You can set $maxeditchars if the columns are to be editable.

You do not need a $calculation if the columns in the headed list are an exact mapping of the columns in the data list. If not, use the $calculation to format the columns in the headed list box. They must contain column names from your list and special column delimiters. You can use the *con()* function to format the calculation, and insert column delimiters using *chr(9),* the tab character. For example, to format three columns the calculation could be

```
con(Col1,chr(9),Col2,chr(9),Col3)
```

You can also use the *style()* function to change the style and color of specific columns. For example, to give Col1 a blue spot icon, make Col2 red and right-justified, and Col3 italic you would enter the following calculation

```
con(style(kEscBmp,1756),Col1,chr(9), style(kEscColor,kRed),Col2,chr(9), style(kEscStyle,kItalic),Col3)
```

Sorting Columns

The method $setsortcolumn(iColumnNumber,bDescending) specifies the current sort column and direction. This controls the drawing of the sort arrow in the heading.

Progress Bar

You can display a Progress Bar in a column inside a Headed List box control, for example, to indicate a percentage value. To enable a progress bar to be displayed, you can use the style() function and the **kEscBar** text style. When enabled, a progress bar with 1 or 2 segments is drawn across the whole column width, so no other content is allowed in the column.

kEscBar can take 3 or 5 parameters: the background color, segment 1 width (%), segment 1 color, then optionally segment 2 width (%), and segment 2 color.

For example, you can use the following in the calculation for a headed list column to draw a red segment of width iPercent % of the column width over a gray bar sized to the column width:

```
style(kEscBar,kGray,iPercent,kRed)
```

In this example, iPercent is a column value in the data list for the headed list.

Displaying Ellipses

Ellipses are shown when there is not enough space to display all the content in a headed list cell (also in the text for a tree list node); this applies for headed lists with more than one column. The **$disableellipsis** property allows you to disable ellipses for individual list fields, if required. This is in addition to the existing $clib.$prefs.$disableellipsis library preference which allows you to disable ellipses for all Headed Lists (and Tree Lists).

Tooltips

The length of text in tooltips for Headed list boxes is unlimited (note in versions prior to Studio 11, the length was limited to 255 characters). Although there is no limit imposed on the tooltip length, in practice the absolute maximum would be 32000, and the longest reasonable size to use would be around 2k.

Headed List events

The $event() method for a Headed List receives specific event messages. In addition to the general entry field events evClick, evDoubleClick, evAfter, evBefore, and the drag and drop events, there are specific events to report clicks on the column headers and when the list data is edited.

Column Headers

When the header is enabled by setting **$enableheader,** user clicks on the header buttons generate the evHeaderClick event with the column number held in pColumnNumber. evHeadedListHeadResize is generated immediately after a column has been resized.

Editing the List

When text editing is enabled by **$maxeditchars,** a headed list box receives three events in a specific order, together with parameters containing the list line, column number and new text entered.

- **evHeadedListEditStarting**
  with parameters pLineNumber, pColumnNumber, is sent on the first click in the selected cell which puts the cell into edit mode; discarding the event prevents editing
- **evHeadedListEditFinishing**
  with parameters pLineNumber, pColumnNumber, pNewText, is sent if the user enters a new value by hitting return or clicking away from the edit field; discarding the event leaves the field in edit mode, for example if pNewText is invalid. Note that you must store the new valid text in the list at this point: Omnis cannot do this since the data is a calculated expression
- **evHeadedListEditFinished**
  with parameters pLineNumber, pColumnNumber, is sent when the edit is completed

You could use the following event handlers for these events

```
# $event() method for the headed list box
On evHeadedListEditStarting
  If pColumnNumber=2 ## bar editing in this column
    OK message (Icon,Sound bell ) {Cannot edit this column}
    Quit event handler (Discard event)
  End If
On evHeadedListEditFinishing
  If pNewText=''
    Quit event handler (Discard event)
  Else
    Calculate cList.pLineNumber.pColumnNumber as pNewText
  End If
On evHeadedListEditFinished
  # do anything necessary here
```

Display Order Events

There is an event, evHeadedListDisplayOrderChanged, with an event parameter pDisplayOrder (containing the $displayorder, see below), which the headed list box receives when the display order has been changed using drag and drop.

Even after changing $displayorder, column numbers in all properties and events related to the headed list box are the original column number specified in the library in design mode. This means that changing $displayorder does not require changes to the methods that manipulate the headed list box.

Mouse Events

You can detect which column the mouse is over in a headed list. The function mouseover(kMHorzCell) returns the column number of the headed list if the mouse is over the headed list.

Editable text

You can allow the user to edit the text in a headed list box field. The $edittext(column number) method puts the field into text edit mode if there is a currently selected line, and the field is the current field. Editing is enabled by $maxeditchars.

Dynamic Column Headers

End users can reorder the columns in headed lists by clicking and dragging the column headers, as appropriate.

The headed list field has a runtime property called $displayorder which is a comma-separated list of column numbers, indicating the order in which columns are displayed by the headed list. Initially, $displayorder is set to 1,2,..., up to and including the value of $colcount, that is, the number of columns in the list. This is reset to the initial value whenever you change the number of columns. You can set this property using the notation, to reorder the columns.

In addition, headed lists have a property called $candragdisplayorder. If true, and $enableheader is also true, the user can drag and drop a column in the header to reorder the columns in the headed list.

When the user changes the order of the column headers, the column order stored in $displayorder is changed accordingly. If you allow the user to drag and drop in the column heading, you can then use $displayorder to save and restore the order the user has selected.

Footer Row

A headed list can have a footer row, which could contain column totals, for example, or any other data or text. Set the **$hidefooter** property to false to show a footer row at the bottom of the headed list (true/hidden by default); this is displayed in a fixed, non-scrollable bar under the scrollable part of the headed list.

The footer bar content is controlled using the **$footer** runtime property, which can be assigned a row variable. For example:

```
# iList (List), iCol1 (Int), iCol2 (Char), iRow (Row)
Do iList.$define(iCol1,iCol2)
Do iList.$add(1,2)
Do iList.$add(2,4)
Do iRow.$define(iCol1,iCol2)
Do iRow.$assigncols(iList.$totc(iCol1),iList.$totc(iCol2))
Do $cinst.$objs.List.$footer.$assign(iRow)
```

HelpMethods

The **HelpMethods** library contains various functions to implement a built-in Help system for your desktop applications.

Hyperlink Control

Group	Icon	Name (type)	Description
Navigation	LINK LINK LINK	Hyperlink Control (Xcomp)	List of hyperlink options

The **Hyperlink** control (or hyplinks) allows you to present a list of options to the end user, where each option in the list is a web-style hyperlink. The control is used in the Studio Browser, but you can use it in your own applications.

The options presented in the hyperlink control list are based on the contents of a list variable, which must be defined with three columns in the following format:

Col1	Col2	Col3
Group (Number)	Command (Number)	Name (Character)

The following method will construct a list for displaying in the hyperlink control.

```
Do hyperlist.$define(group,cmd,nam)
Do hyperlist.$add(1,1,"first entry - group 1")
Do hyperlist.$add(1,2,"second entry - group 1")
Do hyperlist.$add(1,3,"third entry - group 1")
Do hyperlist.$add(0,0,"")
Do hyperlist.$add(2,1,"first entry - group 2")
Do hyperlist.$add(2,2,"second entry - group 2")
Do hyperlist.$add(2,3,"third entry - group 2")
```

When the hyperlink control is clicked, the evLinkClicked event is generated with the pLinkgroup and pLinkid parameters. When the mouse enters or leaves the control the evMousePos event is generated with the pEntered parameter. The $event() method for the control can contain a method to respond to the selected link, for example:

```
On evLinkClicked
  Switch pLinkgroup
    Case 1 ## group 1
      Switch pLinkid
        Case 1
          # Do something...
        Case 2
        Case 3
      End Switch
    Case 2 ## group 2
      Switch pLinkid
        Case 1
          # Do something else...
        Case 2
        Case 3
      End Switch
  End Switch
```

The hyperlink control has the following properties.

- $topmargin and $leftmargin
  Number of pixels from top or left of control before links are shown
- $showarrows
  If true the scroll arrows are shown if the list of options/links extends beyond the border of the control
- $iconid
  iconid of image that can be used as a background skin for the control; if $scale is kTrue the icon is scaled to fit the control
- $hilitecolor
  Color of the text for the link under the mouse
- $extraspace
  Extra number of pixels used to separate links in the list
- $canceltimer
  (Runtime only) Releases mouse capture if the control is tracking the mouse. Sometimes needed on evLinkClicked, e.g. Do $cobj.$canceltimer.$assign(kTrue)
- $vertical
  if kTrue the links are shown in a vertical format (the default); when kFalse, the control displays a horizontal scrolling list of options

Separator Line

You can add a separator line in the list of options in the Hyperlink control; this only applies when $vertical is kTrue. To include a separator line, set the text in the $dataname list to a single - (hyphen) character; the line draws across the width of the control, inset by the left margin. The group id (in column 1) contains the color of the line; kColorDefault means use the IDE line color (as defined in appearance.json). For example:

```
Do ihlkSubList.$add(kColorDefault,0,-)
```

Icon Array

Group	Icon	Name (type)	Description
Lists		Icon Array	Displays list of items as clickable

The **Icon Array** allows you to display a list of items whereby each item is identified by a single icon. These choices are displayed as large or small icons which the user can click on or drag to select a number of items. Each icon also has a short text description which the user can edit, and you can add a button background. The data for an icon array is supplied from a list variable which contains the icon id and text label for each icon. The Studio Browser uses an icon array to display its large icon view.

There is a Standard field called 'Icon Array' as well as an External component (in the Deprecated group: see Deprecated Components). They behave in a similar way and have many properties in common. The standard built-in component is described here.

In addition to the general list field properties such as $multipleselect, the icon array has the following properties

- **$dataname**
  the list variable with at least two columns
- **$maxeditchars**
  the maximum size of the edit field, or 0 if the text cannot be edited
- **$smallicons**
  true for 16x16 icons, false for 48x48
- **$showtext**
  displays text labels
- **$buttonbackground**
  if true shows the icons on buttons
- **$smalltextwidth**
  the width in pixels of the text in small icon mode; must be at least 20
- **$hiliteline**
  if true lines highlight in single selection lists during drag and drop
- **$autoarrange**
  adjusts the number of icon columns when the field size changes
- **$enabledeletekey**
  allows the Delete key to delete the currently selected icons
- **$largetextwidth**
  the width in pixels of the text, when displaying large icons; to use the default width, set this property to zero
- **$extraspacing**
  the extra spacing in pixels added to each icon

Programming Icon Arrays

You must set up a list variable containing the data for your icon array. You can write event handling methods to respond to user clicks, and drag and drop in the field.

Setting up the List

You must define the list variable for an icon array with at least two columns, the first column for the icon id and the second column for the text label. You can use icons from the OmnisPIC.df1 or USERPIC.df1 data files, or #ICONS in your library. You can see the id numbers in the Icon Editor, which you can also use to add your own icons to USERPIC.df1 or #ICONS. You can define and build the list in the window $construct() method.

```
# declare variables IconId (Number 0dp), IconName (Character), and IconLIst (List)
Do IconList.$define{IconId,IconName}
Do IconList.$add(605,'Trash can')
Do IconList.$add(603,'Back arrow')
Do IconList.$add(601,'Pin')
# etc...
```

As a further refinement, you can set the $smallicons property to true and adjust the $smalltextwidth property to limit the length on the node text.

Editing in the Array

In addition to the general entry field events evClick, evDoubleClick, evAfter, evBefore, as well as the drag and drop events, there are specific events for editing an icon array list and for deleting selected lines if the delete key is enabled.

When text editing is enabled by $maxeditchars, the field receives three events in order, together with parameters holding the list line and the new text entered.

- evIconEditStarting
  with the parameter pLineNumber, is sent on the first click in the selected cell which puts the cell into edit mode; discarding the event prevents editing
- evIconEditFinishing
  with parameters pLineNumber and pNewText, is sent if the user enters a new value by hitting return or clicking away from the edit field; discarding the event leaves the field in edit mode, for example if pNewText is invalid
- evIconEditFinished
  with the parameter pLineNumber, is sent when the edit is completed

Handlers for these events might be as follows

```
On evIconEditStarting
  If pLineNumber<10
    OK message (Icon,Sound bell ) {Cannot edit these lines}
    Quit event handler (Discard event)
  End If
On evIconEditFinishing
  If pNewText=''
    Quit event handler (Discard event)
  End If
On evIconEditFinished
  # do anything necessary here
```

If the Delete key is enabled, two events are sent to the field:

- evIconDeleteStarting
  is sent to the field and Delete is pressed. Discarding the event prevents the delete occurring.
- evIconDeleteFinished
  is sent if the delete goes ahead, after all selected lines in the list have been deleted.

Editing the text in an Array

Icon arrays have the following method to allow end users to edit the contents of the field:

- $edittext()
  puts the field into text edit mode; if there is a currently selected line, the field is the current field, editing is enabled by $maxeditchars and $showtext are true

JPEG Control

Group	Icon	Name (type)	Description
Media	JPG	JPEG Control (Xcomp)	Displays JPEG images

The **JPEG** control lets you store and display JPEG graphics data. The image data is held in an instance variable (or row variable column) of Picture type specified in the $dataname property. The JPEG component reports no special events.

For example, you could use a simple window to display JPEG files chosen from a list. When the window is instantiated, a list of images is built and shown in a heading list. The window contains three instance variables: iImageList (List), iImageName (Char), and iImage (Picture). The window actually contains a large jpeg component and a small one to show a thumbnail image, but both fields are assigned the variable iImage.

```
# $construct() method of the window
Set main file {fImages}
Do iImageList.$define(fImages.Name)
Set current list iImageList
Build list from file
Do iImageList.$redefine(iImageName)
```

The $construct() method in the window builds the list of images, but the $event() method behind the heading list loads the selected image. The method loads the image from the Omnis database and assigns it to the instance variable; the large image and the thumbnail are then redrawn.

```
# $event() method for the heading list
On evClick
  Set main file {fImages}
  Do iImageList.$loadcols()
  Single file find on fImages.Name (Exact match) {iImageName}
  If flag true
    Calculate iImage as fImages.Image
    Do $cinst.$objs.thumb.$redraw()
    Do $cinst.$objs.bigimage.$redraw()
  End If
```

The $noscale property specifies whether or not the image is scaled; when set to kTrue, the true height, width, and proportions of the image are maintained, when kFalse, the image is scaled to fit the size and shape of the component itself. The JPEG component has one or two additional properties under the Custom tab in the Property Manager; some of these you can set in design mode, while others are runtime properties. When set to kTrue, $palette specifies that the image uses the color palette stored with the image. $imageheight and $imagewidth contain the height and width of the current image data. $fast specifies that faster though less accurate processing of the image occurs. $nosmooth disables smoothing of the image; when set to kFalse, smoothing occurs which smooths hard edges in the image and decreases the file size. When set to kTrue, the $allowclipboard property lets the user paste an image into the field from the clipboard on the client.

The $writejpeg(cFilename) method writes a JPEG file to the database from the current image data with the filename and path specified in cFilename.

Labeled Fields

Group	Icon	Name (type)	Description
Labeled Fields	Ab	Labeled Fields	Fields and label combined into c

The **Labeled Entry Field** and **Labeled Masked Entry Field** are compound fields combining the respective standard entry field type with a Label object, suitable for creating an entry form containing multiple fields: see Single Line Entry Field and Masked Entry Field for details about these field types.

List Box

Group	Icon	Name (type)	Description
Lists	▤	List Box	Displays list variable contents

The **List box** control lets you display a single column list of data or more complex tabular data in a list format. Omnis lists can display up to 400 columns and an unlimited number of rows. For short, single-column lists you can use the $defaultlines property, otherwise the data for a list field would be constructed in a list variable. The List Programming chapter describes in detail how you can build data using Omnis list variables and methods.

When you create a list box, you must enter the name of your list variable in the $dataname property of the list box field. You must also enter a string calculation into the $calculation property to format the data into columns. The calculation should be in the format:

```
jst(column1,width,column2,width,...)
```

The $calculation for a list box field specifies the column(s) to appear in the field and the width of each column as the number characters, using the *jst()* function. You can include any number of columns from your list variable in your list field, and in any order.

The jst() function

The calculation for your list field can use the *jst()* function to set the column names and their widths. For example, you could enter

```
jst(NAME,10,SALES,5)
```

This calculation puts the NAME variable in a column 10 characters wide and the SALES variable in a column 5 characters wide.

You can include the X modifier to truncate the data in a particular column. For example

```
jst(COMPANY, '20X', NAME, '22X')
```

This calculation puts the COMPANY variable in the first column and truncates any data that is longer than 20 characters. It then puts the NAME variable in the second column and truncates any data that is longer than 22 characters.

You can right justify a column, perhaps one containing numbers. For example

```
jst(NAME, ‘20X’, TEL, -18)
```

This expression left justifies the NAME variable in the first column which is 20 characters wide, and right justifies the value of the TEL variable in a column 18 characters wide.

Note that for single-column lists you need only include a single field or variable name in the $calculation property, that is, you don’t need to use the *jst()* function. If you omit the calculation altogether, no data will appear in your field.

You may need to adjust the column widths or the text properties. For example, you should use a non-proportional font such as Courier if you are using two or more columns in a list box. This ensures the columns line up across the list.

When you select a line in a list box, this normally deselects all other lines, but with the $multipleselect property set, you can select any number of lines. Dragging or shift-clicking selects contiguous lines, while Ctrl/Cmnd-clicking selects non-contiguous lines. You can deselect individual lines with Ctrl/Cmnd-click or all lines by clicking in the white space at the end of the list.

If you open your window and the list or grid field is empty, this probably means that either the list variable behind the field is empty, or you have not entered the $dataname or $calculation property correctly.

Queue Click

The pLineNumber event parameter is set for evClick and evDoubleClick events when generated via *Queue click* or *Queue double click* to a list or list sub-class.

Searching Lists

In versions of Omnis prior to Omnis Studio 5, you were able to tab into or click on a list, such as a list box or headed list field, and search the first column of the list by typing a few characters. The focus in the list would jump to the line containing the characters you typed. In this case, Omnis stored the characters you typed into a search buffer, regardless of the delay between each character you typed, and tried to find a match in the list. You could continue to type extra characters and Omnis would add these to the search buffer. In addition, you could use the + and – keys to find the next and previous matches in the list, and you could use * to represent wildcards in your search.

The old list searching behavior is enabled only when $oldlistsearching is set to kTrue (the property is an Omnis root preference). By default, $oldlistsearching is set to kFalse meaning that the old list searching behavior no longer works in the various list fields, including list boxes and headed list fields. You can still search in a list field by typing a few characters, but if there is a delay in your typing the search buffer is reset and you are able to type another string to search the list again. The +, - and * keys (plus, minus, and asterisk) are treated as normal search characters, rather than having a special function.

List Line Colors

The “alternatelinecolorplatforms” color setting in the appearance.json file allows you to enable alternating colors for list lines. This option is an integer that indicates the platforms on which the odd and even list row colors are used for relevant lists with background theme kBGThemeControl. The values are: 0 for no platforms, 1 for macOS (the default), 2 for Windows, 3 for macOS and Windows.

Selected List Line Colors

The $selectiontextcolor and $selectionbackcolor properties specify the color of the text and background of the selected list lines; these properties apply to standard List boxes, as well as Icon arrays, Headed lists, Checkbox lists, Complex grids, and Tree lists.

The $selectionbackcolor property is the back color of selected lines; kColorDefault means use the default color. When not kColorDefault the specified color only applies when the control has the focus. The $selectiontextcolor property is the text color of selected lines; kColorDefault means use the default color. When not kColorDefault applies irrespective of whether the control has the focus.

Line Background Colors

The $linebackgroundcol property specifies the column number in the data list for the control ($dataname) that contains color values that override the default background color of each line; the value zero or kColorDefault in this column means the normal background color for the line is used. (This property is also available Headed Lists and Check box lists.)

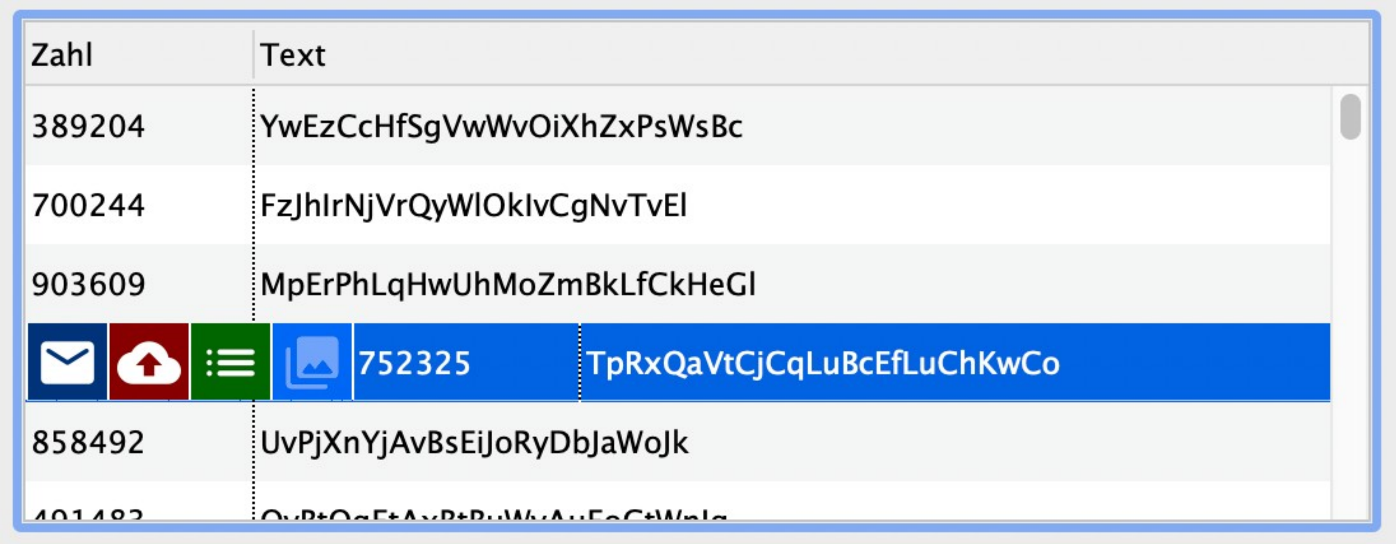
Background theme on macOS

Lists with a background theme of kBGThemeControl draw their line background using alternating colors, like the Mac Finder. Lists also display selected lines in the same way as the Mac Finder.

List Row Buttons

You can add a set of buttons to the **left** and/or **right** side of a row in a standard List or Headed List control. The buttons would typically act on the data in the row, such as opening another window to edit the row data, or deleting the row. There is an example app called **Row Buttons** in the **Samples** section of the **Hub** in the Studio Browser.

The row buttons slide out as the mouse enters the left or right side of the current row in the list, or as the Shift+Control+Left or Right arrow keys are pressed when a line in the list is selected. Selecting a button closes the whole row of buttons, and the name of the selected button is sent to the event method for the list. Row buttons work best with a larger row height (font size for the list), or when $linehtextra is set, which allows you to add extra pixels to the height of each line in a List box (or Headed list).



Zahl	Text
389204	YwEzCcHfSgVwWvOiXhZxPsWsBc
700244	FzJhIrNjVrQyWlOkIvCgNvTvEl
903609	MpErPhLqHwUhMoZmBkLfCkHeGl
752325	TpRxQaVtCjCqLuBcEfLuChKwCo
858492	UvPjXnYjAvBsEiJoRyDbJaWoJk

Figure 145:

Adding Row Buttons

List controls (and Headed lists) have the **$rowbuttons** property which stores the definition for the left and right buttons in the list row. When you select the property in the Property Manager in design mode a dialog opens allowing you to specify the buttons for the row. The button definition stored in $rowbuttons will be applied to every row in the list.

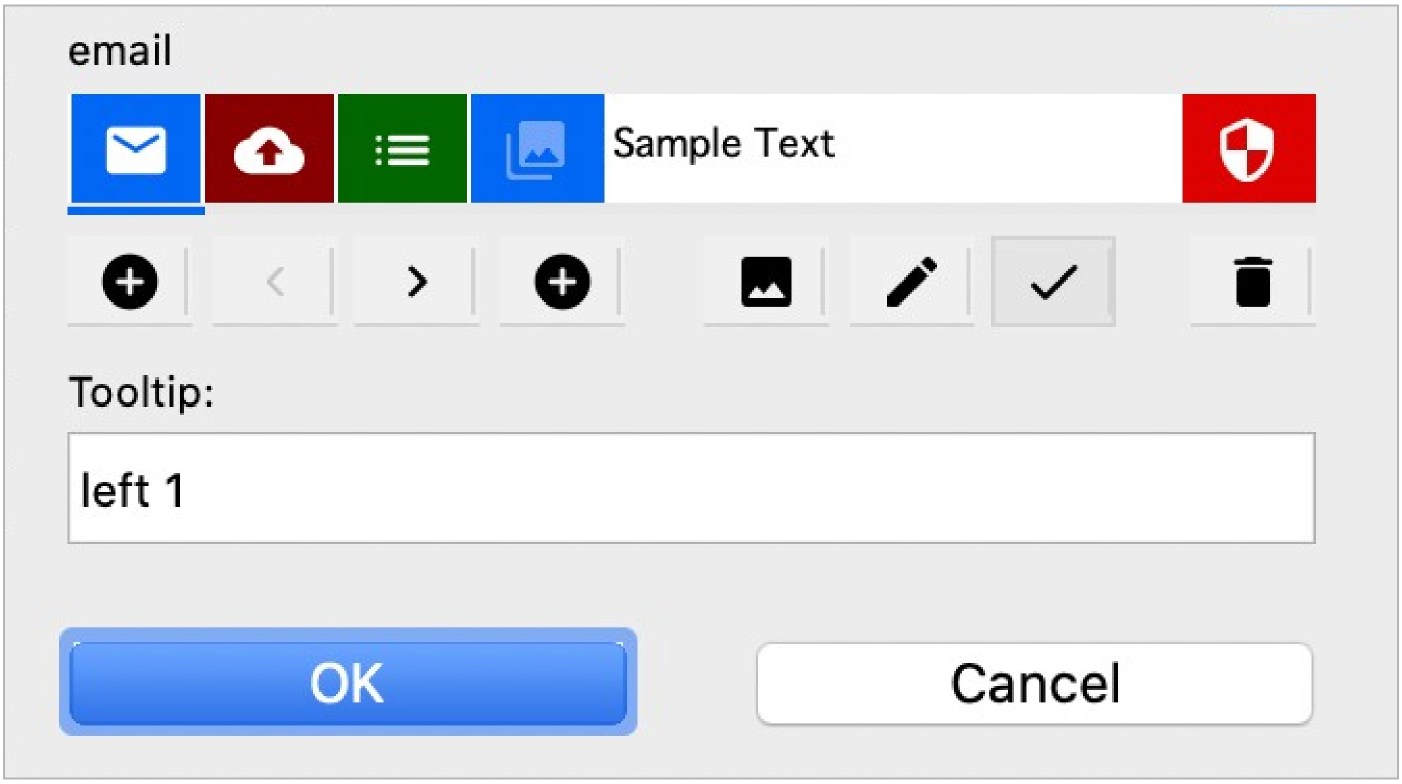


Figure 146:

In the edit dialog, you can add a left or right button using the left or right + icons. Clicking on a button makes it current (shown underlined) allowing you to change its attributes. You can move a button in the row order using the left and right arrow buttons.

The image icon allows you to add an icon, which is an SVG image from an icon set (PNG is not supported); the name of the icon becomes the name of the button, which is shown in the top-left of the dialog, and used in the notation to reference the button. The pencil icon allows you to set the background color of the button. The check icon allows you to disable the button. You can also add a tooltip for a button.

You can delete a button by selecting it and clicking the trash icon.

Setting Row Buttons

The **$setrowbuttons** method allows you to specify the left or right buttons for a list row at runtime, giving you more control over the buttons for individual list rows; for example, you can call this on the evClick event for the list.

The definition for $setrowbutton is as follows:

- $setrowbutton(*bLeftIcon*,[*cIcnIDName, iTintColor, bDisabled, cTooltip*]) adds a row button.
  bLeftIcon: kTrue for a left button, kFalse for a right button
  cIcnIDName: the name of an SVG icon for the button
  iTintColor: color of the button background
  bDisabled: kTrue if the button is disabled
  cTooltip: the tooltip for the button

You can clear the whole row of buttons by sending *bLeftIcon* as either kTrue or kFalse without any further parameters.

```
Do $cobj.$setrowbuttons(kTrue) ## clears the left buttons
Do $cobj.$setrowbuttons(kFalse) ## clears the right buttons
```

The following will add a button:

```
# adds a button only for row 2
If $cobj.$line=2
  Do $cobj.$setrowbuttons(kTrue,"language",kDarkRed)
End If
```

Events

The evRowButtonClicked event is sent to the list control when a row button is selected, and pRowButton will contain the icon name of the button clicked.

```
On evRowButtonClicked
  If pRowButton="language"
    # discards the event for a button with the icon 'language'
    Quit event handler (Discard event)
  End If
  # process the button click
```

Marquee Control

Group	Icon	Name (type)	Description
Other	TEXT	Marquee Control (Xcomp)	Displays scrolling text

The **Marquee** control lets you display continuously scrolling text areas in a window; you could use Marquee for news headlines or stock prices, or anything that needs to grab the user's attention.

You can enter the text for the marquee object in the $message property. You can use the style() function to embed icons and colors in the scrolling text message.

You can set the text and background color using $textcolor and $backcolor, and set up the font using $font and $fontsize. You can set the $speed of the scrolling message (the lower the value, the slower the scrolling) and you can set the $steps which controls how much the message jumps while scrolling. You can scroll the marquee in the opposite direction by specifying a negative value for $steps.

Masked Entry Field

Group	Icon	Name (type)	Description
Entry Fields	###	Masked Entry Field	Entry field with 'mask' to format

The **Masked Entry Field** is the same as a Standard Entry field except that it has additional properties that control or 'mask' the format of the data entered into the field.

Property	Description
$formatstring	data entry formatting string for the field
$formatmode	type of formatting string for the field, either Character, Number, Date, Boolean
$inputmask	input mask for the field

The $formatstring property stores a set of characters or symbols that formats the data in a masked entry field for display, regardless of how the data is stored. The $inputmask property contains a string that formats data as the user enters it into a field. When a user enters data into a field controlled by an input mask, Omnis rejects any characters that do not conform to the format you've specified in the mask.

To enter a format string for a masked entry field, you need to specify the type of data represented in the field by setting its $formatmode property: you can set this to Character, Number, Date, or Boolean. You can enter a format string manually or use one from the dropdown list in the $formatstring property in the Property Manager; the default formats in this dropdown are stored in a range of system tables: #TFORMS, #NFORMS, #DFORMS, #BFORMS. The symbols you can use in $formatstring are described in the Format Strings and Input Masks section in the *Window Programming* chapter.

Modify Report Field

Group	Icon	Name (type)	Description
Reports	🖨	Modify Report Field	Embeds a report class in a wind

A **Modify Report Field** lets you display a report class on an open window. This allows end users to change certain aspects of the report class at runtime, including the height of the Record section, the contents of headers and footers, the position and color of graphics on the report, and so on. When you create a modify report field you specify the name of the report class in the $classname property of the field. Any changes made to the report class in the Modify report field and saved in the class, and similarly, any changes made directly to the report class in your library are visible in the Modify report field when it is next opened.

You can hide or show the outline of the paper and the rulers with $showpaper and $showrulers. You can hide or show the current or all connections for associated report sections with $showcurconns and $showallconns, and you can set the width of the connections shown in the left margin by setting $connswidth. You can show the report sections as narrow lines by enabling the $shownarrowsections property. You can also change these properties at runtime. The $disablesystemfocus property let you disable the system focus indicator in the field.

To make the modify report field fill the entire window you can set its $edgefloat property to kEFposnClient.

Along with the common $redraw() method for a field, an instance of a report modify field has the methods $sortfields() which opens the Omnis Sort fields dialog for the report contained in the field, and $pagesetup() which opens the standard Page Setup dialog.

A modify report field generates an evSelectionChanged event which you can detect in the $event() method for the window field.

Applying changes to selected objects

To change individual objects inside a modify report field at runtime you need to set its $applyselected property. When the $applyselected property is set to kTrue any property changes you direct at the modify report field, such as font and appearance changes, apply to the currently selected object inside the modify report field. For example, a window could contain a modify report field (its $classname property is set to contain a simple summary style report that lists data from a Customers file), a button, and an instance variable that stores a reference to the modify report field.

The pushbutton contains the following $event() method. Note that the variable iModReportField stores a reference to the modify report field on the open window.

```
On evClick
Do iModReport.$applyselected.$assign(kTrue)
Do iModReport.$textcolor.$assign(kRed)
Do iModReport.$applyselected.$assign(kFalse)
```

When you open such a window, select an object inside the modify report field, and click on the pushbutton, the method changes the text color of the currently selected object to red. Note that you have to set the $applyselected property to kFalse when you have finished your changes.

Font and Color Tools

Rather than using pushbuttons to change a Modify Report Field as above, you can create your own set of toolbars and install them in your window containing the report field. The Component Store contains a number of toolbar controls and pickers for setting fonts, lines, and colors that you can use with the modify report field. A further example will demonstrate using toolbars with the modify report field.

You can create a toolbar class containing the appropriate font, line, and color pickers, add suitable icons from the icon data file or #ICONS (or use the default ones), and add it to your window. Each toolbar control would contain a method that applies the current settings from the control to the selected object in the modify report field. The following method is for a font list control on the toolbar.

```
# method contains item reference var iModFieldRef set to

$iwindows.ModReportFieldWin.$objs.ModReportField
On evClick
  Do iModFieldRef.$applyselected.$assign(kTrue)
  Do iModFieldRef.$font.$assign($cinst.$objs.FontList.$contents)
  Do iModFieldRef.$applyselected.$assign(kFalse)
```

Note that the toolbar class also contains an instance variable of type Item reference that stores a reference to the modify report field on the open window, and that the method sets $applyselected. The methods behind the other tools on the toolbar are very similar; here's the method for a color picker:

```
# $event() method for forecolor picker control
On evClick
  Do iModFieldRef.$applyselected.$assign(kTrue)
  Do iModFieldRef.$forecolor.$assign($cobj.$contents)
  Do iModFieldRef.$applyselected.$assign(kFalse)
```

Note that the current selection in a picker control is returned in its $contents property, therefore as the user makes a selection you can use $cobj.$contents to return the value.

Graphics Tools

At runtime, a modify report field has the $tool property which you can set to allow users to place graphics or background objects on your report; you cannot add fields and other foreground objects to a modify report field. You can create another toolbar that uses the $tool property and add it to your window. The toolbar class can contain various button controls, with suitable icons from the icon data file or #ICONS.

Each button in your toolbar class contains a single method that assigns to the $tool property and switches the cursor to the appropriate tool. For example, a 3D Rectangle button could contain the following method; note that the toolbar class also contains an instance variable of type Item reference that stores a reference to the modify report field on the open window.

```
# method contains item reference var iModFieldRef set to $iwindows.ModReportFieldWin.$objs.ModReportField
Do iModFieldRef.$tool.$assign(kRect3D)
```

When you open this window, select one of the tools, and move the cursor over the modify report field, the cursor changes to a cross-hair. The user can draw objects on the modify report field which are saved to the underlying report class automatically.

To use the modify report field to its fullest potential you need to build a number of toolbars that allow the user to change every aspect of the report class, including margins, page setup, sort fields, as well as the color and style of objects on the report. The modify report field is used extensively in the Ad hoc report library supplied with Omnis.

Multibutton Control

Group	Icon	Name (type)	Description
Buttons		Multibutton Control (Xcomp)	A round, animated popout butto opens to show a number of addi options

The **Multibutton** control provides a round, animated popout button that opens to show a number of additional options, each represented by an icon. The button reports the evButtonClicked event with the pButtonid parameter being the selected button. The following image shows the closed state (left) and open state of a multibutton, in this case opening to the right:

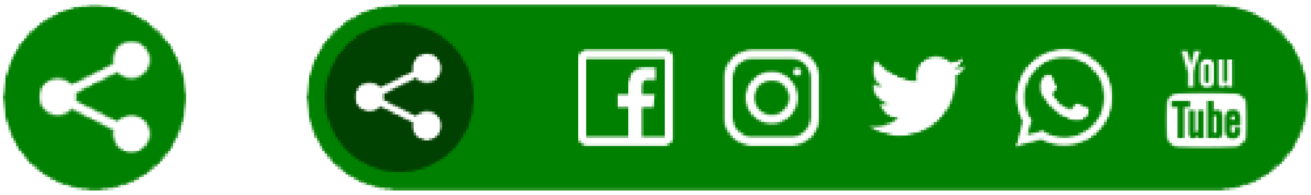


Figure 147:

The Multibutton Control is in the **Buttons** group in the Component Store (it is an External Component but is pre-loaded). The Multibutton Control has the following properties (shown on the Custom tab in the Property Manager):

- **$buttoncolor**
  The background color of the control
- **$buttonopen**
  kTrue if the control is open
- **$expanddirection**
  the direction the control expands, a constant: kMBexpandRight, kMBexpandLeft, or kMBexpandCenter
- **$iconstr**
  comma separated list of icon ids that are displayed when the control is opened, the number of icons determines the number of options; you can provide icons with a transparent background so the background color is seen; see below
- **$openicon**
  the ID of the icon shown to 'open' the control; this is shown in the closed state and can be a different icon as those displayed in the popped out list of buttons
- **$closeicon**
  the ID of the icon shown to 'close' the control; this replaces the icon specified in $openicon

The multibuton reports the **evButtonClicked** so you can use this in the $event method for the control and test the value of pButtonID which is the id of the selected button starting at 1 for the first button in the popped out list of buttons.

Specifying Icons

In general, you should use SVG images for button icons to achieve good scaling of icon images. The $iconstr property is a comma separated list of icon ids that are displayed when the Multibutton control is opened, that also specifies the number of button options in the control.

You can use PNG icons for the Multibutton control, and when using $iconstr the icon id for PNG icons can use the form <id>x<size> where size is one of the available standard icon image sizes, that is, 16, 32, or 48. For example, 2033x48 can be used to specify the bitmap icon with ID 2033 and its 48x48 version. If no size is specified, then the default icon size is used.

Multi Line Entry Field

Group	Icon	Name (type)	Description
Entry Fields	Abc Xyz	Multi Line Entry Field	Entry field allowing multiple line scroll bars

The **Multi Line Entry Field** is the same as a standard, Single Line Entry Field except that it has multiple lines and a scroll bar allowing the end user to enter larger amounts of text. See the description for the Single Line Entry Field for details of entry field properties.

Limiting Line count

The runtime-only property, $linecount, allows you to limit the number of lines of text/data that can be entered into a multi-line field. For example, setting $linecount to 2 would only allow 2 lines of text to be entered into the field.

The following example code for the $event method of a multi-line edit field shows how you could only allow two lines of text to be entered by setting iMaxLines to 2:

```
# set up variable iMaxLines (Integer)
On evClipChangedData,evKey
  Process event and continue
  If $cobj.$linecount>iMaxLines
    Calculate $cobj.$contents as iPrevData
    Sound bell
    Quit event handler
  End If
  Calculate iPrevData as $cobj.$contents
```

Navigation Menu

Group	Icon	Name (type)	Description
Navigation	☰	Navigation Menu (Xcomp)	Cascading menu with images ar options

The **Navigation Menu** is an external component that allows you to build interactive cascading menus within your windows, providing a navigation method similar to that found on some websites. The component is available for Window classes and for the JavaScript client, and they are more or less identical, so please refer to the description for the JavaScript Nav Menu Control for details. (Please note that there is an example for window classes for this component in the Navigation Menu app in the JavaScript Component Gallery on the Omnis website.)

OBrowser

Group	Icon	Name (type)	Description
Media	🌐	OBrowser (Xcomp)	Embed a web page into a windo enables HTML controls

The **OBrowser** external component allows you to embed web pages into your thick client windows, as well as providing the ability to add custom HTML controls to window classes. Even though the OBrowser control is a fully featured web browser, it is really only intended for targeted use with specific web pages, rather than use as a general web browser (you should note that there is no sandbox support). You could use it in your application, for example, to present the end user with information in a web-style layout, such as a Help system or FAQ, or you could embed a landing page that is hosted on your website.

OBrowser is available for both Windows and macOS and both platforms use the **Chromium Embedded Framework** (CEF) as the underlying browser which provides good support for HTML5 and CSS3. OBrowser supports the standard CEF configuration settings using the cefSwitches item within the Omnis config.json file.

To add an OBrowser control to your window class, locate the **OBrowser Control** in the *Media* group in the Components Store and drag it onto your window.

Setting web pages

To navigate to a page in the OBrowser Component, set the property $urlorcontrolname to a full URL with a prefix either http://, https:// or file://, such as 'https://www.omnis.net'. Note that in design mode, OBrowser will navigate to the page (assuming there is a network connection if required), but the page will not respond to clicks, keyboard input etc.

Once a page is open in a runtime window, the user can interact with the page as would be expected, although an attempt to open a popup window will fail. The value of the $urlorcontrolname property does not change while the user navigates through pages. Instead, the read-only property $currenturl contains the URL of the page currently open.

Two more read-only properties provide more state information:

- **$cangoforward**
  If true, you can use $forward() to move to the next URL in the browser back-forward list.
- **$cangoback**
  If true, you can use $back() to move to the previous URL in the browser back-forward list.

The **$contextmenuremovebackforward** property allows you to hide or show the **Back** and **Forward** navigation items from the context menu.

Methods

OBrowser has the following methods related to using it as a web browser:

Method	Description
$forward()	$forward() navigates forwards to the next URL in the browser back-forward list. Returns a Boolean, true for success
$back()	$back() navigates backwards to the previous URL in the browser back-forward list. Returns a Boolean, true for success
$reload()	$reload() reloads the currently displayed URL. Note that if you want to re-open the assigned $urlorcontrolname, you need to re-assign the original value to $urlorcontrolname.
$startdownload()	$startdownload(iDownloadId,cDestPath) starts the file download with id iDownloadId, storing the file at cDestPath. You must execute either $canceldownload() or$startdownload() in response to the evBrowserStartDownload event. Typically you would prompt for a file path, and then call $startdownload().
$canceldownload()	$canceldownload(iDownloadId) cancels the file download with the specified iDownloadId. You can call this in response to evBrowserStartDownload, to cancel the attempted download. You can also call this any time between calling $startdownload() and receiving evBrowserFinishedDownload.
$setdataurl()	$setdataurl(vData,cMediaType[,cWidth,cHeight]) assigns a data URL for the supplied data, with the specified media type, to $urlorcontrolname.vData can be either:binary - the data represented by the URLor another type - OBrowser converts the data to character if necessary, and then UTF-8, to become the data represented by the URL.cMediaType is a MIME type specifying the type of the data e.g. text/plain or image/png.cWidth and cHeight are optional. They are only relevant if vData is binary, and the data is an image. In this case, the data URL represents an img of the specified cMediaType, sized using the CSS sizes width and height e.g.$setdataurl(image, “image/png”, ”, ”)

Events

OBrowser generates various events, described in the following sections.

evBrowserLoadStateChange

Sent to the control when it starts or ends loading its content. This event has one event parameter in addition to pEventCode:

Parameter	Description
pLoading	If true, the control is loading content.

evBrowserFrameLoadError

Sent to the control when an error occurs while it is loading a frame. This event has three event parameters in addition to pEventCode:

Parameter	Description
pUrl	The URL being loaded
pFrame	The browser frame. Empty means the main frame

Parameter	Description
pErrorText	Text describing the error

evBrowserOpenUrl

As mentioned earlier, an attempt to open a popup window will fail. This event is sent to the control when a navigation action by the user wants to open a URL in a new browser window. This event has one event parameter in addition to pEventCode:

Parameter	Description
pUrl	The URL for which opening a popup will fail

evBrowserStartDownload

Sent to the control before starting a file download. Your code must respond by calling one of two OBrowser methods described later: $startdownload() or $canceldownload(). This event has four event parameters in addition to pEventCode:

Parameter	Description
pDownloadId	An integer that identifies this download request
pSuggestedName	The suggested name for the file
pMIMEType	The MIME type of the file
pUrl	The URL of the file to be downloaded

evBrowserDownloadProgress

Sent to the control periodically while a download is in progress. This event has three event parameters in addition to pEventCode:

Parameter	Description
pDownloadId	An integer that identifies this download request
pTotalBytesExpected	The total number of content bytes expected. -1 if the total is unknown
pBytesReceived	The number of content bytes received so far

evBrowserFinishedDownload

Sent to the control when a download has finished. This event has two event parameters in addition to pEventCode:

Parameter	Description
pDownloadId	An integer that identifies this download request
pErrorText	Either empty, meaning the download completed successfully, or error text describing why the download failed

OBrowser Configuration

The OBrowser object has a section in Omnis configuration file (config.json) named "obrowser". This has entries as follows:

Entry	Description
cefSwitches	An array of character strings. Each string in the array is a switch passed to CEF when initialising CEF, e.g. –disable-logging
clearCacheWhenLoaded	Boolean. True if the CEF cache is cleared when OBrowser is first loaded. The cache is in a sub-folder of the Omnis data folder:`chromiumembedded\\cache`
clearLocalStorageWhenClearingCache	Boolean. True if HTML5 local storage is cleared while clearing the cache.

Entry	Description
defaultHtmlcontrolsFolderInDataFolder	macOS only. Boolean. Default is false. If you set this to true, Omnis looks for the htmlcontrols folder in the data folder rather than the program folder (provided that the htmlcontrolsFolder member is absent or empty). This allows you to add your own HTML controls and the Omnis Runtime app to remain code signed.
htmlcontrolsFolder	Character string. By default, the HTML controls are located in the htmlcontrols folder in the Omnis program folder. You can override this by providing a full pathname in this entry.
htmlControlPort	Omnis assigns the port for the OBrowser control dynamically, so it is generally not necessary to set the port for OBrowser
locale	The locale to be used for CEF (the framework used by the obrowser external component to provide its functionality). Defaults to empty, meaning use the locale (as returned by the Omnis locale() function) of the Omnis program. If not empty, it must be a locale string that can be used to set the locale of CEF e.g. it_IT. This affects for example the context menus displayed for HTML pages rendered by obrowser. See below
logSeverity	Integer. CEF log level:0: Default logging1: Verbose logging2: Info logging3: Warning logging4: Error logging99: Logging disabledWhen logging is enabled, the log is written a file in a sub-folder of the Omnis data folder:`chromiumembedded\\log\\cef.log`
messageTimeout	Integer. The timeout in tics (1/60th second units) for synchronous communication between OBrowser and the HTML control. Not all communication is synchronous, but certain messages (e.g. get data) need to be. This defaults to 60 (a second, which should be more than enough). When you are debugging your control you may want to make this much larger, to give yourself time in the debugger.
remoteDebuggingPort	Integer. The TCP/IP port number on which the debugger listens, set to 5989 by default. Set this to zero to disable debugging.
candebug	macOS only. Boolean. If true, context menus for the control allow you to open the developer tools.
useOmnisTraceLogForConsole	Boolean. Default true. If true, console messages from pages hosted by obrowser are redirected to the Omnis trace log.

Drag and Drop in oBrowser

You can drop the URL content from one OBrowser control to another OBrowser control, as well as drop that content onto a Picture field. In this case, the URL of the OBrowser control is used to populate the destination field.

To allow drag and drop of content in the OBrowser control, you need to set its $dragmode property to kDragContent and set $allowjsdraganddrop to kFalse (which disables JavaScript drag and drop and allows drag and drop in the window class). This will enable an OBrowser control to generate drag events, that is, evDrag, evCanDrop, evWillDrop, evDrop, and evDragFinished which can be handled in your event methods.

On evCanDrop, the drag field parameter will contain the source field and on evWillDrop the drop field will contain the destination field. When dropping onto a field the drag field parameter of evDrop will contain the source control. No drag data is provided, however the current URL for the source drag field will be available via the $urlorcontrolname property.

Debugging code in OBrowser

You can debug code running in the OBrowser component in the Omnis Trace log, rather than in the Debug console in your browser. Console log messages sent from OBrowser go to the Omnis trace log *by default*. You can prevent this, and use the normal JavaScript console in your browser, by setting the configuration item "useOmnisTraceLogForConsole" in the "obrowser" section of the Omnis configuration file config.json to false.

You can attach the Chrome dev tools to your oBrowser instance, or JS Remote Form design window. To do this, open Chrome and visit chrome://inspect/#devices, make sure "Discover network targets" is enabled, click the "Configure" button and add "localhost:5989" (replacing 5989 with the oBrowser remote debug port if you have changed "remoteDebuggingPort" in config.json). The oBrowser instances will be listed which you can inspect.

Debugging on Windows

To debug code running in the OBrowser component under Windows, you have to set various security settings to allow debug mode. You can set the security settings in the Omnis config.json file, and start Omnis normally (and if you pass the parameters on the Omnis command line, they are ignored).

You can add the following entry to the "obrowser" section of config.json:

```
"obrowser": {
   "cefSwitches": [
      "allow-file-access-from-files",
      "disable-web-security"
   ]
}
```

The entry is an array of actual switch values to be passed to the Chromium Embedded Framework.

Chromium Safe Storage Prompt (macOS only)

On macOS, when first running an updated version of Omnis Studio, where a previous version was installed, the user will be presented with a prompt when oBrowser is loaded: "Omnis wants to use your confidential information stored in"Chromium Safe Storage" in your keychain – To allow this, enter the "login" keychain password."

The Chromium Safe Storage keychain item is used to grant access to the shared Chromium encrypted data store for secure storage of cookies. It is recommended that access is granted to allow cookies to be encrypted.

The prompt can be disabled by adding the use-mock-keychain CEF switch to the Omnis configuration. To do this, add the following entry to the "obrowser" section of config.json:

```
"obrowser": {
   "cefSwitches": [
      "use-mock-keychain"
   ]
}
```

It is important to note that if this is disabled cookies will NOT be secure.

Setting Locale

The "locale" attribute in the "obrowser" section in config.json allows you to set the client's locale. For example:

```
"obrowser": {
  "locale": "it_IT"
}
```

The locale attribute allows you to specify a language to be used in OBrowser (e.g. in context menu entries), and it can also be used to change the Accept-Language header, which servers can take into account to serve a language-specific page. For example, if the locale is set in config.json to it_IT and the $urlcontrolname property is http://www.google.com, Omnis will load Google's search engine page in Italian. When locale is not specified in config.json, a locale of en-GB is used by default. Note that the Accept-Language is only "for information" for the server, therefore, the server can send back a web page in English even if you requested another language.

In addition, the $acceptlanguagelist property can be set to a comma-delimited ordered list of language codes (ISO-639, without any white space), that is used in the Accept-Language HTTP header. For example, it-IT will try loading a web page in Italian, however the server determines whether the web page it sends back is in the requested language in Accept-Language or not. If $acceptlanguagelist is not specified (the default), then the value of the locale attribute in "obrowser" in the config.json is used. If specified, it will override the value in the attribute.

It is important to note that the $acceptlanguagelist will take effect only upon OBrowser component initialization. This means that you will need to set $acceptlanguagelist, close the window containing OBrowser in order to destroy the OBrowser instance and re-open the window so OBrowser re-initializes and uses the recently set $acceptlanguagelist values.

Certificate Errors

The **$ignorecertificateeerrors** property handles errors when OBrowser encounters errors with the website certificate. If true, a URL with certificate errors will be allowed to be loaded. If false (the default), an error will be raised and sent via the evBrowserFrameLoadError event. When an error is raised, the message in pErrorText will be "ERR_CERT: The certificate for this server is invalid: <errorcode>". This message can be tested to provide a prompt to the user to allow them to proceed to the insecure page after setting this property to true.

Browser Console Errors

If the **$donotredirectconsoletotracelog** property is true (the default), browser console messages generated by OBrowser are not redirected to the Omnis trace log. Set this to false if you wish to see console messages in the Omnis trace log (shown in the Studio Browser or via the Tools menu).

$disablepluginsmacos property (macOS)

The pre-Studio 10.x version of OBrowser on macOS had the property $disablepluginsmacos, but this is now obsolete and will not show in the Property Manager. The notation for this property will not cause an error in your code but it has no effect.

localStorage on macOS

OBrowser on macOS overrides the localStorage for file URLs (only). It writes the localStorage keys & values to a file called localStorage.json in the clientserver/client/ folder.

Cookies

The Chromium Embedded Framework used by OBrowser stores cookies in a SQLite database called Cookies, which is located in the user App Data folder, such as on Windows:

```
C:\Users\<username>\AppData\Local\Omnis Software\OS<version>\chromiumembedded\cache
```

Or in the /Application Support folder at /chromiumembedded/cache on macOS. This database can be managed using a SQLite DAM session.

Video in OBrowser

The type of videos that you can play embedded into a web page and displayed in OBrowser may be limited by the codec used to encode the videos. The CEF does not support all codec types, such as h264 since this requires royalty payments, and so is disabled in CEF. Supported formats are shown here: https://www.chromium.org/audio-video

HTTP headers

The **$headerlist** property is a runtime-only property allowing you to set the HTTP headers tor the embedded browser in OBrowser. It takes a two-column list of HTTP headers to be added, or removed, for each URL request. Column 1 is the header name (without trailing colon). Column 2 is the header value, which you can set to #NULL to remove the header.

The list is applied in line order, so you can modify an existing header by removing it and then adding it. The list is applied to every request made after assigning $urlorcontrolname to a URL. The *referer header* cannot be changed using this method.

OmnisIcn Control

Group	Icon	Name (type)	Description
Media	ICON	OmnisIcn Control (Xcomp)	Displays an icon from an Omnis file

The **OmnisIcn** control lets you display any icon stored in the OmnisPIC or USERPIC icon data files, or the #ICONS system table. Icons can have a transparent background. This control is included for backwards compatibility only, since Omnis image data files and #ICONS can only contain standard resolution icon images, and should therefore not be used for new applications.

Paged Pane

Group	Icon	Name (type)	Description
Containers	⧉	Paged Pane	Multiple pages or panes contain and other controls

The **Paged Pane** provide a number of pages or panes which can contain fields and other controls; they are similar to tab panes except that the panes do not have tabs. However, you can switch the current pane or page using a Tab Strip, a set of radio buttons, a pushbutton, or some other field that is linked to the Paged pane. In addition, a Paged pane can act as a Side Panel: see Side Panels.

In design and runtime mode you can set the number of panes in **$pagecount,** and change the current page using **$currentpage**; you can select the controls on different panes using the Field List. The **$movepage** property allows you to move a page pane in design mode, which is useful when adding new panes and you need to reorder existing panes.

To set up a paged pane, set $pagecount to the number of pages you need. You can add the fields and background objects to each page, changing the current page by setting $currentpage. At runtime, you can change panes in a method that sets $currentpage as required. Using a tab strip, you could set the page in a paged pane field to the selected tab, using the following event method on the tab strip:

```
# $event() method for the tab strip field
On evTabSelected
  Do $cwind.$objs.PagePaneId.$currentpage.$assign([pTabNumber])
```

Alternatively, you can design your own Next and Back buttons that cycle through the pages. For example:

```
# $construct() method for the window
# declare variable cCurPage initial value 1
# declare variable PageRef of type Item reference
Set reference PageRef to $cwind.$objs.PagePaneId

# $event() method for Next button
On evClick
  Calculate cCurPage as PageRef.$currentpage + 1
  If cCurPage > PageRef.$pagecount
    Calculate cCurPage as 1 ## if last pane, go to first
  End If
  Do PageRef.$currentpage.$assign(cCurPage)
  Quit event handler (Discard event)
```

Paged Pane Buttons

When set to kTrue, the **$showpagebuttons** property adds a page indicator to a paged pane (at the bottom in the center) to indicate which page is currently shown. The page indicator contains the same number of dots as there are panes in the control. The end user can change the current pane by clicking on the page counter.

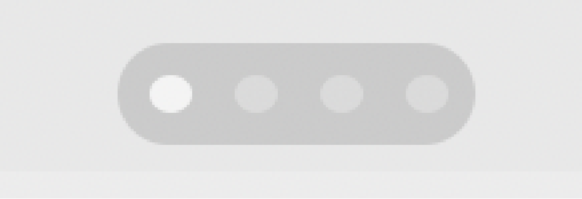


Figure 148:

When set to kTrue, the $animatui property causes the pages slide to the left or right when the page changes; when set to kFalse, the page pane will change instantly when the page button is clicked.

Listing the objects on panes

The $listobjects([iPaneNumber]) method returns a list of objects contained within the specified pane, including all foreground and background window objects. If iPaneNumber is omitted, the list contains information about the objects on *all panes* in the Paged pane field. The list has three columns: *object name* (empty for background objects), *ident* of the object, and *pane number*.

If you mark a field or object as "all panes", it will be included in the list regardless of the pane number specified.

Page names

The $alltabcaptions property contains the values of the $pagename property of the pages.

Side Panels

A **Side panel** is a common UI element in many dashboard style designs. A Side panel is a vertical panel down the side of a window, containing clickable options, such as a menu of options or other content, that can pop out on the left or right side of a window. A side panel can be shown automatically, when the user hovers the pointer over the left or right edge of the window, or linked to a button or menu option to allow the side panel to be opened or closed manually by the end user. When a side panel is opened it is animated.

There is not a separate Side Panel component, rather a side panel is a property of a window component ($sidepanel, see below), so for example, you can create a Side panel using a Page pane. To create or enable a side panel, you need to set the **$edgefloat** property of a control to either kEFposnLeftToolbar or kEFposnRightToolbar and the **$sidepanel** property will become enabled. Any window object *that can be marked as a left or right toolbar* will have the $sidepanel property and therefore can be enabled as a side panel. However, from a practical point of view, it would normally make sense to use a container type field, such as a Page pane, Scroll box, or Group box as a side panel, since you can then add other controls to the container which the end user can interact with.

The following example window contains a Scroll box on the left, enabled as a side panel, which contains a vertical tab strip containing a number of clickable options; in this case, the small "hamburger" button can be used to hide or show the side panel.

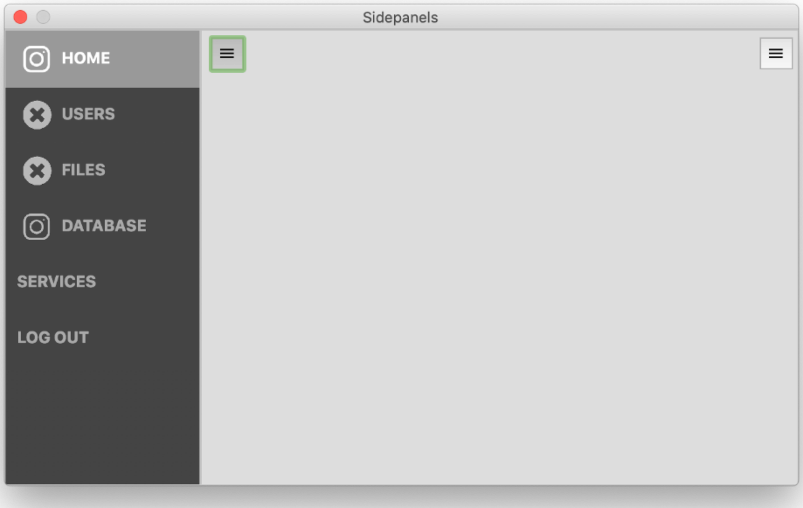


Figure 149:

You can also place one container type control inside another container and enable the second control as a side panel; for example, you could place a scroll box inside a page pane and make the scroll box a side panel. Using containers and other controls in this way, you can create some highly interactive interfaces or layouts in your application, such as the following:

Properties

When a window component is marked as a left or right toolbar (via $edgefloat), the **$sidepanel** property is enabled, and once enabled, you can set the property to kTrue to enable the side panel behavior. You can then set the **$sidepanelmode** property (the "peek" mode) so the side panel is shown and hidden automatically when the end user hovers their pointer over an area to the left or right of the window. The threshold for the peek area is 20 pixels on the left and right panels.

The $sidepanelmode property can be set to either a "push" or "cover" mode, as follows:

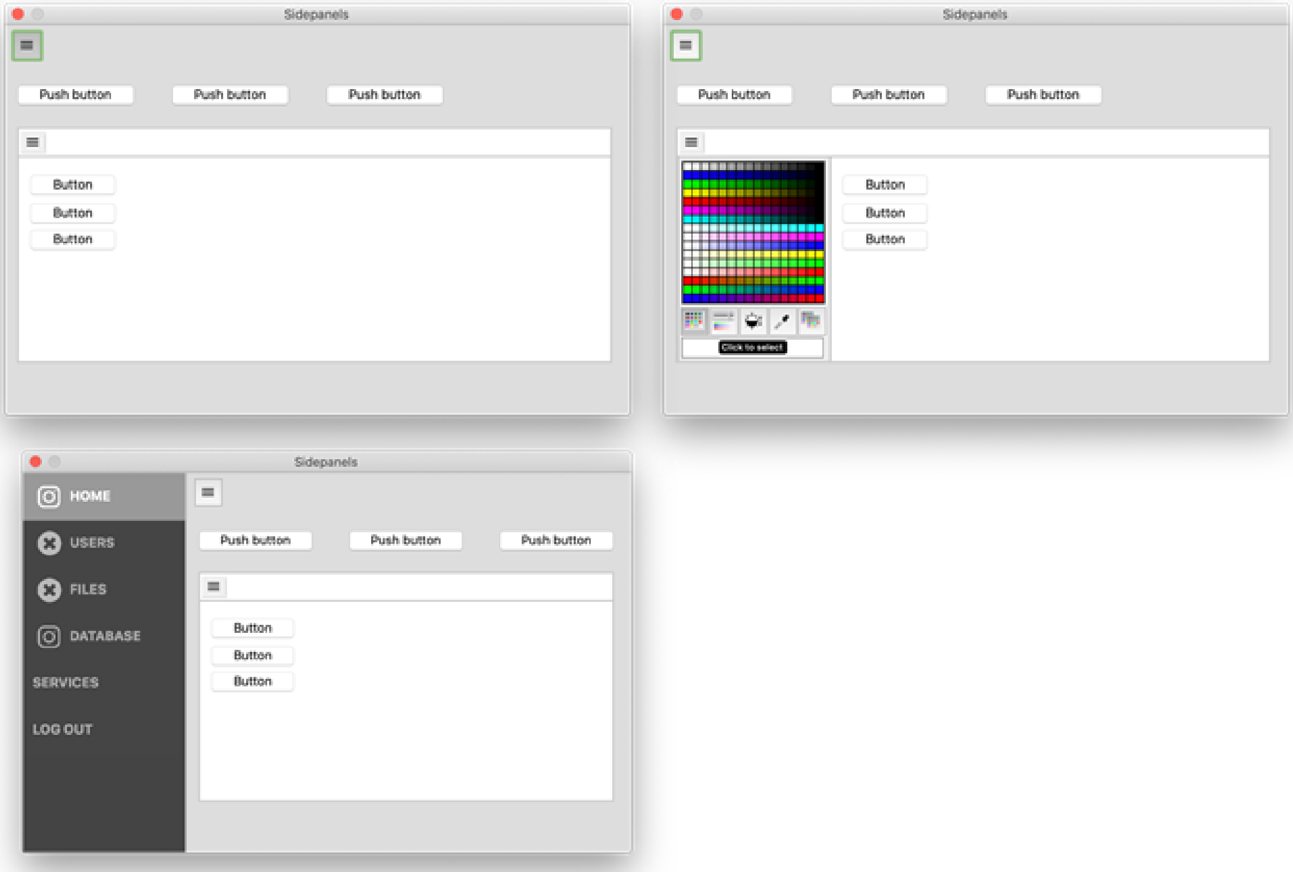


Figure 150:

- **kSidePanelModePush**
  this mode pops out the panel automatically and *pushes* or moves the other controls and content on the window either to the right or left.

- **kSidePanelModeCover**
  this mode pops out the panel automatically which is placed *over the top* of the other controls and content on the window.

- **kSidePanelModeNone**
  the default mode meaning the side panel will not pop out automatically, when the end user hovers over the window edge, but the $showpanel method can be used to show the side panel

When a Side panel is set to "push" mode, the panel does not push any components in the window off the edge of the screen, rather Omnis will adjust the bounds of the window area, that is, the width, and any components that have their edge floats set will be resized accordingly to fit the available window area.

When set to a mode, moving the mouse to where the side panel is fixed, either the left or right side of a window or a container, the panel will automatically animate or pop out. Moving the mouse out of the panel will autohide the panel.

When a side panel is closed (not visible), the panel is internally disabled for tabs. This prevents Omnis from tabbing to any controls within a hidden panel. When disabled, no events are sent.

Side panels support $dragborder meaning if the panel is opened and the border of the control is dragged, closing and re-opening the panel will open to the new dragged size.

Methods

Having enabled a control to behave as as side panel, by setting $sidepanel to kTrue, you can hide and show the panel manually, in your code, using the $showpanel method. In this case, you can set the $sidepanelmode property to kSidePanelModeNone and show or hide the panel using a button and this method.

- **$showpanel**(iAction, [iMode])
  performs an action (iAction) such as kSidePanelActionShow on a side panel object. The panel mode (iMode) is optional, is the display mode (as above) such as kSidePanelModeCover and only used when showing a panel

The side panel action constants are:

- **kSidePanelActionlHide**
  hides the side panel
- **kSidePanelActionShow**
  shows the side panel
- **kSidePanelActionToggle**
  either hides or shows the side panel depending on its current state

For example, the following code for a button shows a scroll box name ‘panel’ that is enabled as a side panel:

```
On evClick
  Do $cinst.$objs.panel.$showpanel(kSidePanelActionToggle,kSidePanelModePush)
```

Design Mode

In design mode, you can hide or show the side panel(s) using the window context menu, which would allow you to design the remainder of the window without the side panel getting in the way. When you Right-click on a window that has side panels enabled, the **Show Panels** submenu option allows you to hide or show the Left and/or Right side panels in design mode. If no panels are enabled for the window or container, the menu items are not present. In addition, clicking on a side panel in design mode will show a small semi-transparent button (shown below), which also allows you to hide or show the panel.

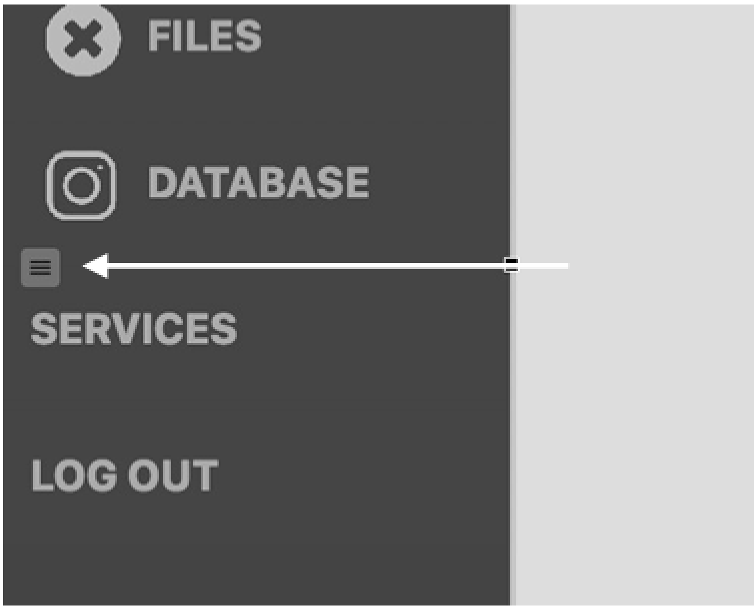


Figure 151:

Example

There is an example app to demonstrate Side Panels in the **Hub** in the Studio Browser, and on the Omnis GitHub repo at: https://github.com/OmnisStudio. Search for Omnis-SidePanels.

Picture Control

Group	Icon	Name (type)	Description
Media	🖼	Picture Control	Displays image data from a pictu variable

The **Picture** control lets you display image data retrieved from a server database (or an Omnis data file in legacy apps, which supports true color shared picture mode). The control requires an instance variable (or row variable column) of Picture type specified in the $dataname property. Pictures can have a horizontal and/or vertical scrollbar and scaling is controlled using the $noscale property. If set to kTrue, the $cachepicture property enables the client to store a copy of the image suitable for drawing without conversion, which provides faster drawing on the client, but more memory is used.

Events

In addition to the general field events **evBefore** and **evAfter,** pictures respond to **evClick** and **evDoubleClick** events which you can detect in the $event() method for the component.

Note that evClick and evDoubleClick are sent to the $event method in a Picture control *only in enter data mode*, while for some other controls (e.g. Lists, Edit fields) clicks are sent regardless of the enter data mode.

Note that evClick and evDoubleClick are sent to the $event method in a Picture control *regardless of enter data mode*. You can set $active to kFalse to prevent the events from triggering.

The **Pics2** tutorial library (in the welcome/tutorial/final folder) contains the **PicsWindow** class that uses the Picture control to display images from a SQLite database.

Icon ID and Color

The $iconid property allows you to use an icon from an iconset in the Picture control, for example, to create a background image for a window. The $iconcolor property can be used to specify a color for a themed SVG icon. Setting either $dataname or $calculation takes precedence over $iconid.

Image Interpolation

By default, Omnis interpolates (smooths) an image when rendering it on ultra-high definition displays. The $nointerpolation property allows you to disable interpolation (set it to kTrue), which may not be required for certain types of image, for example, for displaying a bar code (macOS only). $nointerpolation is kFalse by default which means all images will be interpolated.

Popup List

Group	Icon	Name (type)	Description
Lists	▤	Popup List	Single-column list allowing easy selection

The **Popup list** is most suited to short single-column lists from which the user can select a single choice. The user can click on the field to drop down the list. When you create a popup list you need to enter the name of your list variable in the $dataname property for the field. Enter the name of the variable in your list column in the $calculation property. Use the constant kDefaultBorder for the $effect property to ensure the list has the default border style for the current operating system.

You can open a Popup list with either the **Space** or **Return** key. In addition, on macOS, you can close a popup list with the Space key. When a popup list is closed, you can navigate through its values using the **Up** and **Down** arrow keys to select a value. In this case, an evClick event will be sent as you navigate up and down (the same as for Combo boxes).

Popup Menu

Group	Icon	Name (type)	Description
Menus	▤	Popup Menu	Displays a menu class on a wind

A **Popup menu** is a type of window field that opens a menu when you click on the field. You can create a popup menu using any previously defined menu class, and you can use any of the standard Omnis menus, such as **File** and **Edit,** as a popup menu. When you create a popup menu field you enter the name of the menu class in the field's $menuname property.

You can use the constant kDefaultBorder for the $effect property to ensure the menu has the default border style for the current operating system. All its other properties are the same as any normal window field. You set up the properties of the menu itself in the menu class.

Progress Bar

Group	Icon	Name (type)	Description
Other	▬	Progress Bar (Xcomp)	Indicates progress of a counter

The **Progress Bar** control lets you display a progress bar in a window. The value of the progress bar is specified in the $val property which is typically linked to the value of a counter in a looping command. You can specify the range for the progress bar in the $min and $max properties. You can specify the $backcolor of the bar as well as the $progresscolor. The current value of the progress bar can be displayed either a series of blocks (when $blocks is kTrue) or a continuous strip. Furthermore, the progress bar can be either vertical or horizontal by setting the $vertical property.

Note that the progress bar control has no events or built-in methods of its own. Rather you control it by assigning to the $val property in your code.

For example, a window could contain four progress bars named prog1 to prog4 and a simple looping method behind the pushbutton to activate the progress bars.

```
# $event() for button
On evClick
  For loop from 1 to 100 step 1
    Do method $setprog (loop)
  End For
  For loop from 100 to 1 step -1
    Do method $setprog (loop)
  End For
```

The $event() method for the button steps from 1 to 100 and back to 1 sending the current value in the loop parameter to the $setprog class method which in turn assigns the current value to the $val property for each progress bar.

```
# $setprog() method
# contains pValue parameter
Do $cinst.$objs.prog1.$val.$assign(pValue)
Do $cinst.$objs.prog2.$val.$assign(pValue)
Do $cinst.$objs.prog3.$val.$assign(pValue)
Do $cinst.$objs.prog4.$val.$assign(pValue)
```

Pushbuttons and Button Areas

Group	Icon	Name (type)	Description
Buttons	OK	Push Button	Button that responds to user cli

Pushbuttons are control fields that activate either user-defined methods (or standard Omnis database commands such as Find, Next, and Previous; for desktop apps using the Omnis datafile only). When you click on a user-defined button, Omnis triggers the **evClick** event and runs its **$event()** field method.

Button areas behave in exactly the same way as pushbuttons except that they are invisible on an open window (shown with a dotted or gray line in design mode). Button areas let you place an invisible and clickable control area on top of a graphic, or behind the whole window, for example.

You set the text for a pushbutton in the **$text** property. Under Windows, you can use the “&” character before a letter to specify a key to use with Alt key to push the button from the keyboard instead of with the pointer. For example, if you specify the text “&Cancel Tour”, you can use the Alt-C key combination to activate the button.

If you want to display more than one line of text on a pushbutton, use two forward slashes // to separate the lines. For example, setting the $text property to “Hello//World” results in a pushbutton displaying Hello and World on separate lines.

Pushbuttons have the following **Action** properties.

Property	Description
$buttonmode	mode or type of pushbutton or button area; buttons are user-defined by default which means you can add your own method
$actedata	if true the button is active during enter data; note the button will not work if this is set to false, in particular on modeless enter data windows
$actnomethod	if true the button is active when no methods are running; note the button will not work if this is set to false
$inactnorec	if true the pushbutton is inactive when there is no current record (applies to Omnis data files only)

Pushbuttons have the following **Appearance** properties.

Property	Description
$nogray	if true the button does not gray when inactive
$noflash	if true the button area does not flash when clicked (button areas only)
$buttonstyle	the drawing style of the button
$iconid	id of the icon used for picture buttons
$iconcolor	the icon color when a themed SVG icon is used

Styled Text

Styled text can be used in the $text property for push buttons (and radio buttons and check boxes) when the $styledtext property is kTrue.

Button Mode

When you create a pushbutton from the Component Store its **$buttonmode** is set to kBMuser by default, i.e. its mode is user-defined. This means you can enter your own $event() method behind the button which will run when the user clicks on the button.

Most of the button modes are *only appropriate to desktop apps using an Omnis data file* since they run standard Omnis database commands (such as Insert, Find, Next, and OK). You would have to create your own buttons to insert or next through data in a SQL database, or use one of the wizards to create a SQL window with ready-made buttons. The $buttonmode property is also available for Button Areas, and all modes are supported except for the color, linestyle, and pattern picker modes.

Button Style

The $buttonstyle property controls the drawing style or appearance of a push button, and can be set to one of the following: kSystemButton (the default style), kUserButton, kNoBorderButton, kHeadingButton, kComboButton, kRoundButton, kLargeRoundButton, or kIDEButton.

For headed lists and tree lists, $buttonstyle is the style of the header button, either kSystemButton or kUserButton.

IDE Button Style

The **kIDEButton** button style is used to style buttons in the Omnis IDE, but you can use the style in your apps if you want. The appearance for the kIDEButton buttonstyle is defined in the “IDEbutton” section in the Appearance configuration file (appearance.json). You can override the border and text colors set in the theme, by setting them to a color other than kColorDefault, but note that a disabled button always uses the disabled text color. Note that any changes you make to this will be reflected in the IDE.

Buttons styled with kIDEButton support hot tracking on both macOS and Windows, which can be controlled via the “hottracking” item in the “IDEbutton” section: 1 means buttons are hot on macOS, 2 means buttons are hot on Windows, 3 means buttons on both macOS and Windows are hot, and zero means hot tracking is not used on any platform.

Button Timer

You can add a timer to a pushbutton using the **$timeout** property to delay the evClick event. This is a runtime only property, and only assignable when the button has text.

You can assign a positive integer N to start a countdown timer that runs for N seconds, appending the time left to the button text; you can assign zero to stop the timer. When the timer expires, the button receives an evClick event with pTimeout set to kTrue to indicate the timer has finished. When $timeout is active, the text for the button updates once a second.

When sending evClick for the timeout, no evAfter is generated for the current field (this ensures the timeout event is received).

Click behavior on macOS

On macOS (for Studio 8 onwards) pushbuttons flash when they are clicked; this is the default behavior for macOS buttons. You can disable this behavior by setting the “macOSbuttonNewTextDrawingStyle” item in the “macOS” section of config.json to false.

Radio Groups and Buttons

Group	Icon	Name (type)	Description
Buttons	◉	Radio Button	Round button that can be either
Buttons	◉ Opt1 ◉ Opt2	Radio Button Group	Round buttons that can be eithe in mutually exclusive group

The **Radio Button Group** and **Radio Button** present a number of mutually exclusive buttons that can be either on or off: selecting one of the radio buttons deselects all other buttons in the group. The Radio Button Group field provides a ready-made group of buttons, whereas Radio Button fields have to be created individually and grouped together by numbering them consecutively (set their $order property to consecutive numbers). In this respect, Radio Button Groups are easier to create and move about the window.

There is an example app to demonstrate the Radio Button window control in the **Hub** in the Studio Browser, and on the Omnis GitHub repo at: https://github.com/OmnisStudio, search for Omnis-CheckRadio.

The field you associate with a Radio group or set of Radio buttons should be numeric (when using a number of radio button fields, they should all have the same $dataname). You enter the text to be displayed to the right of each radio button in the $text property for the object; for the radio group this is a comma separated list of text values.

Clicking a radio button sets the value of the field/variable in $dataname to zero for the first button, one for the second button, two for the third button, and so on.

The Radio Group field has the following properties:

- **$text**
  Comma separated list of values specifying the text for each button, e.g. Item 1,Item 2,Item 3,Item 4
- **$buttoncnt**
  The number of buttons in the group
- **$columncnt**
  The number of columns to split the group of buttons, e.g. using values $buttoncnt=4 and $columncnt=2 gives you a group of 2x2 buttons
- **$horizontal**
  if kTrue the Radio group will display the controls in a horizontal order; if kFalse the group is displayed vertically
- **$iconid**
  0 causes the control to display a default radio button according the current OS; alternatively this can be set to a multistate icon, including a size constant
- **$iconcolor**
  the icon color when the icon is a themed SVG icon; default is kColorDefault

An evClick event is sent to the $event method for the field when the control is selected or a value altered using the keyboard (using tab and space bar to select a button). The value of the variable specified in $dataname is updated when a button is selected.

Round Button

Group	Icon	Name (type)	Description
Buttons	👤	Round Button (Xcomp)	Round Button showing progres individual values

The **Round Button** control (RoundButt) provides a graphical & highly configurable button for your windows. You can use the Round Button to show the progress of a process, or to show individual values in a group of data points such as percentages. The Round button control uses transparency so requires a minimum of Windows 8 or higher.

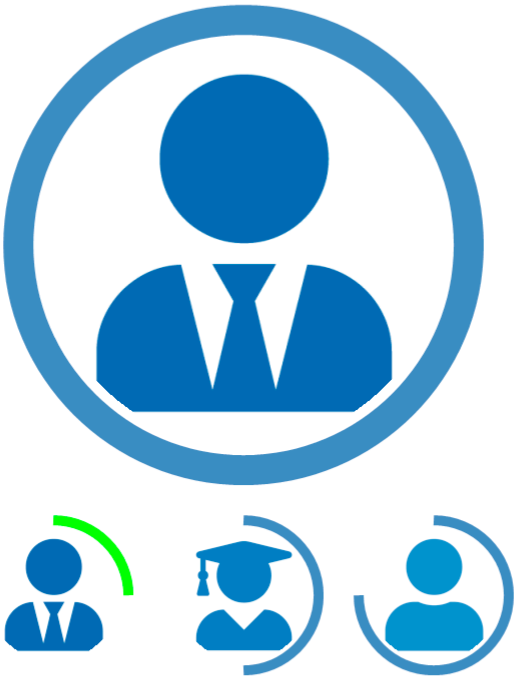


Figure 152:

The Round button has a number of properties which can be set at runtime to indicate progress.

- **$centerimage**
  an optional image which will be clipped inside the circular progress bar, including the amount specified in $progressgap.
- **$progressalpha**
  the alpha value of the progress bar: 0-255 with zero being transparent
- **$progresscolor**
  the RGB color of the progress bar
- **$progressgap**
  the gap between the inside of the progress bar and the center image
- **$progressstartangle**
  the starting angle of the progress bar: 0-359 with zero at the top
- **$progressvalue**
  the current value of the progress bar as a percentage: 0-100 with 100 being progress complete, that is, the progress bar is a complete full circle
- **$progresswidth**
  the width of the progress bar in pixels

As well as these properties to configure the appearance of the control, the control responds to a standard click which you add event processing to.

Screen Report Field

Group	Icon	Name (type)	Description
Reports	▤	Screen Report Field	Displays the output of a report

The **Screen Report Field** lets you display the output of a report on a window, rather than sending the report to the default Preview screen. You use the *Send to a window field* command to direct output to a screen report field. The user can copy data from a screen report field instance by dragging the pointer or mouse on the report to select some data.

The screen report field has all the properties of a standard window field in addition to the $showpaper under the Appearance tab in the Property Manager. If you set $showpaper to true, it changes the field to page preview mode.

The current page count is reported in the $pagecount property (read only), while $currentpage is the currently displayed page and is assignable at runtime. When more than one page is visible, the value indicates the page that is most visible.

This type of field does not have a $dataname or $classname, and it does not generate any events of its own apart from the events for a standard field.

You could put the following method behind a pushbutton on your window or a toolbar control to print to your screen report field.

```
On evClick
  Set report name ReportName
  Send to a window field {ScreenReportFieldName}
  Print report
```

The screen report field has two methods:

- $zoom(bZoomOn=kTrue)
  enables zoom mode when the screen report field is in page preview mode
- $redirect(bPrompt=kTrue)
  redirects the current report by prompting for a different print device, rather than the device specified in default preferences

Printing the report to PDF

You can use the $print() method and the bToPDF and cPDFPath parameters to print the report in a Screen Report field to a PDF file. When bToPDF is kTrue and a path is specified in cPDFPath, a PDF file is created in the specified location. For example:

```
Do ScreenReportField.$print([bToPDF=kTrue,cPDFPath='<path>'])
```

If cPDFPath is empty, a prompt is shown allowing the end user to specify a path for the PDF file. If the parameters are omitted or bToPDF is kFalse the method prints the report displayed in a screen report field to the current report destination, e.g. the Screen or Printer.

Scroll Box

Group	Icon	Name (type)	Description
Containers	◫	Scroll Box	Group other fields in a scrollable

The **Scroll Box** allows you to group fields in a scrollable area, and is similar to the Group box. See Group box.

Shape Field

Group	Icon	Name (type)	Description
Shapes	Abc	Shape Field	Shape with some field propertie other shapes see Background O

The **Shape field** is a graphic object (rectangle, line, or text) that has some general field properties, such as $visible, $active, and $enabled. Therefore, you can hide a shape field, make it inactive, or disable it just like an ordinary field. Shape fields can contain methods including a $event() method to detect events, so you can detect when the mouse enters or leaves the field.

You can specify the $effect, $bordercolor and $linestyle properties to apply border style effects to shape fields.

Sidebar Control

Group	Icon	Name (type)	Description
Navigation	◫	Sidebar Control	Displays a list of options with gro

The **Sidebar Control** displays a list of options with groups. It can be built on the fly or loaded from a database as the window is instantiated. The list must contain lines that define the group names and the items in the group, in the following format:

Line #	List columns
1	Group 1 name
2	Item 1 name, IconID, Value
3	Group 2 name
4	Item 1 name, IconID, Value
5	Item 2 name, IconID, Value
6	Group 3 name
7	Item 1 name, IconID, Value
8	Item 2 name, IconID, Value
9	Item 3 name, IconID, Value

You can create and build the list in the $construct() method of the window using a method similar to the following. The list iSidebarList is an instance variable in the window, along with the column variables iGroup, iIconID, iName, iID. The iID variable is used in other methods in the window to pass which item in the sidebar is chosen.

```
# $construct() of window
Do iSidebarList.$define(iGroup,iIconID,iName,iID)
# the remainder of method builds content of sidebar list
Do iSidebarList.$add("Yellow",0,"") ## Group 1
Do iSidebarList.$add("Yellow",k32x32+1,"Yellow",1)
Do iSidebarList.$add("Red",0,"") ## Group 2
Do iSidebarList.$add("Red",k32x32+2,"Red",2)
Do iSidebarList.$add("Green",0,"") ## Group3
Do iSidebarList.$add("Green",k32x32+3,"Green",3)
```

The sidebar component reports the evIconPicked event which passes the pLinenum parameter containing the line number in the list corresponding to the item selected. When considering the value of pLinenum you have to take account of all groups and items in the list. Using the list above pLinenum will be value 2 (ie line 2 in the list) for the first item in the group 1, value 4 for the item in group 2 and value 6 for the item in the third group.

You can detect the evIconPicked event in the $event() method for the sidebar and branch your method according to the item selected. The following method

```
# $event method for sidebar component
On evIconPicked
  Do iSidebarList.$line.$assign(pLinenum)
  # selects the list line according to item selected in sidebar
  Do iSidebarList.$loadcols()
```

```
# loads the values in the selected list line, including iID
Switch iID ## branches according to value of iID
  Case 1
    Do something..
  Case 2
    Do something..
  Case 3
    Do something..
End Switch
```

Single Line Entry Field

Group	Icon	Name (type)	Description
Entry Fields	Ab	Single Line Entry Field	Field into which users can insert view existing data

A **Single Line Entry Field** (or Edit field) is a type of window field into which users can insert data or view existing data. For window classes, there are several types of entry fields available, which are separate controls: the *Single Line Entry Field*, the *Multi Line Entry Field*, and Masked Entry field. You can modify a window entry field to create a *display* or *local field* that you can use to display data.

Entry Field Properties

Entry fields have the following properties, in addition to the standard properties, such as $active, $enabled, and $dataname (see below):

Property	Description
$allowcopy	If true, when the field is active and disabled,the user can set focus to the field, select text with mouse or select all, and copy to clipboard. Note that the field does not generate click events when it is active, disabled and $allowcopy is kTrue
$animateui	If the library preference $animateui is true, all objects that support $animateui will animate aspects of their interface. The object property only applies when the library preference is false.
$autocorrectspelling	If true, and the user types a separator (e.g. space or comma) when no text is selected, the control replaces a misspelt word immediately before the selection with a correctly spelt word
$autofind	If true, the field is an automatic find field
$autotablen	The number of characters entered before automatically tabbing out of the field

Property	Description
$bordericonstyle	The style applied to plain border styles. You can set border integrated icons, icon colors and tints
$borderradius	Radius for rounded border corners. 1 to 4 pixel values separated by -, in order topleft,topright,bottomright,bottomleft. If bottomleft is omitted it is topright. If bottomright is omitted it is topleft. If topright is omitted it is topleft
$calculated	If true, the field is calculated
$contentpadding	Padding inside a border. 1 to 4 pixel values separated by -, in order left,top,right,bottom. If bottom is omitted it is top. If right is omitted it is left. If top is omitted it is left
$contenttip	Text which is displayed in the field when it is empty, to help the user understand what content should be entered
$disabledefaultcontextmenu	If true, the default context menu for the object will not be generated in response to a context click. $disabledefaultcontextmenu for $clib and $cobj must both be false for the menu to be generated
$disablediacriticalpopup	If true, long holding a character will not show a popup if diacritical character alternatives are configured for the character
$fadewhendisabled	If true, the field fades its content when $enabled is kFalse
$fieldstylefocused	The style in system class #STYLES assigned to this field in addition to $fieldstyle when the control has focus
$gridsection	The type of the complex grid section containing the object. One of the kGrid... constants
$negallowed	Only applies when the dataname is a numeric type. If true, the entry field allows negative values
$passwordchar	If set, the object draws this instead of each value in the data, allowing private entry of passwords; if set, the data length cannot exceed 255 characters. Using * (asterisk) on macOS or Windows with themes enabled, draws a solid circle

Property	Description
$righttoleft	If true, the edit field edits text with a right-to-left reading direction; the edit field is then suitable for languages such as Arabic
$showellipsis	If true, show an ellipsis if the text is too long (only applies when the control is read-only, $horzscroll and $righttoleft are both kFalse, $align is kLeftJst and $passwordchar is not set)
$shownulls	If true, the field displays null values using the text 'NULL'
$showspellingerrors	If true, the control underlines spelling errors using a dotted line
$stripspaces	If true, the control strips leading and trailing spaces from the data before storing it in the dataname
$stripthousandonpaste	If true and a number field, thousand characters will be stripped from clipboard content on paste
$text	The text or calculation stored with the object
$tooltippos	A kTooltipPos… constant that specifies where $tooltip appears relative to the control: kTooltipPosBottom, kTooltipPosLeft, kTooltipPosMouse, kTooltipPosRight, or kTooltipPosTop
$unqindex	If true, the field corresponds to a unique index
$uppercase	If true, the entry field is upper case only
$vertcentertext	If true, single line text (or any text in a kText background object) is vertically centered in the height of the field. If false, the text is vertically positioned according to the rules of Omnis Studio 5
$zeroempty	If true, the field shows a value of zero as an empty string

$dataname for Windows Controls

The variable specified in the $dataname property of a window control *must be an instance variable,* or in some cases a column in a row instance variable in the form VarName.ColumnName. You can click into the $dataname property in the Property Manager and type the first letter or few letters of a variable and then select its name from the list that drops down. For example, to select an instance variable in the window class, type "i" to display a list of all instance variables and select one from the list.

Alternatively, you can type the name of a new variable, press Return and define the new variable in the 'Create Variable' dialog, adding its Scope, Type (and Subtype if applicable), Initial value, and Description. See Variables for more information about creating and using variables.

Password Entry Fields

Single line entry fields have a property $passwordchar which specifies the character to be displayed for every character entered in the field. When the property is set, the data in the field cannot exceed 255 characters, and while the focus is on the field the Cut and Copy items on the Edit menu are disabled.

Local Fields

A *local field* is a field that depends on the value of the prior field in the tab order. Omnis redraws these fields immediately after redrawing or changing the prior field. You usually use local fields to display data changed as a result of an entry in a preceding field. Omnis will not automatically execute the field procedure on recalculation. You can have more than one local field running in sequence after a non-local field.

Using a local field after a list box, you can set up a spreadsheet-like edit bar for a selected list line. When you select a line in a list, the local field changes to display that line; you can then edit the line and put the updated line back into the list.

A calculated display field following a field you specify as part of the text or calculation should have the **local** property to ensure up-to-date display of the display field value.

Entry and Display Field Calculations

You can specify a validation expression for the data in the field. You can use input masks to force the user to input data in certain basic formats, but more complex logical constraints require an expression. To make a calculated field you must set its $calculated property to true. You can enter an Omnis expression into its $text property. When the user leaves the field, Omnis validates the data using the expression. If the expression evaluates to false, Omnis beeps and returns the cursor to the field. For display fields, the $text property lets you enter a character or numeric value or Omnis expression that is displayed in the field.

Display and Inactive Fields

A *display field* is a type of window field that you use to display data only, that is, the user cannot enter data into a display field. To change a standard entry field into a display field you change its $enabled property to false; to display data in the field you set its $calculated property to true and enter the data in its $text property. The user can't tab to a disabled display field or click in it and enter data, but a display field still accepts mouse events, such as mouse leave events. To make a field completely inactive you need to change its $active property to false, regardless of its $enabled setting. Such an inactive field does not receive mouse events and you cannot enter data into it.

Disabled Field Appearance

The $fadewhendisabled property controls whether the contents of a disabled window field are greyed out or not: this affects the Entry, Masked Entry Field, and Multiline Entry Field window field types (kEntry, kMaskedEntry, kMultilineEntry). When kTrue (the default is kFalse), and if the field's $enabled property is kFalse, the field content will be partially greyed out.

There theme colors "fadewhendisabledcolor" (defaults to kColorWindow) and "fadewhendisabledalpha" in appearance.json (and the theme templates) allow you to control the color and transparency of $fadewhendisabled: "fadewhendisabledalpha" is the amount of alpha used when fading to kColorWindow (default is 140).

Floating Edges and Positioning

The $edgefloat or "floating edge" property for window components, including entry fields, allow the components to be resized automatically when the end user resizes the window at runtime (note a window also has $edgefloat). The $edgefloat property can be set to one of the **kEF...** constants which determines which edges of the component, if any, will "float" or reposition automatically when the window is resized. The following kEF... constants are available for entry fields:

Constant	Description
kEFnone	No floating edges
kEFall	All edges floating
kEFbottom	Bottom edge floating
kEFbottomAndCenterLeftRight	A combination of kEFbottom and kEFcenterLeftRight
kEFcenterAll	All edges floating, keeping the control centered in its parent

Constant	Description
kEFcenterLeftRight	Left and right edges floating,keeping the control centered horizontally in its parent
kEFcenterTopBottom	Top and bottom edges floating, keeping the control centered vertically in its parent
kEFleftRight	Left and right edges floating
kEFleftRightAndCenterTopBottom	A combination of kEFleftRight and kEFcenterTopBottom
kEFleftRightBottom	Left,right and bottom edges floating
kEFright	Right edge floating
kEFrightAndCenterTopBottom	A combination of kEFright and kEFcenterTopBottom
kEFrightBottom	Right and bottom edges floating
kEFrightTopBottom	Right,top and bottom edges floating
kEFtopBottom	Top and bottom edges floating
kEFtopBottomAndCenterLeftRight	A combination of kEFtopBottom and kEFcenterLeftRight

All $edgefloat constants prefixed with **kEFposn…** will reposition the control in the specified region of the window; as you select one of these constants in design mode the control will snap to the chosen region, and when the window is resized at runtime the control will “stick” to this region. The **kEFposnClient** constant stretches the control to fit the available area within its parent or subform. The following kEFposn… constants are available for entry fields:

Constant	Description
kEFposnBottomToolBar	Place field in bottom toolbar position
kEFposnClient	Place field in the client position
kEFposnFooter	Field is the horizontal footer
kEFposnHorzHeader	Field is the horizontal header
kEFposnJoinFooter	Field is located where the horizontal footer and vertical header meet
kEFposnJoinHeaders	Field is located where the horizontal and vertical headers meet
kEFposnLeftToolBar	Place field in left toolbar position
kEFposnMainHeader	Field is the main header
kEFposnMenuBar	Place field in menu bar position
kEFposnRightToolBar	Place field in right toolbar position
kEFposnStatusBar	Place field in status bar position
kEFposnTopToolBar	Place field in top toolbar position
kEFposnVertHeader	Field is the vertical header

Copying Text from Disabled Fields

When set to kTrue, the $allowcopy property allows the end user to copy data from a disabled field ($enabled is kFalse). When the field is enabled, the setting of $allowcopy is ignored.

Control Characters

You can specify that control characters are visible in Edit fields (at runtime only): this also applies to Multi-line edit fields, as well as the editable field part of Combo boxes, Data grids, and String grid controls. A library preference $showcontrolcharacters enables this for the whole library, or you can set the property for individual controls of those types: you can set the property in the Property Manager or in your code.

```
Do \$cobj.\$showcontrolcharacters.\$assign(kTrue)
Do \$clib.\$prefs.\$showcontrolcharacters.\$assign(kTrue)
```

When set to true, control characters are drawn using a suitable symbol, rather than space which is the default (when the property is false). Control characters are characters with a value less than Space (with the exception of carriage return for window controls which use CR as a line delimiter) and Del (0x7f).

Strip Control Characters from Edit Fields

If the $pastestripscontrolcharacters property is true, then all unused control characters are removed when pasting character data into the edit field, or the editable part of combo boxes, data grids, or string grids. This property applies to Edit Fields (Single-line entry), Multi-line Edit fields, Masked Entry fields, and Token Edit fields (as well as Combo boxes, Data grids, and String grids). Plus there is a library preference $pastestripscontrolcharacters to strip control characters for all controls.

The Control characters that are stripped include 0-0x1f and 0x7f. All control characters are “unused” except for the carriage return line delimiters used by certain fields.

The library preference $clib.$prefs.$pastestripscontrolcharacters defaults to kTrue in a new library, and kFalse in existing (converted) libraries.

Strip Spaces

When set to kTrue, the $stripspaces property ensures that all leading and trailing spaces are stripped from the data before storing it in the variable or field. This property is set to kTrue to maintain compatibility with previous versions, which means leading and trailing spaces are stripped from data. If however you want to retain the exact data that is entered by the user, including any leading and trailing spaces, you need to set this property to kFalse.

Strip Thousand Separators

If the edit field is a number type field and the $stripthousandonpaste property is kTrue, when the end user pastes data from the clipboard the data is checked for thousand separators and if present they are stripped allowing the content to be pasted into the field.

Content Tips

Entry fields have the **$contenttip** property which is a text string which is displayed in the entry field when it is empty and before the end user has entered any text. Using content tips may help the user understand what content should be entered into fields in the forms in your application. For example, for a Last name field you could enter ‘Enter your last name’ into $contenttip to prompt the end user for their last name. As soon as the cursor enters the field the content tip will disappear.

The text in a content tip can be styled. When you enter the content tip in the Property Manager, a text editor allows you to select various styles including bold, italic, underline, and colors for the text. You can use the style() function to format the text when assigning a value to $contenttip, such as con(style(kEscColor,kRed),‘Enter your last name’).

There is an example app called **Field Border Icons and Content Tips** in the **Samples** section of the **Hub** in the Studio Browser.

Animated Content Tips

Content tips for entry fields can be animated, meaning that when enabled and the focus enters the field, the content tip inside the field will ‘float’ or move up above the field. You can, for example, use an animated content tip instead of a separate label control, which also creates a more interactive UI.

To allow animated content tips, the **$animateui** property supported for Entry fields, Multi-line Edit fields, Token Entry fields, and Combo boxes (the entry field part). When set to kTrue for these field types, and when the focus enters the field, the content tip will float above the field. When animated, the content tip will shrink to 80% of the edit field font size, and use the same font colors as the edit field.

This feature is not supported for Entry fields or Combo boxes that are inside a Complex grid.

Content Padding

The $contentpadding property allows you to add padding around content inside an Entry Field. It is specified in pixels, with 1 to 4 pixel values separated by -, in the order left, top, right, bottom; if a single value is specified it is applied to all four sides. If the bottom value is omitted, the top value is used. If the right value is omitted, the left value is used. If the top value is omitted, the left value is used. For example, a value of '3-2-1' gives a 3 pixel gap on the left, a 2 pixel gap on the top and bottom, and a 1 pixel gap on the right.

Show Ellipsis

If $showellipsis is true, an ellipsis is shown in the field if the text is too long; the property only applies when the control is read-only (i.e. the data is not being edited), $horzscroll and $righttoleft are both kFalse, $align is kLeftJst and $passwordchar is not set. Note that the edit field always includes at least the first character of the text, so very narrow edit fields will sometimes show truncated text, but in most cases this will not be apparent.

Field Border Icons

You can add icons to the left and right border of Entry fields to provide a visual hint or feedback, adding to the ease of use for the end user. For example, you could show a check mark icon to indicate when a field has been correctly filled out. There is an example app called **Field Border Icons and Content Tips** in the **Samples** section of the **Hub** in the Studio Browser.

Field border icons can be added to all window class entry field types, including *Single Line Entry fields, Multi Line Entry fields, Masked Entry fields,* and *Token Entry fields*; the new property only applies when the field border style in $effect is set to kBorderCtrlEdit (that is, when $fieldstyle is the default CtrlEditText), kBorderCtrlList, or kBorderPlain. Note that the icons are for display only, and do not report any events, so for example, you cannot add code to them to react to clicks.

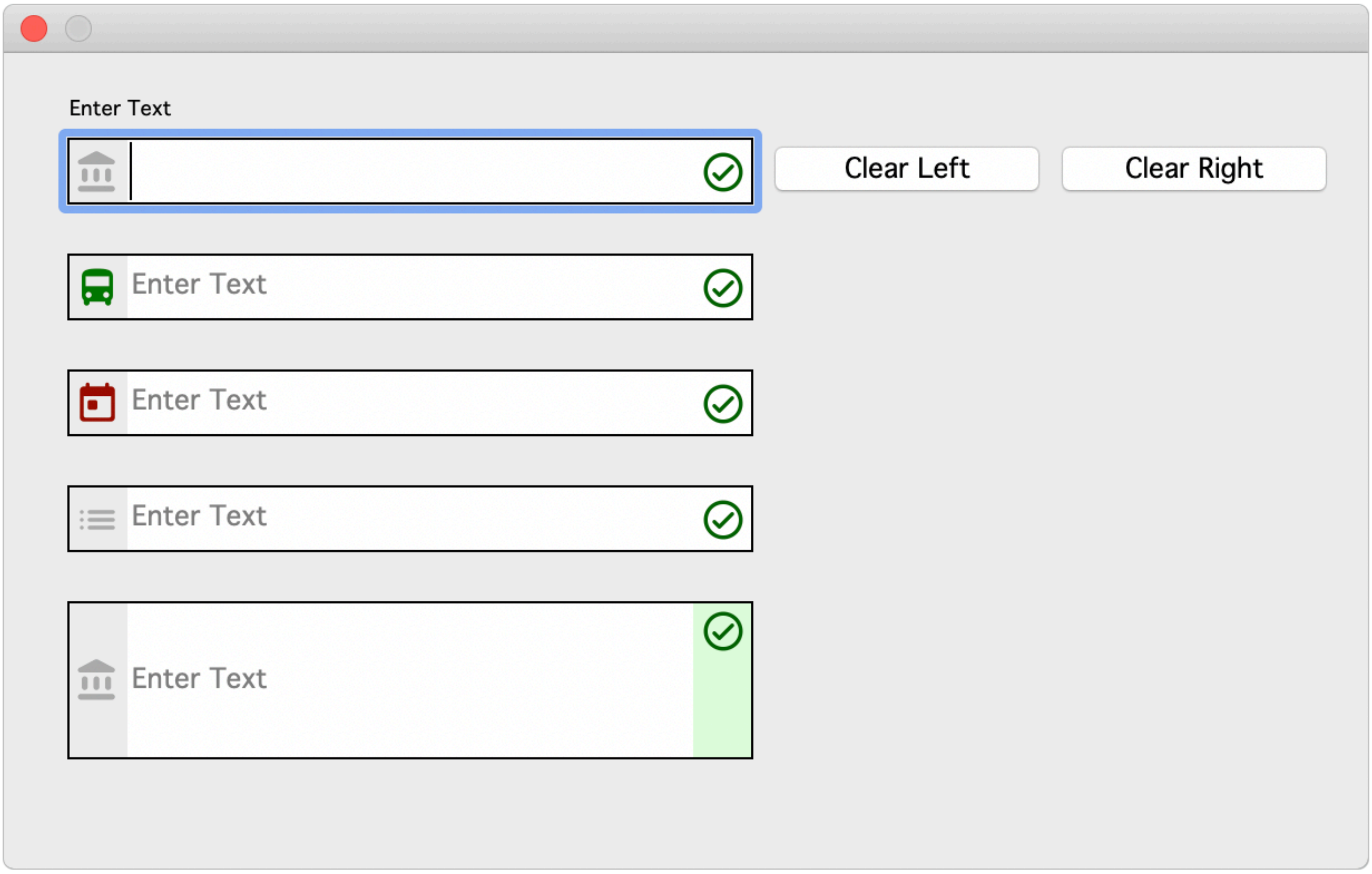


Figure 153:

You can set the color of the icon (if using an SVG), the background color of the icon area, and the vertical alignment. By default the background color is the same as the control. The content area of the edit field is adjusted inside the frame to accommodate icons if set. If the control is below 20 pixels in height the icon will scale down.

The **$bordericonstyle** property stores the left and right icon configurations for an Entry field, including the iconid of the left and right icon (which can be a SVG or PNG specified by character or integer iconid), plus the icon color and background color (e.g. kColorDefault or a RGB value). The single property stores the settings for the left and right icons, which you can specify in the Property Manager:

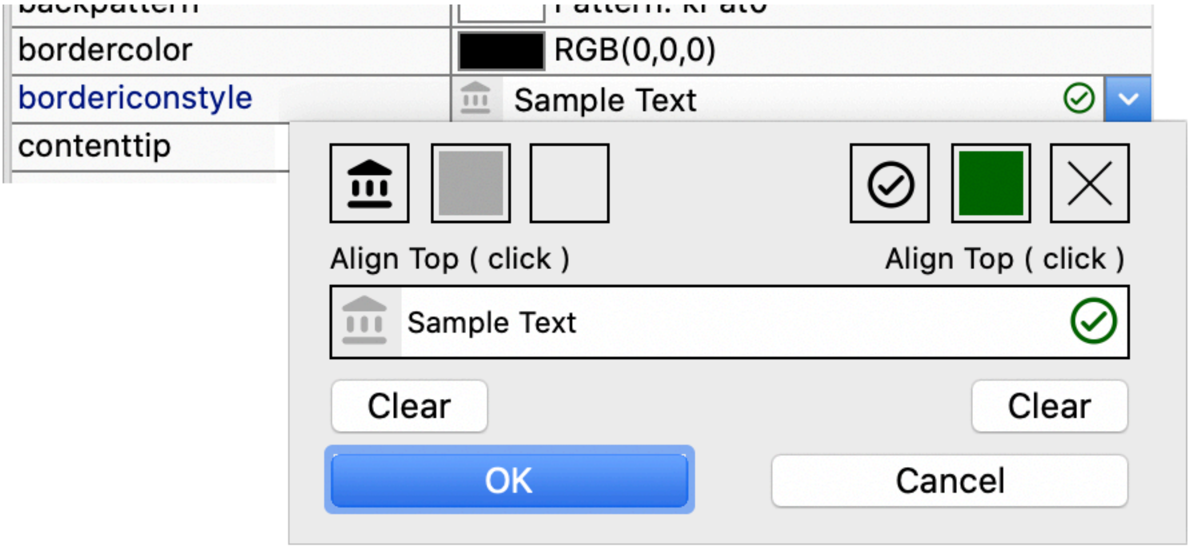


Figure 154:

The **$setbordericonstyle** method allows you to set the left or right icon for an entry field, or clear the icon(s); the method has the following syntax:

- **$setbordericonstyle**(bLeftIcon[,cIcnIDName,iIcnTintColor,iBackTintColor])
  bLeftIcon should be kTrue to enable a Left icon, or kFalse for a Right icon
  cIcnIDName is the name ID of the icon (can be a string for a SVG icon)
  iIcnTintColor, iBackTintColor are the colors for the icon and background

The following code for a field event method shows a warning icon on the right if no content is added, otherwise if content is added a check mark icon is shown:

```
On evAfter
 If len($cobj.$contents)<=0
 Do $cobj.$setbordericonstyle(kFalse,"cancel",kDarkRed)
 Sound bell
 Quit event handler (Discard event)
Else
 Do $cobj.$setbordericonstyle(kFalse,"check_circle",kDarkGreen)
 End If
```

To clear an icon, you pass bLeftIcon as either kTrue (left icon) or kFalse (right icon) with no value for cIcnIDName, as follows:

```
Do $cinst.$objs.FIELD.$setbordericonstyle(kTrue) ## clears the left icon
```

Emoji and Symbols in Edit Fields (macOS)

On macOS, the standard **Emoji and Symbols** menu item from the Edit menu is supported for Entry fields. This will display the Character Viewer to allow entry of emoji, symbols, accented letters and characters from other languages. Note this does not support drag and drop from the character viewer into Omnis.

You can remove the Emoji and Symbols menu item by setting the “useCharacterPalette” item in the “macOS” section of config.json to false.

Dictation for Edit Fields (macOS)

End users can enter text into an edit field on macOS using the built-in Dictation feature, which tries to convert audible speech into meaningful text. To allow dictation to occur the focus must be in the edit field, which must itself be editable, i.e. not disabled, and dictation must be enabled on the client computer. Dictation is available wherever text input is required, that is, in Single- and Multi-line edit fields, the edit part of Combo boxes, and edit fields in Complex grids.

Disabling Dictation

Support for Dictation is turned on in Omnis by default, so to disable it you need to edit the Omnis configuration file. There is a "useDictation" option in the "macOS" group in config.json, which must be set to false to disable dictation; note you have to quit Omnis in order to apply the change to the config.json file. Dictation will be disabled when you restart Omnis.

```
"macOS": {
  "useDictation": false
}
```

Using Dictation in Edit fields

To enter dictation mode, place the cursor in the edit field and select the *Start Dictation* option from the Edit menu on macOS, or press the Function key twice (Fn + Fn). This will open the dictation popup (usually at the insertion point, or in the center of the screen) and put the computer in listening mode. Dictation can be stopped or cancelled by clicking on Done in the popup, or using the *Stop Dictation* menu option.

Dictation Level

There are two levels of dictation provided by macOS: *Standard* or *Voice Control (in macOS Catalina or later)*. These can be enabled from System Preferences->Keyboard->Dictation, or System Preferences->Accessibility.

Standard dictation (the default) requires an internet connection and provides speech to text translation using Apple's servers. On older systems, the text is not translated until the Done button is pressed on the popup. On newer systems text is translated and placed into the field while the end user is speaking. Dictation will end automatically when text is entered from the keyboard or the field loses the focus.

Voice Control has been introduced in macOS Catalina to improve on and replace the earlier Enhanced Dictation feature. Speech can be dictated in Studio via Voice Control when Dictation is enabled in Studio. Voice Control is enabled via the Accessibility System Preference. When an edit field in Studio is accepting input and Voice Control is active then speech input will be translated into text via the Voice Control speech engine.

Diacritical Characters

End users can enter various characters with diacritical marks by using a popup. The new **Diacritical Character** popup is available on end user windows (window classes) in any entry field that accepts text including Single Line Entry fields, Multi Line Entry fields and Combo boxes, as well as in most edit boxes in the Studio IDE, including the Method Editor. (Note *this feature is not available for JavaScript Edit controls in remote forms,* although if your app is running on a mobile device the soft keypad may provide a similar function for entering diacritical characters.)

To enter a diacritical character, the end user needs to hold down the character key, and if the character has additional diacritical options, a popup will be shown containing that character with a range of diacritical marks applied from which a selection can be made. For example, the end user can hold down on the 'a' character key and a small dialog will popup containing a number of diacritical variations for the 'a' character, as shown:

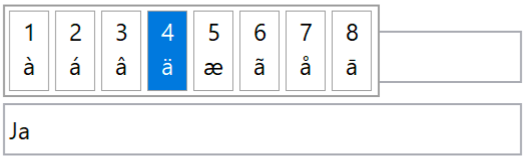


Figure 155:

When the popup is shown the end user can press the Left and Right arrow keys to move back and forth, and then press Return to select a character. Clicking the mouse or pointer on a character will also select a character. In addition, the top row in the popup contains a number index and pressing a numeric key will select the corresponding character. The Shift key can also be used along with the character key to enter an uppercase diacritical character.

Pressing Escape or clicking away from the popup will dismiss the popup.

When a character is selected, the popup is dismissed and the last typed character in the original field will be replaced with the selected character.

Popup Content

The popup content varies from language to language. To support different languages a new folder called “Keyboard” has been added to the “Local” folder in the Omnis tree. If you remove this folder no diacritical popup will be shown.

Five languages are supported: English, French, German, Italian, and Spanish. There is a file for each of these languages: en.json, fr.json, de.json, it.json and es.json. When the popup is required, Omnis will load the popup content based on the language of the client, for example, when using English on the client the en.json file is loaded from the Keyboard folder.

You can add more files to this folder to support more languages. The json files have the following structure:

```
"diacritical": {
  "A" : "A À Á Â Ä Æ Ã Å Ā",
  "C" : "C Ç Ć Č",
  "E" : "E È É Ê Ë Ē Ė Ę",
  "I" : "I Î Ï Í Ī Į Ì",
  "L" : "L Ł",
  "N" : "N Ñ Ń",
  "O" : "O Ô Ö Ò Ó Œ Ø Ō Õ",
  "S" : "S Ś Š",
  "U" : "U Û Ü Ù Ú Ū"
}
```

Omnis will search this file for the character key being held down and using the above table present a popup with the various options and index numbers.

You can disable this feature for individual fields in a window class by setting the $disablediacriticalpopup property to kTrue, which is on the Action tab.

macOS Keyboard Layout

On macOS, the user has an option to show a keyboard language menu allowing them to switch between different STANDARD language keyboard input layouts.

With the option **diacriticalpopupuseosxkeyboardlayout** in config.json set to true, depending on the selected keyboard layout in macOS, Omnis will ignore its own ‘current’ language setting and load a file from the keyboard folder. A ‘language to file mapping’ must also exists in config.json.

For example, if Omnis is in English but diacriticalpopupuseosxkeyboardlayout is true, and the user has selected French from the standard keyboard layout menu in macOS, Omnis will load the ‘fr.json’ file for the diacritical character popup content.

```
"diacriticalpopup": {
  "diacriticalpopupuseosxkeyboardlayout": true,
  "com.apple.keylayout.British": "en",
  "com.apple.keylayout.German": "de",
  "com.apple.keylayout.French": "fr",
  "com.apple.keylayout.Spanish": "es",
  "com.apple.keylayout.Italian-Pro": "it",
  "com.apple.keylayout.Italian": "it"
}
```

Slider Control

Group	Icon	Name (type)	Description
Other		Slider Control (Xcomp)	Draggable button to set a value

The **Slider** control provides a graphical thumb component that the user can drag to control the numeric setting of another component in your window, e.g. a volume control, progress bar, or a setting of some kind.

There is a control called ‘Slider Control’, as well as an External component called ‘Slider Pal’ (in the Deprecated group: see Deprecated Components). They behave in a similar way and have many properties in common. The standard built-in component is described here.

The current value of the slider is specified in the property $val; at design time you can enter a default value, and at runtime $val holds the current value according to where the slider is positioned. You can specify the range for the slider in the $min and $max properties. Most of the other properties are self-explanatory and handle the appearance of the slider component. The Slider reports 3 events: evNewValue, evStartSlider, and evEndSlider which you can detect in the $event() method for the component. These events all pass the current value of the Slider in the pNewVal parameter. As the user drags the Slider thumb the evNewValue event is trigered and pNewVal is sent to the $event() for the Slider.

For example, a window could have 3 sliders that let the user choose a color by setting the Red, Green, and Blue value of a field that is used to display the color. Each slider has the following properties set: $min=0, $max=255, $val=0, $markfreq=64, $bigrange=kFalse. The window contains 3 instance variables iRed, iGreen, and iBlue for the current Red, Green, Blue value, and next to each slider there is a display field showing the current value of the appropriate slider.

```
# $event() behind RedSlider
On evNewValue
  Calculate iRed as pNewVal
  Do $cinst.$objs.Red.$redraw()
  Do method $setcolor
```

The $event() methods for the green and blue sliders are almost identical except that they act on their respective color variables and display fields. The color of the display field in the window is set using a class method $setcolor as follows:

```
# $setcolor class method
# CurrentColor is name of the display field
Do $cinst.$objs.CurrentColor.$backcolor.$assign(rgb(iRed,iGreen,iBlue))
```

As a further refinement to the example, you could make the Red, Green, and Blue fields enterable to allow the user to choose the color either using the slider or by entering a value.

The only code you need to add is behind each of the entry fields; note that you need to set the value of the slider to the value the user enters into each field, as well as limiting the value entered to 255. For example the $event() method for the Red field is as follows:

```
On evAfter
  If iRed>255
    Calculate iRed as 255
    Do $cinst.$objs.Red.$redraw()
  End If
  Do $cinst.$objs.RedSlider.$val.$assign(iRed)
  Do method $setcolor
```

The $event() methods for the green and blue entry fields are almost identical except that they act on their respective color variables and entry fields.

Split Button

Group	Icon	Name (type)	Description
Buttons	OK▾	Split Button	Standard button with a dropdow

The **Split Button** control combines a standard button with a dropdown menu, allowing you to provide multiple, alternate actions grouped together in a single button control. The window control is very much like its JavaScript equivalent which is described here.

String and Data Grids

Group	Icon	Name (type)	Description
Lists	▤	String Grid	Grid to display character data or

String and Data grids can display data from a list variable in an enterable table format. String grids display character-based data only, whereas Data grids can display any type of data. You can scroll these grid types horizontally and vertically, and you can make the first column and/or first row non-scrolling headers if required. If you tab out of the last column in the last row in a data or string grid, a new row can be added to the grid.

String Grids

You can use string grids to display character-based data from a list. The row height and column width are set at design time, but columns can also be sized at runtime if the first row is fixed. String grids have the following properties

- **$dataname**
  the source list variable
- **$defaultheight** and **$defaultwidth**
  the default row height and column width in pixels
- **$cellleftpadding**
  allows you to add content padding to the left of all cells in the grid
- **$designcols** and **$designrows**
  number of columns and rows displayed in design mode
- **$fixedrow** and **$fixedcol**
  sets top row or first column as a fixed header or column
- **$extendable**
  if true a new row is added when you tab out of the last column of the last row
- **$columnwidths**
  specifies a comma separated list of column widths in pixels; when you assign a list to this property it must be in quotes, runtime only

A string grid instance has the read-only properties

- **$gridrows** and **$gridcols**
  the number of rows and columns
- **$gridhcell** and **$gridvcell**
  the current cell column and row number

When you create a string grid, set the $dataname property to the source list variable. Set the number of list rows and columns in $designrows and $designcols, then add the list-building method behind the grid field.

You can make the first column and row fixed and non-scrolling by setting $fixedrow and $fixedcol. This sets the first row and column of your list data as the column and row headers. In addition, setting $fixedrow lets you size the columns both at design and runtime by dragging in the column header. Dragging individual columns overrides any values set in $defaultwidth. If you want to have variable column widths at design time, but not have a fixed row, set $fixedrow back to false after sizing the columns. If you change $defaultwidth after manually sizing the columns, a message asks whether you wish to keep the non-default widths.

Data Grids

Data grids are very similar to string grids with regards to their appearance, but with some extra features.

- data can be of many types, even pictures
- the row height adjusts to fit the data
- column header names are added as a property

When you create a data grid, set $dataname to the source list variable, set $autosize if you want the row height to adjust to fit the data, then set the number of list rows and columns in $designrows and $designcols, and add the list-building method behind the grid field.

You use $fixedrow to adjust the column widths at design time. You can enter the column headings in the $columnnames property in the dialog that pops up when you click on the property dropdown; note you must set the $designcols property to the number of columns you require in the grid before being able to add the corresponding column headings.

With the $autosize property on, character-based columns will size to a maximum of 5 lines deep; for larger amounts of data cells will scroll. Columns containing pictures will size to fit the picture. Different data types are displayed in different ways in a data grid: Boolean data types become droplists with true/false options, and lists are shown as droplists. Character, number, date and all other types map to edit fields.

Binary and List data cells

If the property $showbinarylength is set to kTrue, the window class data grid displays the length of binary data, and also allows the droplist button for a cell to open a modal window to edit list, row and binary columns.

Programming Data and String Grids

String grid cells are normally enterable except those in a fixed row or column. The grid receives specific events when the user clicks in the grid; it does not receive evClick and only receives evBefore and evAfter when entering or leaving the field. When the user clicks in a data cell, two events are sent, as follows

- **evCellChanging**
  returns pHorzCell, pVertCell, and pCellData event parameters
- **evCellChanged**
  returns pHorzCell, and pVertCell event parameters

Note these events are not available for complex grids. The parameters pHorzCell, pVertCell are the column and row numbers and pCellData is a character variable holding the contents of the updated cell. You can use the evCellChanging event to validate cell data entered by the user. If you discard the event the data is not changed.

```
# $event() method for the grid field
On evCellChanging
  If pCellData = ''
    OK message {You must enter a value}
    Quit event handler (Discard event)
  End If
```

If the user tabs out of the last column of the last row and the $extendable is set, the evExtend event is sent with the parameter pRow. You can use this to set up the new row with default data or stop the grid extending, as follows

```
# $event() method for the grid field
On evExtend
  If $cobj.$gridrows > 20
    OK message {This grid cannot have any more lines}
    Quit event handler (Discard event)
  Else
    Calculate pRow.Column1 as Default Val
  End If
```

For the fixed column of a data grid, the function mouseover(kMLine) can be used to report the line number of where the drop occurs when the end user has dragged the row to reorder the list.

Scrolling Tips for String and Data Grids

The following properties control string and data grid scrolling.

- **$vscrolltips**
  if true, enables vertical scrolltips showing the current row number when scrolling; the scrolltip contains the value of column 1 of the current row while scrolling

- **$hscrolltips**
  if true, enables horizontal scrolltips showing the current column number when scrolling; the scrolltip contains the value of row 1 of the current column while scrolling

- **$cellbordercolor**
  the grid cell borders

- **$gridendcolor**
  the color of empty grid where no data appears at the end of the grid

- **$gridhcell** and **$gridvcell**
  return the current cell using (row,column) coordinates; the first row of the grid is row 1, the first column is column 1

You can replace the default scrolltips for string and data grids by intercepting the evScrollTip event, and providing your own scrolltip string, evScrollTip has three event parameters:

- **pIsVertScroll**
  if true, the current scrolltip is on the vertical scroll bar, otherwise it is on the horizontal scroll bar

- **pScrollPos**
  the list row number for vertical scrolling, otherwise the column number for horizontal scrolling

- **pScrollTip**
  the scrolltip text, if you do not assign a value the default scrolltip is used

You can also use the *Quit event handler* command with the discard event option if you do not want Omnis to display a scrolltip.

Subwindows

Group	Icon	Name (type)	Description
Subwindows	🗔	Subwindow	Embeds another window in the window

A **Subwindow** is a field that can contain another window class. You can put any window class into a subwindow field; in this context, the window class inside a subwindow field is referred to as the subwindow class, and the window containing the subwindow field is referred to as the parent window. The subwindow class can contain any number of fields or window objects, such as a group of radio buttons, a set of standard pushbuttons, or it might contain a single field only, such as a complex grid field. The window class can contain its own methods which in effect become the methods for the subwindow field. Subwindow fields let you design sets of window objects and their associated methods, store them as separate window classes, and reuse them on different windows as subwindow fields with all their variables and methods encapsulated.

In design mode, the subwindow field appears as a single object, so you cannot access the fields contained in the subwindow class. The title bar and size borders of the subwindow class are ignored.

In runtime, the fields contained in the subwindow field appear on the open window as standard fields and are part of the normal tabbing order. A subwindow field is a container field, but it does not contain the $objs and $bobjs groups like other container fields; its objects are treated as part of the parent window.

When you create a subwindow field, you need to set the $classname property to the name of a window class or select one from the droplist; normally you should leave the $dataname property empty. When you place the subwindow field it will resize to accommodate the subwindow class. You can edit the subwindow class at any time by right-clicking on the subwindow field and selecting Subwindow Class from the context menu.

If you enable the $nobackground property, under the Appearance tab, the background of the subwindow field becomes the same color and pattern as the parent window. Normally, the text style of individual fields inside your subwindow class is retained. However, you can force these fields to use the text style of the subwindow field if you enable their $subwindowstyle property in the original window class.

Opening a window containing a subwindow field or any number of subwindows creates an instance of each window, which belong to the same task as the parent window instance and contains all the variables of its class. Omnis calls the $construct() methods of all the subwindow classes first in tabbing order, then the $construct() method of the parent window instance. The reverse happens on closing the parent window, with the subwindows being destructed after the parent window instance. It is important not to include an *Enter data* command in a subwindow $construct() method as this affects the opening of the parent window.

You can send parameters to the subwindow's $construct() method by including a list of parameters in the $parameters property when you create or modify the subwindow field.

A subwindow instance inherits the properties and methods of its class and superclasses, as well as having the normal properties of a window field. They are available only within the instance and not to the parent window since the subwindow is private to itself. Within a subwindow instance, $cobj refers to the current internal subwindow field rather than the container field. From an internal field method, you can access subwindow field properties using $cobj.$abc, whereas subwindow class methods such as $control() must be accessed using $cinst.$control().

Floating Fields

Fields inside the subwindow can have their $edgefloat properties set so that they resize with the parent window.

There is a setting "floatWindowSubclass" (default true) which allows you to override the Subclass window floating behavior. The setting is in the "defaults" section in config.json. You can set this to false to revert to the previous floating behaviour with subclasses (i.e. not floating if width or height are overridden).

Subwindow Events

To the parent window, the subwindow is a single field and never has the focus, but does receive some events. Field events in the subwindow are sent only to the subwindow $event() method and not to the parent window. If the **nobackground** property is not set, click and scroll events on the subwindow are sent to:

- the subwindow $event() method
- the subwindow field $event() in the parent window
- the parent window $event() method
- the task $control() method

Mouse enter and mouse leave events are sent only to the subwindow $event(), while mouse up, mouse down on the subwindow are passed to the parent window.

Drag and drop

It is possible to drag and drop data to and from the fields inside a subwindow as though they were in the parent window, and also to and from the subwindow field itself provided $nobackground is off. When dropping data from the subwindow pDragValue will usually be set up, or alternatively you could use a custom $contents to hold the drag value (since a subwindow has no default $contents property).

You cannot use the 'drag field' mode to move an internal field out of a subwindow. You can use the 'drag field' and 'drag duplicate' modes to move or duplicate the complete subwindow. When you duplicate a subwindow field, a new instance of the subwindow class is constructed.

The drag and drop modes for the subwindow field belong to the field rather than to the window inside the field. They are therefore not known when the subwindow is designed, so the subwindow's methods need to either switch off unsupported modes in the $construct() method or be capable of supporting all modes.

Nesting Subwindows

You can nest subwindows up to 999 levels deep, although in practice you probably won't reach this limit. Beyond this level, the most deeply nested window/field is not set up when the window is opened and becomes a display field showing an error message.

A subwindow can contain a grid field and vice versa. When a grid contains a subwindow, there is only one instance of the subwindow in the grid field, and not one per line. When a row of the grid is redrawn the current field values are set up and the subwindow is redrawn. Therefore, a subwindow within a grid which displays instance variables will not work correctly, since all rows of the grid will share one set of instance variables.

A grid field cannot contain a subwindow which itself contains a grid. If this occurs the nested grid is not set up, and it becomes a display field showing an error message.

Subwindow instance notation

Subwindows have a runtime property $subinst. This is an item reference to the instance contained by the object. You can use its $methods group to manipulate the methods of the subwindow instance. For example:

```
Set reference iref2 to $cinst.$objs.Container_1016.$subinst
Set reference iref to iref2.$methods.//$submethod//
Calculate iref.$methodtext as 'Send to trace log {Submethod called}'
```

Changing the class for a subwindow

You can switch windows in a subwindow field at runtime by assigning a new window class name to the $classname property of a subwindow control. When you change the window class of a subwindow field, an instance of the new window class is opened (its $construct is executed) and the previous instance is closed (when $multipleclasses is kFalse).

If the $multipleclasses property of the subwindow field is set to kTrue and you change the window class of the subwindow field, the previous instance is cached and remains open but is temporarily hidden. In most circumstances this will be more efficient, but if at runtime you change a window class that is used in a subwindow field you would need to take steps to refresh the subwindow.

There are two messages which are sent to the subwindow instance if $multipleclasses is set to kFalse:

- $swhide()
  is sent just before an instance is going to be hidden, because $classname has been set to a different class.
- $swshow()
  is sent just after an existing instance, which had previously been hidden, is shown.

If you change $multipleclasses from kTrue to kFalse in an open window instance, any currently invisible instances are destructed. All instances are destructed when the main window closes.

Subwindow Examples

The following examples use subwindows containing tab strips and pushbuttons. In a window that contains a long list sorted alphabetically, you might want to allow the user to scroll the list with a single mouse click to show items starting with a given letter. This can be done using a subwindow containing either a tab strip or a set of Rolodex-type buttons.

There are two Subwindow example apps in the **Samples** section of the **Hub** in the Studio Browser, including the one using the Rolodex-type buttons which is described below.

To create the tab strip subwindow

- Create a window and put a *Tab strip* field on it
- On the tab strip field, set the **tabs** property to *A,B,C,..,Y,Z*
- You may want to change the **selectedtabtextcolor** property to highlight the tab selected
- Add a $control() method to the window and enter the single command

```
Do redirect $cwind
```

Subwindow field events are not passed beyond the subwindow field, but you can use the *Do redirect* command to redirect events to the $control() method in the parent window. The subwindow is completely generic and you could use it on any window.

- Create a new window and place a *Subwindow* field at the top
- Set the subwindow field **classname** property to the name of your tab strip window
- You may want to set the subwindow field **nobackground** property to false
- Place a List box field below the subwindow field and set its **dataname** to the name of your list variable and enter a **calculation** if necessary

You need to add the following methods to the parent window. The $construct() method builds and sorts the list using list commands, but you could equally use the $define() and $sort() list methods.

```
# $construct() method in parent window
Set current list cList
Define list {cCol1}
# build your list of data
Clear sort fields
Set sort field cCol1
Sort list
```

The $control() method in the parent window detects the tab strip event.

```
# $control() method in parent window
On evTabSelected
  Set search as calculation

    {upp(mid(cCol1,1,1)) = chr(64 + pTabNumber)}
  Search list (From start,Do Not Load Line)
  If flag true
    Queue scroll (Down,Page) {ListField}
    Redraw lists
  End If
```

The search calculation in the $control() method uses the *chr()* function to derive 'A' to 'Z' from pTabNumber of 1-26, and compares it to the value of the first column in the list using *mid().* When a matching line is found, it will appear at the bottom of the list box and *Queue scroll* pages down to bring it into view.

To create the Rolodex buttons subwindow

The following example describes a subwindow containing a set of pushbuttons with the letters of the alphabet. Rather than creating a window with 26 buttons manually, you can do it automatically using the notation. You can paste this code into any method, but set up the variables first, and run it to create a window *wSubWin*.

```
# Declare variables cWRef and cRef of type Item reference
# Declare variables cLeft and num of type Number
Do $clib.$classes.$add(kWindow,'wSubWin') Returns cWRef
# returns a reference to the new window class
Do cWRef.$height.$assign(60) ## edit this to change the height
Do cWRef.$width.$assign(330) ## edit this to change the width
Calculate cLeft as 5
For num from 1 to 13 step 1
  Do cWRef.$objs.$add(kPushbutton,10,cLeft,15,15) Returns cRef
  # returns a reference to the new object
  Do cRef.$text.$assign(chr(num+64))
  Do cWRef.$objs.$add(kPushbutton,35,cLeft,15,15) Returns cRef
  Do cRef.$text.$assign(chr(num+64+13))
  Calculate cLeft as cLeft+25
End For
```

Unlike the tab strip window described above, the parent window needs to receive the button text, which is not supplied as an event parameter, so *Do redirect* will only pass on the evClick. However, you can do this by calling a custom method, called $alphabutton() perhaps, in the parent class methods, and pass a parameter.

- Add a $control() method to you subwindow class containing the buttons, with the following code

```
On evClick
  Do method $cwind.$alphabutton($cobj.$text)
```

- Create a new parent window and place a *Subwindow* field at the top
- Set the subwindow field **classname** property to *wSubWin* and **nobackground** to kTrue
- Place a *List box* field below and set its **dataname, calculation,** and **multipleselect** properties

For the class methods in the parent window, the $construct() method is the same as the example above. The $alphabutton() custom method is similar to the $control() method above, but it has no event handling code and the search calculation is different

```
# declare parameter pChar of type Character
Set search as calculation {mid(cCol,1,1) = pChar}
```

To create a radio button subwindow

- Create a window with a set of radio buttons and declare a numeric variable, say iNum, for the $dataname property for each radio button
- Add the window as a subwindow field to your parent window

In the same way as for the previous examples, you can either pass up the event using *Do redirect* and get the ident of the button clicked from $cobj, or declare a custom method, say $buttonval, in the parent window, called by *Do method $cwind.$buttonval(iNum)*.

Switch Control

Group	Icon	Name (type)	Description
Buttons	○	Switch Control (Xcomp)	iOS style switch with animated s

The **Switch Control** has the appearance of an "iOS style" switch: when the switch is turned on or off (clicked) the round button slides across and the background changes color. The on/off state is assigned to the $switchon property. The following shows the OFF (left) and ON state:

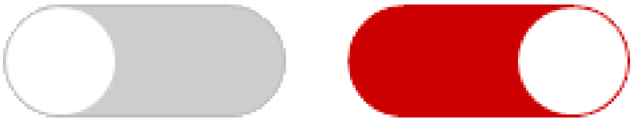


Figure 156:

The Switch Control is in the **Buttons** group in the Component Store (it is an External Component but is pre-loaded): note you may need to resize the control to make the background part visible. The Switch Control has the following properties (shown on the Custom tab in the Property Manager):

- **$switchbutton**
  the color of the round button part of the switch; the default color is white
- **$switchcolor**
  the background color of the switch when switched on; the default color is dark green
- **$switchon**
  true if the switch is on; setting this in design mode sets the default state when opening the window
- **$transparencywhenoff**
  the amount of transparency when the switch is turned off, an alpha value from 0 to 255; the default value is 50

You can test the value of $switchon in the $event method for the control to branch depending on its true/false value.

Tab Pane

Group	Icon	Name (type)	Description
Containers	▭	Tab Pane	Multiple pages or panes with tal

The **Tab pane** control provides separate panes, on each of which you can place any number of fields and background objects. You can switch panes in design and runtime modes by clicking on the tab belonging to the pane. You can set the position, number and style of the tabs, and whether the tabs have icons in the properties for the tab pane. When you create a tab pane, you can set $tabcount to specify the number of panes; you can click on each tab and add the fields and background objects as required. Note that a Tab pane can be used as a Side Panel.

Tab pane fields have the following properties:

- **$taborient** and **tabstyle**
  the position and style of the tabs; either at the top or bottom (kTopTabs or kBottomTabs), with square, rounded, or triangular shaped panes, either kAnimatedArrow, kDefaultPanes, kIDEPanes, kMacOS8Panes, kRoundedPanes, kSquarePanes, kTrianglePanes
- **$tabcount** and **$currenttab**
  number of tabs or panes, and the currently selected one
- **$imagenoroom**
  when insufficient room shows just picture and not text for each tab
- **$showimages**
  shows icons for tabs; specify icons under pane properties
- **$showfocus**
  shows the focus for the selected tab
- **$multirow**
  if true forces the tabs to stack rather than scroll when the field has many tabs
- **$forecolor, $backcolor,** and **$backpattern**
  sets the color and pattern of the area behind the tabs, not the tab panes
- **$selectedtabcolor**
  the color of the selected tab; defaults to kColor3DFace
- **$tabcolor**
  the color of non-selected tabs; defaults to kColor3Dface
- **$colortabselectedhighlightmacos**
  the color of the active tab on macOS only

An individual pane has the following properties, set under the ‘Pane’ tab in the Property Manager:

- **$movetab**
  allows you to move a tab in design mode, which is useful when adding new tabs and you need to reorder existing tabs
- **$tabcaption**
  text or label for the tab
- **$iconid**
  id of an icon from an icon set (or #ICONS or icon data file); you cannot use icons larger than 48x48 pixels for tabs (enable **$showimages** to show icons)
- **$tabtooltip**
  tooltip for the tab; you must enable the Omnis preference **$showwindowtips** to show object tooltips

Tab panes have the following methods:

- **$allpanes**
  $allpanes(rField[,bAllPanes]) gets/sets the all panes flag for the given field reference
- **$enablepane**
  $enablepane(iPaneNumber,bEnable) sets the enabled state of the pane
- **$ispaneenabled**
  $ispaneenabled(iPaneNumber) returns true if the pane is enabled
- **$ispaneshown**
  $ispaneshown(iPaneNumber) returns true if pane is visible
- **$listobjects**
  $listobjects([iPaneNumber=0]) when iPaneNumber is zero (the default), returns a list of all objects directly contained in the control. Otherwise it returns a list of all objects on the pane in iPaneNumber; see below

- **$panenumber**
  $panenumber(rField[,iPane]) gets/sets the pane number for the given field reference
- **$setpaneinfo** and **$getpaneinfo**
  sets the pane information using a three column list containing the pane information (tab label, tooltip, icon), or returns a list; see below
- **$settabinfo**
  $settabinfo(iTabNumber[,cTabCaption,cTabIcon]) sets caption and icon for the tab in iTabNumber
- **$showpane**
  $showpane(iPaneNumber,bShow) shows or hides the pane

When the user clicks on a tab at runtime, the field receives the event **evTabSelected,** with the parameter pTabNumber containing the number of the tab clicked. $currenttab changes to the current tab (and pane). Discarding the event will prevent the current tab from changing.

Tab pane fields can have a $control() method to control events for each of the contained fields.

To access individual tabs or panes using the notation, you must first set $currenttab. For example, to change the text on the second tab use

```
Do $cinst.$objs.TabPane.$currenttab.$assign(2)
Do $cinst.$objs.TabPane.$tabcaption.$assign('New tab text')
```

You can set $disabledfocus to kTrue to completely disable a tab pane field, that is, it will not receive the focus and cannot be tabbed to.

Styled Text

When the $styledtext property is kTrue, the text for the tabs can use styled text for the $tabcaption and $alltabcaptions properties to display styled text in the tab captions.

Listing the objects on panes

The $listobjects([iPaneNumber]) method returns a list of objects contained within the specified pane, including all foreground and background window objects. If iPaneNumber is omitted, the list contains information about the objects on *all panes* in the Tab pane field. The list has three columns: *object name* (empty for background objects), *ident* of the object, and *pane number*.

If you mark a field or object as "all panes", it will be included in the list regardless of the pane number specified.

Getting and Setting the Tab labels

You can get and set the text assigned to the tabs in a tab pane using the $getpaneinfo() and $setpaneinfo() methods.

- **$getpaneinfo**
  Returns a three column list containing the pane (tab) information; columns are the tab label, the tooltip, and the icon ID
- **$setpaneinfo**
  $setpaneinfo(lPaneInfo) sets the pane (tab) information from the specified three column list; columns are the tab label, the tooltip, and the icon ID; returns kTrue if successful

Tab style on macOS

On macOS, when using tab panes with kDefaultPanes as the tab style, the tab panes have the standard macOS appearance. However, the multi-row property is ignored for kDefaultPanes on macOS; this limitation on macOS applies to custom tab panes, as well as built-in windows that use tab panes, such as the Omnis Catalog.

Tab Strip

Group	Icon	Name (type)	Description
Navigation		Tab Strip	Set of tabs only that can be linke paged pane

A **Tab strip** (or Tabstrip) control contains a set of tabs only (it does not have pages or panes, like the Tab pane control), but is typically linked to a paged pane control. The tab strip offers similar functionality to a radio button group in that only one tab can be selected at a time. You could use a tab strip in conjunction with the paged pane field to hide and show a series of fields on your window.

You enter the text and number of tabs for the field in the $tabs property. Enter a text label for each tab separated by commas. For example, the text Tim,Sue,Bill will enable three tabs with the specified text.

Tab strip fields have the additional appearance properties:

- **$backcolor**
  sets the color of the area behind the tabs
- **$tabcolor** and **$selectedtabcolor**
  the color of the tabs, and the selected tab
- **$tabtextcolor** and **$selectedtabtextcolor**
  the color of the text on the tabs, and the selected tab text color
- **$showedge**
  if true shows the edge of the tab strip
- **$overlap**
  the overlap for the tabs in pixels
- **$tableftmargin**
  the indent for the left tab in pixels

The **$selectedtabtextstyle** property allows you to assign the font style of the currently selected tab, overriding the font style for all tabs set in $fontstyle (this property applies to all modes except kTabStripOriginal).

Events

Tab strips receive **evTabSelected** which you can handle in the same way as tab panes.

Tab Animation

The Tab Strip has a property, $squaremode, which has several different settings to provide alternative appearance and animation options (the $animateui library property must be enabled to allow the animation). The $squaremode property is set to kTabStripOriginal by default which means it has the same appearance and behavior as in versions prior to Studio 10.0.1. The other options include:

- **kTabStripSquare**
  the tabs have square corners and fill the entire control area, but there is no animation when the tab changes
- **kTabStripAnimSquare**
  the tabs have square corners and the tab change is animated
- **kTabStripAnimLine**
  the current tab is indicated with a *line* and the tab change is animated
- **kTabStripAnimDot**
  the current tab is indicated with a *dot* and the tab change is animated
- **kTabStripAnimRndSquare**
  the tabs have rounded square corners and the tab change is animated

The $animateui library property *must be enabled* to use the Tab Strip animations; if the preference is set to false, you can still use the square, line and dot options but they will not be animated.

The $verticaltabs property is available for all animated tab strip styles, plus when the tab strip has the focus, the Left, Right, Up, and Down keys can be used to change the current tab.

The animated styles have extra options for an icon and caption: $tabiconid and $tabcaption can be set under the 'Pane' tab in the Property Manager (like Page pane and Tab pane).

There is an example app to demonstrate the Tabstrip animation in the **Hub** in the Studio Browser, and on the Omnis GitHub repo at: https://github.com/OmnisStudio. Search for Omnis-TabStrip.

Adding Tabs

The **$addtab()** method allows you to add a new tab to a Tab Strip. The new tab is added after any existing tabs and its tab number is returned. The syntax is:

- $addtab(cTabName[,cTabIcon,cTabGroupName])
  adds a new tab to the tabstrip with optional tab icon and tab group
  cTabName - name of the new tab
  cTabIcon - name or ID of the icon for the new tab; you can include the size, e.g. "article+16x16"
  cTabGroupName - name of the group into which the tab is added; only applies if your tab strip has groups, see below

- $addtab(cTabName[,cTabIcon,cTabGroupName,bDoNotChangeTab])
  adds a new tab to the tabstrip with optional tab icon and tab group
  cTabName - name of the new tab
  cTabIcon - name or ID of the icon for the new tab; you can include the size, e.g. "article+16x16"
  cTabGroupName - name of the group into which the tab is added; only applies if your tab strip has groups, see below
  bDoNotChangeTab - if kTrue the current tab is unchanged, otherwise if kFalse or omitted the new tab becomes the current tab

When a new tab is added, by default the new tab becomes the current tab. However, passing **bDoNotChangeTab** as kTrue will suppress the tab change event leaving the current tab unchanged.

For example, the following code adds a new tab called 'Info' with the 'article' icon from the material icon set:

```
Do $cinst.$objs.tabs.$addtab("Info"),"article+16x16")
```

You can assign to the $tabs property to specify or change the tabs. For example, to add a third tab called Bill you can use:

```
Do $cinst.$objs.TabStrip.$tabs.$assign('Fred,Sarah,Bill')
```

Tab Strip Groups

You can arrange the tabs vertically into separate groups, but only when the Tab Strip is in **vertical mode** with one of the animated styles: kTabStripAnimSquare, kTabStripAnimLine, kTabStripAnimDot, or kTabStripAnimRndSquare. There is an example app called **Tab Groups** in the **Samples** section of the **Hub** in the Studio Browser.

The individual tabs are specified as a comma-delimited list in the $tabs property, but each tab has the $tabgroupname property, (under the Pane tab in the Property Manager), which specifies the group the tab belongs to, and $tabgroupcolor specifies the color used for the background of the tab group header. When you specify a group name in $tabgroupname, the group is added to the Tab strip and the tab is added to the group automatically.

The properties $tabgrouptextcolor, $tabgroupfont, $tabgroupfontstyle, and $tabgroupfontsize allow you to set the text color, font type, font style, and font size for the tab group headings.

Expanded Groups

The **$expandedgroups** property allows you to show all the groups and tabs in one view. When $expandedgroups is set to kTrue (default is kFalse), a Tab Strip will expand the groups displaying all groups and tabs. In this state the groups cannot be selected, they are just shown as headings. Note $expandedgroups is only supported in vertical mode. The following screenshot shows a Tab Strip in a non-expanded state (left) and expanded state (right):

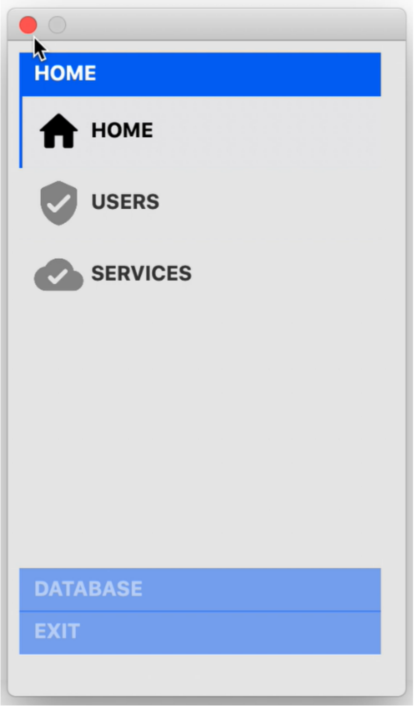


Figure 157:

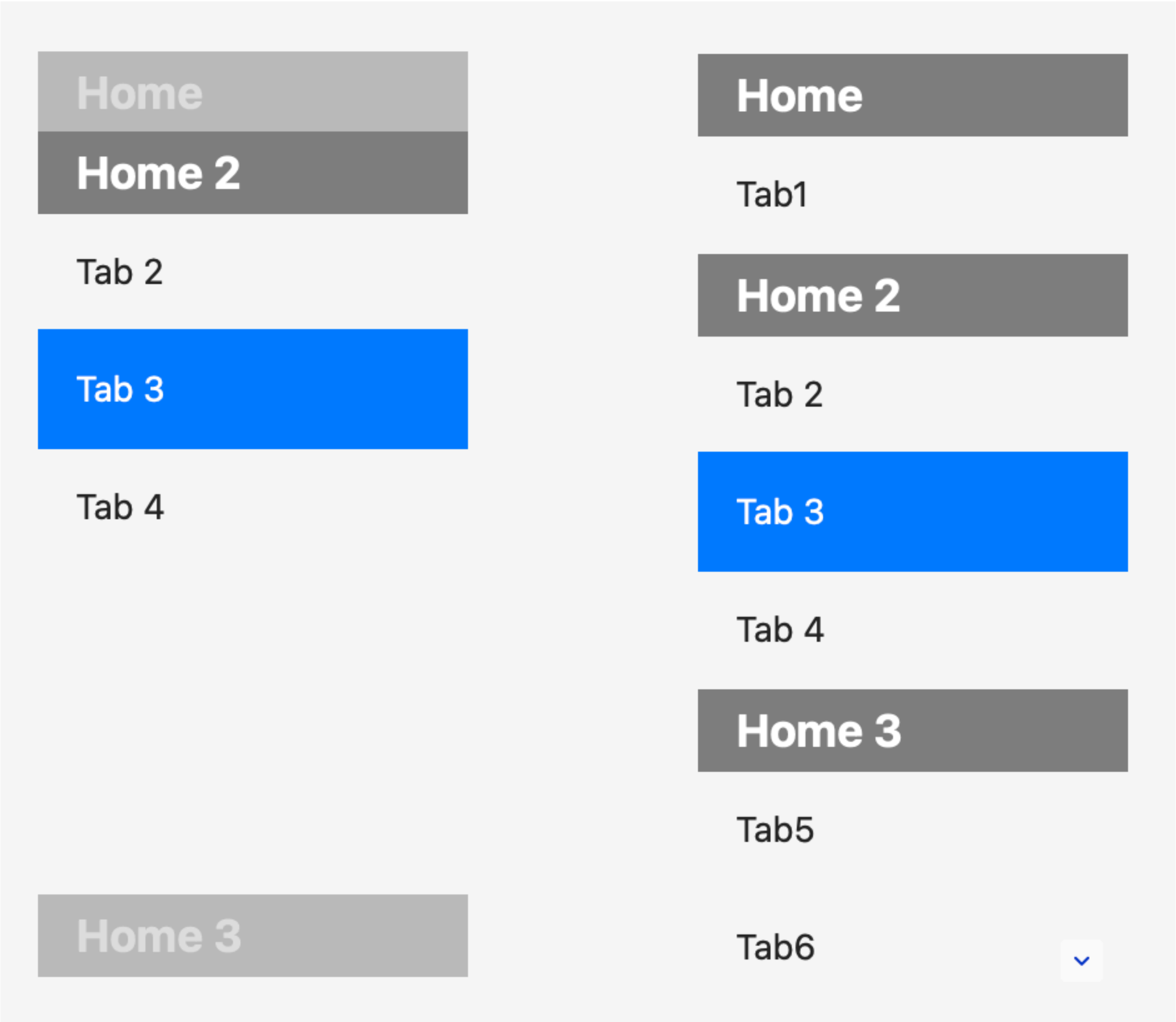


Figure 158:

Scroll Bars

A scroll box will be added to a Tab Strip in vertical mode when required, for example, when its groups are fully expanded and tabs at the bottom are hidden (applies when $verticaltabs and $expandedgroups are kTrue). In design mode, you need to click the scroll button to scroll the Tab Strip. In runtime, moving the mouse into the scroll area scrolls the control without needing to click.

Events

The **evTabGroupChanged** event is sent with the pTabGroup parameter when the control is about to change groups, which can be discarded if required.

Keyboard Navigation

When grouped mode is enabled and the Tab Strip has the focus, you can use the keyboard to navigate the tabs or groups: the Up and Down keys navigate between the *tabs* within the group order (the groups will open automatically as you navigate to a tab), while the Shift Up and Down keys will jump up and down between groups, opening the group as it gets the focus.

Close Boxes on Tabs

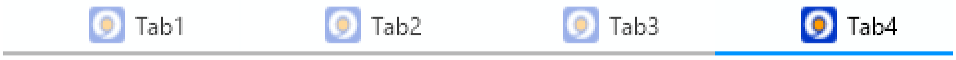
You can add a close box to all tabs by setting the **$showtabclosebox** property to kTrue. This property only applies to horizontal tab strip controls using any of the animated display options, e.g. when $squaremode is set to kTabStripAnimRndSquare. The close box option allows the tab strip to behave in a similar way to tabs in a web browser.

Pressing a close box generates an **evTabClosing** event, which can be trapped in the $event method for the control, or optionally it can be discarded.

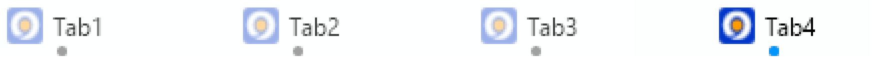
Animated Line and Dot Modes

The **$tabcolor** property specifies the color to be used for the line path or placeholder dots for non-selected tabs when $squaremode in a Tab Strip is set to kTabStripAnimLine or kTabStripAnimDot. In the following example, tab 4 is selected and the non-selected lines and dots are shown under tabs 1 to 3 since $tabcolor is set to kColor3DShadow.

kTabStripAnimLine



kTabStripAnimDot



By default the color in $tabcolor is kColorDefault which means the line path or placeholder dots are not shown (as in previous versions), so you need to select a color in $tabcolor to show the non-selected lines and dots.

Token Entry Field

Group	Icon	Name (type)	Description
Entry Fields	AbX	Token Entry Field	Field which tokenizes entered te

A **Token Entry Field** allows the end user to enter text which then becomes tokenized; a *token* is a single block of text that can be easily selected, moved, copied or deleted *as a single item*. The behavior of the Token Entry Field is very similar to the Recipients (To:) field in an email program, such as Apple mail or Google gmail, where you enter each email address as a block or token. When you start to type a name, a popup list will appear containing all entries that match what you typed and you can select one of the emails in the popup list, or you can complete the email address manually. You can then press Tab or Return to complete your selection and the text becomes a single block or token.

A token in the token entry field is defined to be a *character string* that either conforms to the syntax specified by a regular expression associated with the field ($tokenregexp), or matches one of a set of possible values in a list associated with the field ($tokenlist). As the token entry field operates on text, its $dataname needs to be set to a Character variable.

The token entry field allows the end user to enter and display a delimited list of tokens, separated from each other by a delimiter character (such as a comma, the default delimiter). Tokens also have an additional syntax (defined by RFC822), where they can be expressed in the form displayText<tokenValue>. In this case *tokenValue* is the actual token that conforms to the regular expression or a value from the token list, and *displayText* is the text displayed in the field when the item is tokenized (and not being edited). For example, consider the comma-delimited text string containing four tokens (email addresses):

Bob Mitchell<bob.mitchell@omnis.net>,Jason Gissing<jason.gissing@omnis.net>,Bob Whiting<bob.whiting@omnis.net>,colin.richardson

When displayed in the token entry field, it will look like the following, when the third token is currently being edited:

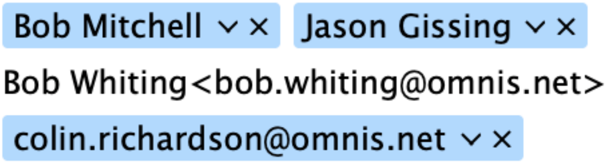

Bob Mitchell ⌄ ×
Jason Gissing ⌄ ×
Bob Whiting<bob.whiting@omnis.net>
colin.richardson@omnis.net ⌄ ×

Figure 159:

Using a Token Entry Field

As you start typing text into a Token Entry Field, the text will become a token value: if the field uses the additional display text syntax, then this needs to be part of the text you are entering. As soon as you press Return or a delimiter character, this terminates entry of the new token, the token is then displayed as either a valid token or an invalid token, and the caret is positioned after the token, ready for you to enter the next token.

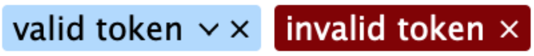


Figure 160:

There is a down arrow button for valid tokens, which you can press to open a menu for the token, while there is an x button for invalid tokens that you can press to delete the token.

The token entry field does not allow empty tokens by removing consecutive delimiter characters. In addition, the token entry field does not support overtype mode, and does not allow leading or trailing whitespace for tokens and will strip these automatically.

Once you have entered some tokens, you can use the arrow keys to navigate around the tokens, and then insert a new token by typing some text if required. While you are entering a new token value, mouse selection of text, and left arrow, right arrow and other relevant key combinations are restricted to the new token being entered, to prevent premature termination of edit mode for the new token. However, up or down arrow will terminate entry of the new token.

When the caret is positioned between tokens, pressing forward delete will delete the next token, and pressing backspace will select the previous token (this latter behaviour is how the macOS standard token entry field behaves). In addition, when the caret is positioned before a token, pressing return starts editing the token.

The token entry field also supports popup assistance, providing a list of tokens that match the text entered so far, allowing you to select one of the values in the popup list to populate the new token. The popup only displays after a pause in typing, the length of which can be configured in config.json (“defaults”, “tokenEntryPopupDelay”, default value 500 milliseconds). If there is a single valid token selected, or if the caret is positioned before a valid token, pressing Alt/Option+Return opens the menu for the token if present.

You can select a token or contiguous range of tokens using the arrow and other keys, like any other objects, so that you can delete them or copy them to the clipboard for example. Double-clicking on a token (when the Shift key is not pressed) starts editing the token. The Undo option works as you would expect for an Entry field.

You can use drag and drop to re-order tokens, or to move or copy tokens from one token entry field to another. In order to do this, a couple of lines of Omnis code are required (described below).

Properties

The token entry field supports many of the standard properties supported by other entry fields. The $objtype property for the token entry field has the value kTokenEntry. The $dropmode property has a new constant kAcceptTokenEntry to accept drop data.

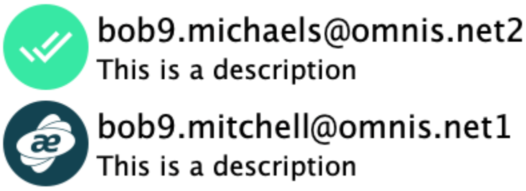
$tokenlist

The $tokenlist property is the name of the list variable containing all possible token values which is used for popup assistance and token validation. If omitted or empty, the list for popup assistance is obtained by sending the evGetTokenList event.

When specified, the $tokenlist can have just one Character column containing all the valid token values, possibly including some displayText. The $tokenlist can contain more than one column, in this case it must have a column with the name 'name' that contains all of the valid token values, possibly including some displayText. The list can optionally have additional character columns named 'desc' and 'iconid', that contain a short description of the token, and an icon to represent the token.

You may need to consider performance when deciding whether or not to use $tokenlist. Performance is affected by a combination of the number of possible token values, and the likely number of tokens entered.

The entry field popup will always display either the token text or displayText. In addition, the popup will display the desc value and the 32x32 icon with the specified icon id if either or both of these columns are present. The popup draws the desc column using the same font as the token entry field, with a point size 2 smaller (unless the token entry field font point size is less than 7, in which case it uses the same font). For example:



bob9.michaels@omnis.net2
This is a description
bob9.mitchell@omnis.net1
This is a description

Figure 161:

$tokenregexp

The $tokenregexp property is a regular expression used to validate the syntax of a token. If $tokenlist has a list of all possible tokens, you can still use $tokenregexp for a pre-validation step that reduces searches of $tokenlist.

The regular expression in $tokenregexp uses the new PCRE2 implementation. If you omit both $tokenlist and $tokenregexp, all token values are valid.

$tokencase

If true, tokens are case sensitive. This affects both searches of the token list and execution of the regular expression.

$tokenmenu

The $tokenmenu property contains the contents of the token menu. If specified, all valid tokens have a dropdown button that can be used to open this menu (as a context menu). The selected token is available as the pToken event parameter of evOpenContextMenu.

You can distinguish between the user using the context menu and the user using the token menu, because pToken has the value #NULL when using the context menu.

Note also that pToken includes the displayText as well as the token value if the token has a displayText component.

$showtokendeletebutton

If $showtokendeletebutton is true, all tokens have a delete button when the control is enabled.

The $tokendelimiters property contains one or more delimiter characters which are used to separate tokens (default is a comma). Each character specified can be used to separate tokens from each other in the data stored in $dataname. The first delimiter character is the default that the control uses when it needs a delimiter.

Note that this means that token values and displayText cannot include any of the delimiter characters, nor can they include < or >.

Token Colors

The token entry field has 4 token-specific color properties: $validtokenbackcolor, $validtokentextcolor, $invalidtokenbackcolor, $invalidtokentextcolor.

Each of these can be set to kColorDefault, which means use the corresponding entry in the token section of appearance.json (this is a new section for this control).

Events

The token entry field supports the same standard events as the Multi Line Entry Field, with a few exceptions noted here. It does not generate evClick, and will only generate evDoubleClick when the control is read-only. In the latter case, if $allowcopy is kTrue, it will only generate evDoubleClick when double-clicking on a token.

As the control does not scroll horizontally, it does not generate evHScrolled.

evGetTokenList

The evGetTokenList event is sent to the token entry field when $tokenlist is not specified, to get the list of possible token values for the popup. This event has two event parameters:

- pNewText: The text entered by the user, for which the list of possible token values is required.
- pTokenList: The list of possible token values to be displayed by the token entry popup. Assign this parameter when processing evGetTokenList.

The list to return via pTokenList has the same characteristics as $tokenlist. Note that whereas the token field sorts the list of values to display in the popup when using $tokenlist, the token entry field does not sort the list returned via pTokenList.

A simple example:

```
On evGetTokenList
 Do pTokenList.$define(cTokens)
 For #1 from 1 to 10
   Do pTokenList.$add(con(pNewText,"Test",#1))
 End For
```

evTokensAdded and evTokensDeleted

When the $sendtokenevents property is set to kTrue (default is kFalse), the control sends the events evTokensAdded and evTokensDeleted.

- **evTokensAdded**
  sent to the token entry field when one or more tokens have been added (if $sendtokenevents is true)
- **evTokensDeleted**
  sent to the token entry field when one or more tokens have been deleted (if $sendtokenevents is true)

Both of these events have an event parameter, pTokenChanges which is a list of tokens that have been added or deleted, comprising two or three columns: name (token), display (display text) and optionally the tag.

Methods

$gettokens

$gettokens([bSelOnly=kFalse, bIncludeInvalid=kFalse, bIncludeDisplayString=kFalse, bSort=kFalse, bRemoveDuplicates=kFalse, bSplit=kFalse])

Returns a single column list of the tokens stored in the control.

- bSelOnly. Only return selected tokens.
- bIncludeInvalid. If true, return all tokens, otherwise just return valid tokens.
- bIncludeDisplayString. If true, the values added to the list include the displayText component.
- bSort. If true, sort the returned list in ascending order. Sorting uses $tokencase to determine if case-sensitive sorting is required.
- bRemoveDuplicates. If true, remove duplicate entries. Note that if bRemoveDuplicates is kTrue bSort must also be kTrue.
- bSplit. If kTrue, the returned list has two or three columns: name, display and optionally the tag. The bIncludeDisplayString parameter is ignored if bSplit is true.

$droptokens

$droptokens([bRemove=kFalse, bSetFocus=kTrue])

Call this during evDrop for the token entry field, to insert dragged tokens into the current drop location, optionally remove them from the drag source, and optionally set focus to the drop destination. Returns true for success.

- bRemove. If true, remove the dragged tokens from the drag source, if it is a token entry field.
- bSetFocus. If true, set the focus to the destination token entry field where the drop is occurring, after moving or copying the tokens.

You can set up the token entry field to accept data dragged from other controls, and if that data is text it can be used with $droptokens(); in this case, bRemove is not applicable. For example, in $event for a token entry field:

```
On evDrop
  Do $cobj.$droptokens(kTrue)
```

In this case, you need to set $dropmode to either accept all, or include kAcceptTokenEntry.

Token Tags

Tokens can have a **tag,** which is a character string that the application can use to identify the source of the token, or some other information about the token. Although the tag is part of the data, it is not visible to the user.

To use tags, set the $tokentagseparator property, which is a single character that separates the token tag from the rest of the data for a token. The property defaults to empty, meaning tags will not be used. You can enter \t and \f to use chr(9) and chr(12) respectively. Using ~ as the tag separator, each token can be one of the following, depending on whether tags are being used:

- tokenValue
- displayText<tokenValue>
- tokenValue~tag
- displayText<tokenValue>~tag

You should ensure that the tag separator and < > characters used when including display text are not part of the displayText or tokenValue.

The characters \t and \f can be entered in the $tokendelimiters to use chr(9) and chr(12) respectively. Using \t or \f allows the unambiguous use of JSON syntax characters in tags.

When $canedittokens is kFalse (default is kTrue), the end user cannot edit a token as text (by double clicking on it, or pressing return when it is selected); however, you can type some text to cause the token popup menu to display. When using a tag in the token, data you would typically set $canedittokens to kFalse, because otherwise the tag could become meaningless if the user edits the token data.

Trans Button Control

Group	Icon	Name (type)	Description
Buttons	👆	Trans Button (Xcomp)	A ‘rollover’ type button

The **Trans Button** control implements a ‘rollover’ type button. As the mouse enters the control a different icon is displayed. The control supports a transparent background.

The ‘on’ and ‘off’ icons are specified using the $insideicon (on or over state) and $outsideicon (off state) properties. The text or label for the button can be specified in $text. The text can be boldened when the mouse enters the button by enabling the $boldover property.

When set to kTrue, the $nodrawhotrect property prevents the rectangle drawing during hot tracking: the property is set false by default for compatibility, which means the rectangle is displayed. Note that the hot rectangle is not displayed on macOS, so this property has no affect when running on macOS.

Transform Control

Group	Icon	Name (type)	Description
Other		Transform Control (Xcomp)	Adds animation or effects to wir objects

The **Transform** external component allows basic non-blocking animation and transformation effects to be added to window classes and window objects. (Note this is not for JavaScript objects on remote forms).

Transformation can be applied to any numeric window or object property accessible via standard Omnis notation, including background, foreground and external components. Transformation cannot be applied to attributes that require character values (e.g. text and date fields) and cannot sensibly be applied to attributes that use constant values (e.g. colors and Boolean values).

Note: there is a tech note on the Omnis website that provides more information and example libraries for the Transform component:

https://www.omnis.net/developers/resources/technotes/tnxm0004.jsp

How does it work?

During standard Omnis method execution, assignment of property values is normally actioned immediately. The transform object works by making these assignments incrementally using a specified number of steps. The delay between each step can be specified and it is also possible to specify a 'convergence effect' to be used. Convergence allows a number of additional animation steps to be added that slow or graduate the animation to give the desired visual effect.

There can be multiple transform objects on a single window, each with its own set of 'state' methods and animation settings. Each object uses its own internal timer values, allowing multiple transform objects to execute independently of one another.

Adding a Transform Object

The Transform component is in the **Other** group in the Component Store. If you double-click on the component, you will note that aside from $construct, $destruct and $event, no default methods are shown. The transform component's primary method *$transform()* is private and should not be overridden.

Aside from being placed on the window, the transform component has no visual display capability so it may be desirable to set the component's $visible property to kFalse or position it beyond the window's edge.

Creating Transform States

The transform object uses methods to encapsulate each 'target state', that is, after the transform object is placed on the window, one or more methods are created inside the object that represent each state. Each method line takes the form of an assignment statement, e.g.

```
Do $cwind.$objs.object.$attribute.$assign(value)
```

Calculate…as statements may also be used although the transform component will attempt to convert these to Do…$assign() statements. Non-assignment statements, including but not limited to computational, control and conditional statements will be ignored by the transform object and should be avoided.

Assignment statements can include full notation, e.g. $root.$iwindows.myWindow… as well as contextual notation using $cinst and $cwind. Square bracket notation and other context-sensitive addressing may not be used on the *left-hand-side* of the assignment as the underlying timer object will not have access to context-specific information during execution.

Assignment *values* can include literal values and instance variables including those derived from list and row variables as well as context-specific notation, such as $cwind. This is possible since acquisition of the current and target values (as well as expansion of $cinst/$cwind) is performed during initialization of the transformation.

Wait Statements

When parsing a state method, the transform component additionally recognizes 'wait' statements. These are written in to the method as comments and take the form:

```
# wait n
```

where n specifies a number of animation frames to wait before proceeding. Wait statements create the effect of staggering the commencement of animation for any statements that follow.

Invoking Transformation

To invoke transformation to a given state, you have to call the object's $transform() method, passing the name of the target method as a parameter, for example:

```
Do $cwind.$objs.myTransform.$transform('$state1') Returns #F
```

Note that it is not sufficient to simply invoke the object's state method directly.

Aside from their use by the $transform() method, there is nothing to prevent transform object methods from being called explicitly by other window objects or via the transform event methods. If you add non-assignment statements to such methods and call them explicitly, these methods will execute as normal. This also allows various transform states to be tested if required.

If $transform is called on an object that is already executing, the current transform terminates and the new transform is calculated based on the current values of the window properties. It is also possible to invoke $transform() from either of the evTransformBegin or evTransformComplete event messages although this should be carried out judiciously in order to avoid unwanted recursion.

Transform Object Methods

The transform object supports a single method that analyzes the specified target 'state' method, builds a list of applicable assignment statements and compiles them into blocks of statements to be executed during each animation frame. The compiled statement list is then passed to an internal timer object which processes the list until transformation is complete.

Method	Description
$transform()	$transform(cState) invokes transformation from the window's current state to the specified state. cState is the name of a private or public ($) method inside the transform object. $transform() returns kTrue on success, kFalse otherwise.
$event()	The transform object supports two custom event messages:evTransformBegin and evTransformComplete.evTransformBegin is called immediately when $transform() is called. evTransformComplete is called when the transform execution list is exhausted.In either event, the method name representing the target state is passed via event parameter 2.If $transform() is called on an object that is already executing, evTransformComplete is not called for the current transform and evTransformBegin will be called for the new target state instead.

Transform Object Properties

Property	Description
$animdelay	The delay in milliseconds between each frame of the transform animation. The default value is 20ms.

Property	Description
$numsteps	The number of steps required to complete the transform (default 20). Note that when non-linear convergence is specified, this is an approximation only since additional steps are automatically added to facilitate the decreasingly smaller step sizes which occur during convergence.
$convergence	Specifies the type of convergence to be used as animation completes. The default value; kConvergeSine causes the step size to decay gradually/sinusoidally. kConvergeLinear turns-off convergence and the step size will be constant. kConvergeOvershoot employs decaying sinusoidal convergence to create an overshoot/bounce effect.

Tree List

Group	Icon	Name (type)	Description
Lists		Tree List	Displays list data in an expandab hierarchy

There is a standard built-in component called ‘Tree list’, and an External component called ‘Tree List’ (in the Deprecated group: see Deprecated Components). They have many properties and events in common. The standard built-in component is described here.

A **Tree List** provides a graphical way of displaying a hierarchy of items, defined as follows:

- Each item in the hierarchy is a node.
- The node at the top of the hierarchy is a root node.
- A node can contain one or more child nodes, and each of these child nodes can contain its own child nodes.
- If a node does not have any child nodes, it is a leaf node.
- The node that contains a child node is referred to as the parent node of the child.
- Each node in the tree can have at most one parent node. If a node does not have a parent node, then it is a root node.

An example of such a hierarchy is the set of disk volumes on a computer, and the folders and files they contain:

- The root nodes are the disk volumes.
- There is a child node for each folder and file.
- The files are leaf nodes
- The parent node of each file or folder is the folder or disk volume containing the file or folder.

Each node in the tree has a *node name*, which is drawn as text representing the node. In the above example, the node name would be the volume, file or folder name.

Each node has an expand/collapse box which has two states, expanded and collapsed. When the box is collapsed, the children of the node are not visible; the user clicks on the box to expand the node, and show the children. When the box is expanded, the children of the node are visible; the user clicks on the box to collapse the node, and hide the children.

A tree list can also have additional columns. In this case, the node names of the tree are in column 1, while the additional columns of data are columns 2 and greater. A tree with additional columns of data inherits some of the appearance and behavior of a headed list box. Each node of the tree has an associated row variable containing the data for columns 2 and greater.

The appearance of tree lists is governed by a number of properties for the tree and for individual nodes, which you can set up either in design mode or using the notation. You can select a specific icon for each node, and for the expand/collapse box, and specify the color for the node name. You can also show lines connecting the nodes and change the horizontal and vertical spacing to accommodate large icons. The position and state of the node icons can also be set.

The tree Appearance properties are

- **$treelinehtextra**
  sets the distance in pixels between lines in the tree; useful when nodes have large icons

- **$treeleftmargin**
  the distance in pixels from the left before the tree starts drawing

- **$treeindentlevel**
  the distance in pixels between levels of nodes in the tree; useful when nodes have large icons

- **$defaultnodeicon**
  node icon id used by all nodes with a zero **iconid**.

- **$expandcollapseicon**
  the icon id used for the expand/collapse box; defaults to the +- icon

- **$showhorzlines** and **showvertlines**
  if true show connecting horizontal and vertical lines

- **$shownodeicons**
  if true shows node icons

- **$nodeiconpos**
  controls position of the expand/collapse box if a node has children: kIconOnNode next to node icon or node name; kIconOnLeft on the left of the tree; kIconSystemSet according to the operating system (under MacOS left side of tree, under Windows and Unix next to the node icon or node name)

- **$treenodeiconmode**
  controls whether the normal or checked state of a multi-state node icon is drawn; kNodeIconFixed to use the $checked property of the node, kNodeIconLinkExpand to draw the checked state if the node is expanded, kNodeIconLinkLine to draw the checked state if the node is the current line.

- **$designcols**
  The number of columns. For a tree list with additional columns, set this to a value greater than one

- **$columnnames**
  A comma separated list of column headings for the tree list header

- **$hideheader**
  A Boolean. When false, the tree list displays a header for its column(s).

- **$showcolumnlines**
  When true, the tree list draws lines between its columns

- **$boldheader**
  When true, the tree list draws the text in the header using a bold font

- **$defaultwidth**
  The default column width in pixels

- **$headerfillcolor**
  The color used to fill the header

- **$hilitecolumns**
  When true, additional columns are highlighted when their node is the currently selected node. This defaults to true.

- **$showheaderlines**
  If true (the default), header separator lines are drawn in the header

You can use 32x32 or 48x48 size icons for tree list nodes. You need to turn on the $useiconsize appearance property of the tree object to assign larger icon sizes to tree lists. You may also need to adjust the $treelinehtextra and $treeindentlevel properties to accommodate the larger icons.

Tree List Node Icons

Nodes in a Tree list have an *expand/collapse box*, drawn using an icon from an iconset (or #ICONS or icon datafile in legacy apps). Each node in the tree can also have a node icon. This icon can be a multi-state icon, and properties of the tree list and the node determine if the normal or checked icon is drawn.

The **$iconcolor** property for a tree node sets the icon color when using a themed SVG icon. The **$defaulticoncolor** property for a tree list control sets the icon color when using themed SVG icons *and the $iconcolor property of the item is kColorDefault*. If $defaulticoncolor is also kColorDefault, then themed icons use the text color.

Tree List Hot Tracking

Hot-tracking behavior for a tree list can be enabled using the $hot property. Hot-tracking means that nodes are underlined when the mouse moves over the text for the node, plus the color of the node text changes to the system colour kColorHotlight. When you click on the highlighted text, the node expands and any previously expanded node will collapse, provided it has the same parent as the expanding node. The $hot property can have three values: kHOTplatformDefault, the field has the default hot-tracking behavior for the current operating system (hot-tracking for XP with themes only); kHOTnotHot, the field does not have hot-tracking behavior; kHOThot, the field has hot-tracking behavior.

Populating a Tree List

You can enter default entries for the tree list using the $treedefaultlines property. You can also populate the tree list at runtime using various methods described later in this section.

To enter default lines in a Tree list, click on the button displayed for the $treedefaultlines property in the Property Manager; this displays the default lines dialog which lets you build a tree by adding root nodes and child nodes. You can edit the node names, change the icons, and add child levels. Right-click on a node to modify it, using the following options:

- **Always Show Expand Box**
  shows an expand/collapse box even when there are no child nodes: this toggles the node property **showexpandalways**
- **Enterable**
  lets the user edit the node name
- **Node Color**
  presents a palette to choose node color; restored by **Default Color**
- **Node Icon**
  displays the available icons; restored by **Clear Icon**
- **Node Ident**
  lets you enter an ident number, cleared by **Clear Ident**

Tree Node Properties

In addition to the Tree List object itself, you can use the notation at runtime to manipulate the properties of a node. A node has the properties

- **$showexpandalways**
  if true the node will always draw an expand/collapse box; useful for populating a node on the evTreeExpand event
- **$iconid**
  id for the node icon: if 0, the node displays the **defaultnodeicon** for the tree list, or no icon if **defaultnodeicon** is 0.
- **$ident**
  a long integer you can use to identify the node. The **ident** value can be any value you like, and it enables you to program the behavior of the tree list via events and methods.
- **$textcolor**
  the color in which the tree list draws the node name: if kColorDefault, the tree control’s **textcolor** is used
- **$drawinactive**

When true, the tree list draws the node name using the gray text color. This overrides the **textcolor** property. The default value of this property is false.

- **$name**
  node name: the text drawn in the tree list for the node
- **$enterable**
  if kTrue, the node name can be edited by the user at runtime. To edit the node, the user clicks on the text for the current line.
- **$seedid**
  a unique number assigned to the node when Omnis creates it. The **seedid** is only unique within the context of a single tree list object.

- **$nodeparent**
  returns an item reference to the node's parent node

- **$isexpanded**
  true for nodes which are in the expanded state

- **$checked**
  if true, the node is "checked". Depending on the **treenodeiconmode** property of the tree list, this controls how the node icon (if any) is drawn.

- **$level**
  returns a number indicating the indent level of the node: level 1 indicates root nodes

- **$tag**
  A user value stored with the node. You can use this to store additional information with the node. Omnis does not use this information. It is there to help you program the tree.

- **$rowdata**
  Used for a tree list with additional columns, this is a row variable value containing the data for columns 2 and greater.

Tree List Animation

The Tree List has the $animateui property, which means that when enabled the contents of the list will display by dropping down gradually as you open the control. If $animateui is disabled the tree list content drops down instantly, as in previous versions.

Tree List Methods

Tree list methods include

- **$clearallnodes**()
  clears the tree of all nodes

- **$count**()
  returns the number of root nodes in the tree

- **$add**(name[,ident,rowForColumns])
  adds a new root node to the tree: ident is optional and defaults to 0; rowForColumns is optional, and is a row variable containing the data for additional columns. $add returns an item reference to the new node

- **$remove**(itemref)
  deletes the child node identified by tree node item reference itemref

- **$getvisiblenode**(iVisLine)
  returns an item reference to the node for the visible line number iVisLine

- **$findnodename**(itemref, name, recursive, ignoreCase)
  searches either the tree (pass itemref = 0) or the node itemref for a node which has the specified name. recursive is a Boolean which controls whether the search only includes immediate children, or searches the children of children and so on; pass a true value for a recursive search. ignoreCase is a Boolean: pass a true value to ignore case when comparing names. $findnodename returns an item reference to the first matching node found, or NULL if no matching node exists.

- **$movenode**(rItem,rItemMoveAfter[,bMoveNode]) moves rItem to after rItemMoveAfter, and returns kTrue if successful. bMoveNode lets you specify where the node is moved to and can have the following values (a constant):

Constant	Description
kMoveNodeAfterDst	moves rItem, after rItemMoveAfter - nodes all operate on the same level
kMoveNodeToDstFirstChild	moves rItem into rItemMoveAfter and sets rItem as the first child node
kMoveNodeToDstLastChild	moves rItem into rItemMoveAfter and sets rItem as the last child node

Node Methods

Nodes also have methods, which include

- **$clearallnodes()**
  clears all nodes beneath the node
- **$count()**
  returns the number of nodes immediately below the node
- **$add**(name[,ident,rowForColumns])
  adds a new child node beneath the node: ident is optional and defaults to 0; rowForColumns is optional, and is a row variable containing the data for additional columns. $add returns an item reference to the new node
- **$expand()** and **$collapse()**
  expands or collapses the node
- **$first()**
  returns an item reference to the first child node of the node

Moving tree list nodes

You can move a tree list node using the $movenode() method. $movenode(rItem,rItemMoveAfter[,bAddAsChild=kFalse]) moves rItem to after rItemMoveAfter, and returns kTrue if successful. The optional bAddAsChild Boolean parameter can be used to insert the node rItem as a child of rItemMoveAfter (if set to kTrue) or inserted at the same level (kFalse, the default). For example:

```
Set reference node1 to $cinst.$objs.tree.$findnodename($cinst.$objs.tree.$first(),"Node2",kTrue)
Set reference node2 to $cinst.$objs.tree.$findnodename($cinst.$objs.tree.$first(),"Node5",kTrue)
Do $cinst.$objs.tree.$movenode(node1,node2)
```

Interchanging Data with Lists

Using the methods $setnodelist() and $getnodelist() you can either populate the whole tree or a node from a list variable, which must contain a sorted list, or retrieve data from the tree to a list variable.

- **$setnodelist**(listmode, noderef, listname)
  lets you populate the tree or a tree node from a list: listmode is one of kRelationalList, kFlatList and kTreeColList; noderef is either zero to populate the entire tree, or a node reference; listname is a list variable e.g. tList
- **$getnodelist**(listmode, noderef, listname)
  lets you retrieve information from the tree, or from a node and its children into a list: listmode is one of kRelationalList, kFlatList and kTreeColList; noderef is zero to retrieve the entire tree, or a node item reference; listname is a list variable e.g. tList

For a relational list (kRelationalList), you supply list data such as:

Node		
RootNode	Child 1	
RootNode	Child 2	Child 1
RootNode	Child 2	Child 2
RootNode	Child 2	Child 3
RootNode	Child 3	
RootNode 2	Child 1	
RootNode 2	Child 2	

To populate the whole tree from a relational list, you would use the line

```
Do $cwind.$objs.TreeList.$setnodelist(kRelationalList,0,tList)
```

In this case the tree contains all the information but the nodes are all in the default state.

The flat list option (kFlatList) lets you specify the node property settings $iconid, $ident, $enterable, $isexpanded, and $textcolor as the final 5 columns of the list. For example:

Name	Name	$iconid	$ident	$enterable	$isexpanded	$textcolo
RootNode		0	100	0	0	0
RootNode	Child 1	0	101	0	0	0
RootNode	Child 2	0	102	0	0	0
New Root		0	200	0	0	0
New Root	Child 1	0	201	0	0	0
New Root	Child 2	0	202	0	0	0
Last Root						

The $isexpanded value can be the sum of the constants kTREEnodeExpCollapseAlways and kTREEnodeExpanded; setting it to value 0 or 1 will display the node collapsed, value 2 will show the node expanded:

- kTREEnodeExpCollapseAlways
  indicates that the node always has an expand-collapse box.
- kTREEnodeExpanded
  indicates that the node is currently expanded.

The command

```
Do $cwind.$objs.TreeList.$setnodelist(kFlatList,0,tList)
```

sets the contents of the tree list using the list. This example assigns the list to an existing node:

```
Set Reference CurrentNode to TreeRef.$currentnode
Do CurrentNode.$setnodelist(kFlatList, CurrentNode,tList)
```

To retrieve data from the tree, $getnodelist() does the opposite to $setnodelist() and copies data from the tree to a list. There is no need to define the list first. This line retrieves the whole tree and its node properties to a list:

```
Do TreeRef.$getnodelist(kFlatList,0,tList)
```

This code retrieves the current node data but no node properties to a list:

```
Set Reference currentNode to TreeRef.$currentnode
Do TreeRef.$getnodelist(kRelationalList,currentNode,tList)
```

Finally, the kTreeColList option allows you to use the same information as kFlatList, together with the data for additional columns. There is one extra column required by the kTreeColList option. The column is itself a row, and it contains the data for columns 2 and greater of the tree. For example:

Name	Name	Row	$iconid	$ident	$enterable	$isexpanded
RootNode			0	100	0	0
RootNode	Child 1		0	101	0	0
RootNode	Child 2		0	102	0	0
New Root			0	200	0	0
New Root	Child 1		0	201	0	0
New Root	Child 2		0	202	0	0
Last Root						

Each row of the list has a row variable value in the column Row.

Node Tooltips

The **$getnodetooltip**(rNoderef) method is called (if $tooltip is empty) to get the tooltip to display when the mouse is over the specified node rNoderef. It returns either a styled text tooltip string, or an empty string (meaning use the default tooltip text). To implement this method, right click on the method name in the method tree, and override it.

Tree List Events

The $event() method for tree lists receives event messages in response to user actions in the tree.

The evTreeExpand event indicates that a node is about to be expanded and provides a reference to the node in pNodeItem. You can use this to populate a node using $add(), for example:

```
On evTreeExpand
  Set Reference NewNode to pNodeItem.$add('NewNode', 100)
  Calculate NewNode.$textcolor as kRed
```

If you have set up your nodes as default lines as described above and given them idents, such as:
Windows 100
Reports 200

you can expand the node from the appropriate list ($setnodelist is described later).

```
On evTreeExpand
  Set Reference TreeRef to $cwind.$objs.TreeList
  Switch pNodeItem.$ident
    Case 100
      Do TreeRef.$setnodelist(kRelationalList,pNodeItem,tWinList)
```

The evTreeCollapse event indicates a node is about to be collapsed and provides the reference in pNodeItem. You can use this to clear child nodes, for example:

```
On evTreeCollapse
  Do pNodeItem.$clearallnodes()
```

The evTreeExpandCollapseFinished event is sent to confirm that the evTreeExpand or evTreeCollapse message is finished. You can use this event to update other controls or states.

Changing a Node Name

The evTreeNodeNameFinishing event is sent to a tree list before the node name is updated with some new value entered by the user. It provides the parameters pNodeItem for the node, and a character variable pNewText containing the new text entered. pNodeItem.$name still holds the original text. This message is normally used to validate the new node name.

For example, you can use $findnodename to make sure the new name is unique:

```
On evTreeNodeNameFinishing
  If pNewText = ''
    OK Message {Name must contain a value}
    Quit event handler ( Discard Event )
  Else
    Do $cobj.$findnodename(pNodeItem,pNewText,kTrue,kTrue) Returns Found
    If not(isnull(Found)) & Found.$seedid<>pNodeItem.$seedid
      OK Message ('Name must be unique')
      Quit event handler ( Discard Event )
    End If
  End If
```

The evTreeNodeNameFinished event is sent to a tree list after the node name has been changed. It provides the parameters pNodeItem for the node and a character variable pNewText containing the new text entered.

Formatting Text in Tree Lists

You can use the style() function and the Text formatting constants (listed in the Catalog under Text escapes) to format text in several areas of Omnis including the node text in a tree list.

To use the style() function in a tree list, set the $styledtext property of the tree list to kTrue and add the style() function directly to the definition of the tree list. For example:

```
# Method '$makeTreeList'
# create local vars lLevel0, lLevel1, lLevel2 (Chars)
# lExpand, lIcon, lIdent, lTextcolor (Long Integers)
# lEnterable (Boolean)
# and instance var iList (List)
Do iList.$define(lLevel0,lLevel1,lLevel2,lIcon,lIdent,lEnterable,lExpand,lTextcolor)
Do iList.$add(con(style(kEscStyle,kBold),'Germany'),'','',0,100,0,2,0)
Do iList.$add('Germany',con(style(kEscStyle,kItalic),'Manufacturer'),'',0,101,0,2,0)
Do iList.$add('Germany','Manufacturer',con(style(kEscStyle,kUnderline),'Porsche'),0,102,0,2,0)
Do iList.$add('Germany','Manufacturer',con(style(kEscLTab,100),'Mercedes'),0,103,0,2,0)
Do iList.$add('Germany','Manufacturer',con(style (kEscColor,kBlue),'BMW'),0,103,0,2,0)
```

The following code can be added to the $construct() method of the window to initialise the list and set the tree list contents to the list data using the $setnodelist() method.

```
# $construct() method of window
Do method $makeTreeList
Do $cinst.$objs.treelist.$setnodelist(kFlatList,0,iList)
```

Video Player

Group	Icon	Name (type)	Description
Media	▶	Video Player (Xcomp)	Plays video file from disk or rem

The **Video Player** (AVPlayer) lets you load and play a video file from disk or on a remote server. It has the following properties:

Property	Description
$allowfullscreen	kTrue if the full screen option is available (macOS only at the moment)
$controlstyle	The type of controller shown on the player, a constant: kAVControlsFloating, kAVControlsInline, kAVControlsMinimal, kAVControlsNone
$controltime	The current play time of the player in seconds
$controlvolume	The current volume of the player - 0 to 10
$videoresize	The video resize mode during playback, a constant: kAVVideoResizeAspect, kAVVideoResizeAspectFill, kAVVideoResizeFill

The Video Player has the following methods:

Method	Description
$load	$load(cUrlOrLocalPath) will load the movie
$pause	$pause() will pause the movie
$play	$play() will play the movie

WAV Player

Group	Icon	Name (type)	Description
Media	∿	WAV Player (Xcomp)	Plays a WAV sound file

The **WAV Player** (or WavPlay) lets you load and play WAV sound files from disk. The component is available as a window component as well as a non-visual component. The same methods are available in the different types of component.

The WAV Player has the following methods:

Method	Description
$playwavfile	$playwavfile(cPathName) plays the wav file with pathname cPath
$stopwav	$stopwav() stops all wav files currently being played

Background Objects

Any graphic objects you place on the window background are considered to be *Background Objects*. They do not hold data, rather they are graphical devices for enhancing the appearance of the desktop UI.

Component Group	Component name
Labels	Label String Label (Xcomp) Text
Shapes	3D Rectangle Line Oval Rectangle Rounded Rectangle See also Shape Field

Background Images

You can paste an image into a window as a background object using the **Edit>>Paste** (Ctrl/Cmnd-P) or **Edit>>Paste from File** menu items. The former choice pastes a picture you have cut or copied to the clipboard; the latter pastes an image file directly onto the window background.

Under the Appearance tab, the $tile property lets you tile an image across the whole area of the object, otherwise the image is stretched when and if you resize the object. In addition, under the Action tab, you can enable the $cachepicture property to force Omnis to cache the image on the client machine which allows the window to be drawn quicker.

Background Object Names

Windows class Background objects, such as Label and Text objects, have the $name property which defaults to the ident of the object, that is, a number which is generated automatically. You can assign your own name which will help you identify the object more easily: you cannot set $name to a number, but the name can include a numeric character. Setting the name to empty resets it to its default number ident.

Using Non-TrueType fonts

The item 'backgroundObjectsMustUseTrueTypeFont' in the 'windows' section of config.json controls whether or not TrueType fonts are used for background objects. If true (the default), TrueType fonts must be used. When false, you can use non-TrueType fonts for background objects, but note that in some situations, e.g. in drag bitmaps, the text may not draw.

Corner Radius

The Rounded Rectangle background object (and Shape Field) have the **$cornerradius** property, which is the pixel radius of the corners of the object (assign a value <= 0 to set this property to its default value). The maximum value of $cornerradius is 255.

In versions prior to Studio 11, rounded rectangles were not drawn with the same amount of rounding on Windows and macOS: when the value of $cornerradius is 0, this difference is maintained, for compatibility, and after converting a library to Studio 11, $cornerradius defaults to zero. When you set the value of $cornerradius to a positive integer (1-255), both platforms use the same amount of rounding.

Label and Text Objects

The **Label** object lets you place text labels on a window, typically to label other data-bound fields or objects. The text for the label is contained in the $text property. The style of the text in the label object is controlled by the $fieldstyle property. You can set the style to None and assign your own font and style properties. You can set the color of the text in $textcolor.

The **Text** object is very similar to the label object. They have the same text properties under the Property Manager, but different usage: label objects do not support several features that are available for text objects. The main differences between label and text objects are as follows:

- Label objects *do not* wrap, and are therefore suitable for field labels or short single line text objects. Text objects do support multiple lines
- Label objects *do not* support the use of square bracket notation, whereas Text objects evaluate variables or calculations contained in [ ] placed in the text
- Label objects *do not* support rich text features within the text, whereas Text objects let you assign different character formatting (italic, bold, underline, and so on) to individual characters or words within a single text object

You can specify the $effect, $bordercolor and $linestyle properties to apply border style effects to both Label and Text objects.

Aligning Text

The $vertcentertext property allows you to control the vertical alignment of text for single- and multi-line Text objects. If set to kTrue, single-line text (or any text in a kText background object) is vertically centered in the height of the field.

When $vertcentertext is set to kFalse, and if you give the label the same height and $top property as the field, the text will draw on the same base-line on all platforms, provided the field and label have the same font, point size and style (when $vertcentertextis false, matches the behavior prior to Studio 10).

There is an entry on the Align menu, to align labels vertically. To align a number of labels and fields, select the objects and select the Center Text Vertically option. The new align option attempts to find labels for fields (objects that have the $vertcentertext property within the selection), and sets $vertcentertext to kTrue. The $top and $height of the label is also set to match the corresponding field. You can use Undo to reverse the alignment.

If you have used a Multi Line Entry Field with a label, the above will not work satisfactorily. To achieve the correct base-line drawing, use a label with $vertcentertext set to kFalse, and set its $top coordinate to the $top coordinate of the multi-line field plus the height of the top border of the multi-line field.

String Labels

The **String Label** object is specifically designed to allow you to create different language versions of your applications. The String Label gets its text content from a String Table depending on the setting of the client Locale, so can display the correct language for the current client automatically.

If you intend to provide localized versions of your application, you should use the String Label to create all the text labels in your windows, rather than using the Labels and Text objects that do not provide localization support. See the Localization chapter for more information about using the String Label and localizing your application.

Deprecated Components

Some external components have been deprecated and have been moved to the **Deprecated Components** group in the Component Store. They can be displayed using the **Exclude Group** option, available by right-clicking on the Component Store. ***They should not be used for new applications and are only included for backwards compatibility.***

The following External components (Xcomps) are available in the 'Deprecated' group in the Component Store; they are all external components and may need to be loaded to appear in the Component Store. See Loading External Components.

External Component
Color DropList Control
Fade Picture Control
File DropList Control
File List Control
Filter List Control
Formroll Control
Hot Picture Control
Html Object
Icon Array
Icon DropList Control
Line DropList Control
Pattern DropList Control
Picture List Control
Rtf Viewer
Slider Pal
Tab Bar Control
Tile
Timer Control (Milliseconds and Seconds)
Tool Pal
Tooltip
Tray Icon
Tree List
Wash
Zoom Control

Color DropList Control

This is a Deprecated component; it should not be used for new applications and is only included for backwards compatibility.

The **Color DropList** control is contained in the 'Cool DropList' package; so too are the Icon droplist, Line droplist, and Pattern droplist. They all behave and are setup in a similar manner using a two-column Omnis list variable to populate the droplist contents: the first column of the list should contain a color, icon_id, line, or pattern value, and the second column can contain a text description. For color values you can use the Omnis color constants; the icon_id is the ID from one of the Omnis icon data files or the #ICONS system table; the line or pattern is specified by a number in the range 1 to 16. For example the following method builds a list for a Color Droplist:

```
Do iColorlist.$define(ivColor,ivColortext)
Do iColorlist.$add(kRed,'Red')
Do iColorlist.$add(kBlue,'Blue')
Do iColorlist.$add(kGreen,'Green')
Do iColorlist.$add(rgb(178,178,0),'Olive')
Do iColorlist.$line.$assign(1)
```

When the user selects a line in such a droplist, an evClick event occurs. You can trap this event in the $event() method for the object and return the value of column 1 for the selected line in the list.

Fade Picture Control

This is a Deprecated component; it should not be used for new applications and is only included for backwards compatibility.

The **Fade Picture** (Fadepict) component displays image data and lets you apply a fade effect to an image as it is loaded, using one of over 30 different styles including slide up, slide down, wash left, wash right, spiral in, spiral out, and so on. For example, you could apply a fade effect from one image to the next, as you step through an image database, to produce a similar effect as a slideshow presentation.

The Fadepict component can display picture data retrieved from either a server database or Omnis data file; the latter supports the new true color shared picture mode. The component requires an instance variable (or row variable column) of Picture type specified in the $dataname property. The style of fade is stored in the $fadestyle property which you can set in design mode, under the Custom tab in the Property Manager, or at runtime using the notation. If set to kTrue, the $fadeondatachange property ensures a fade is triggered when the image data changes in the object. The $disolvesize property affects the size of the blocks in the fade when the fade style is one of the dissolve styles. Images are displayed same size unless $stretch is set to kTrue and in this case they are stretched to fit the field size; $borderh and $borderv let you add a border inside the field when $stretch is kTrue.

As well as the evBefore and evAfter field events, the Fadepict component reports the evFadeFinished event which you can detect in the $event() method for the object.

For example, you could use the Fadepict component to display images in a simple picture database, rather than using the regular picture field; when the image changes the new image is 'faded in' using one of the fade styles.

The $construct() method in the window opens the picture database and loads the first record; the image is stored in the carPict field in the fCars file class. Note that the images are stored in the library, which is loaded using sys(10). The method then assigns the data to the iPicture variable and the variable data is sent to the client.

```
# $construct() method of the window
Open data file (Do not close other dat) {[sys(10)],cardata}
Set main file {fCars}
Find first
Calculate iPicture as fCars.carPict
Do $cinst.$senddata(iPicture)
```

The window contains a Next and Previous so you can cycle through the database; if you reach the beginning or end of the database the last or first image is loaded to give the appearance of a continuous stream of images.

```
# $event() method for the Next button
On evClick
  Next ## or Previous
  If flag false
    Find first    ## or Find Last for Previous button
  End If
  Calculate iPicture as fCars.carPict
  Do $cinst.$senddata(iPicture)
  Do $cinst.$objs.formfade.$redraw()
```

As an extra refinement you could add two lines of code to the Next and Previous methods to fade the image using one of the 35 fade styles picked at random using the randintrng() function. Note that you can set the $fadestyle using one of the style constants, such as kFadeBounce, or using their numeric equivalent; the group of fade style constant are numbered 1 to 35 in the order they are listed in the Property Manager. The method for the Previous button would be:

```
# $event() for the Previous button
# window contains iFadeStyle of Number type
On evClick
  Previous
  If flag false
    Find last
  End If
  Calculate iFadeStyle as randintrng(1,35)
  Do $cinst.$objs.formfade.$fadestyle.$assign(iFadeStyle)
  Calculate iPicture as fCars.carPict
  Do $cinst.$senddata(iPicture)
  Do $cinst.$objs.formfade.$redraw()
```

File DropList and File List Controls

These are Deprecated components; they should not be used for new applications and are only included for backwards compatibility.

The **File DropList** and **File List** controls are part of the FileList package and behave in the same way. Both controls can display the contents of a folder, either in a standard list format or as a droplist.

The Filelist control displays files, directories, or volumes in a standard list or droplist, based on the contents of an Omnis list variable which is assigned in the $dataname property of the component.

- $showdirectories, $showfiles, $showvolumes
  setting one of these properties to kTrue specifies what will be displayed in the list, either directories (folders), files, or volumes
- $treeview
  if set to kTrue the list is displayed as a tree list
- $filefilters
  specifies the file extensions which will be included in the list, e.g. '*.dll', while '*.*' will include all file types; multiple extensions are specified as a semicolon separated list; the Mac creator and type can be specified in the format '<Creator>|','<Type>|')
- $hidebundlecontent
  If true, then bundles are treated as files instead of folders under macOS
- $path
  is the path of the directory which will be displayed and loaded in the list; this defaults to the C:\ drive or Macintosh hard disk

The $refresh() method is used to populate or update the list. When the user selects a list line, you can return the path to the selected file.

- $fullpath()
  $fullpath(iLineno) returns the full path of the specified line number
- $refresh()
  $refresh([bClearlist]) builds or refreshes the list by re-reading the folder/volume; the $redraw() method will redraw the list field

Filter List Control

This is a Deprecated component; it should not be used for new applications and is only included for backwards compatibility.

The **Filter List** (Filterlist) control is similar to a standard list field except that you are able to change the color and height of the lines in the list and the color of the selected line. The Filter list has the following methods.

- $::hiliteline
  $hiliteline(iLinenum,bHilite,cTextline) highlights or un-hilites the specified line iLinenum according Boolean bHilite; plus the text in the hilited line can be replaced with the text in cTextline parameter
- $scrolllist
  $scrolllist(bVert,iType) scrolls the list vertically or horizontally depending on bVert; the iType can be value 0-8 and determines how the list is scrolled

Formroll Control

This is a Deprecated component; it should not be used for new applications and is only included for backwards compatibility.

The **FormRoll** component is a graphical pushbutton that highlights when the user passes their mouse over the button. The component has a number of properties to control the look and behavior of the object when the mouse is placed over it. You can specify the image ($outsideimage and $insideimage) and text ($outsidetext and $insidetext) to be displayed when the mouse is either inside (over) or outside (not over) the button. You can also specify the text offset using the $textx and $texty properties, and set the spacing of multi-line text using $betweenlines.

The FormRoll components reports the evIsInside and evClick events, which have to be enabled in the $events property for the object.

Hot Picture Control

This is a Deprecated component; it should not be used for new applications and is only included for backwards compatibility.

The **Hot Picture** (Hotpict) control lets you create a 'hot' or clickable area in a window, usually over a photo, map, or a graphic. For example, you could display a map of the US with the various states defined as different hot areas.

The Hotpict component appears on your window as a single rectangle placed over your whole image with separate hot areas defined within the rectangle. In design mode, you can use various mouse/key combinations to add new hot areas, add nodes to the current area, or move existing areas, as follows. Note that right click on a Mac means Ctrl-click.

Action	Mouse/key press
Add a hot area	Shift-Right click in the Hotpict field; new hot area is added to the top-left of the Hotpict component, you can move the new area, add nodes and reshape area as below.
Move area within Hotpict	Right click inside the area and drag
Select an area	Right click inside the area; the Property Manager displays the properties for the selected area.
Delete hot area	Ctrl/Cmnd-Right click on the area
Add node to an area	Shift-Right click on a node; new node is added next to existing node
Move node to reshape area	Right click on node and drag
Delete node	Ctrl/Cmnd-Right click on node

When you Right-click inside a hot area within the Hotpict field rectangle, the Property Manager shows the properties for the selected area. The $currentid property specifies the id for the current hot area. As you add hot areas to the field in design mode, consecutive numbers are assigned to each hot area and in most cases you can use these numbers to identify an area; you can however change the ids or add your own default value for each area. Alternatively, you can assign a name to each hot area in the $currentname property. At runtime, the $arealist property contains a list of areas in the Hotpict field.

You can assign a cursor to each area in the $currentcursor property; the cursor is displayed when the user's mouse enters the area. You can also highlight an area in several ways: $invertonenter inverts the area when the mouse enters the area; $frameonenter hilights the area by displaying a frame; and $flashonclick inverts when the user clicks the area.

The Hotpict component reports the evAreaClicked event which you can detect in the $event() method for the Hotpict component. The event passes the pAreaid and pAreaname parameters containing the id and name of the selected hot area. In the above example, pAreaname could be used to pass the name of the state selected, assuming 'Texas' has been added to $currentname for that area.

Html Object

This is a Deprecated component; it should not be used for new applications and is only included for backwards compatibility.

The **Html Object** control lets you display HTML files or documents in a window class; it is used in the Omnis environment in the Help system. (Note you can use the OBrowser control to display a full web page, with support for JavaScript and other browser functionality.)

The HTML Object does not have a $dataname property, rather you specify the path to an HTML text file in the $filename property. You can set the filename in design mode or set it dynamically at runtime.

The HTML component has the following properties and methods:

- **$filename**
  pathname to the HTML to be displayed by the component
- **$fontsizeadj** (runtime only)
  increases or decreases the size of the text in the current HTML file, in a range from -3 to 3 with the default being 0 (zero)
- **$eventhwnd** (runtime only)
  returns the hwnd reference of the HTML component when an event is triggered
- **$searchwords** (runtime only)
  contains a list of words that will be highlighted in the current HTML file; the words should be separated by spaces; the specified words and the background page surrounding the words are shown in their inverse

- **$startanimatescroll()**
  $startanimatescroll(horzscrollunits,vertscrollunits,interval) scrolls the HTML component using the settings in horzscrollunits, vertscrollunits, interval
- **$stopanimatescroll()**
  stops scrolling the current HTML component
- **$pathtoapi**(path)
  converts an Html file name and path to a disk name and path; uses the correct separators for the correct operating system; performs the reverse of $pathtohtml()
- **$pathtohtml**(path)
  converts a standard disk name and path to a full Html file name and path; performs the reverse of $pathtoapi()
- **$getselectedtext**(text)
  returns any text currently selected in the HTML component

The Html Object reports the following events:

- **evAnimateScrollEnd**
  sent when the HTML component is finished scrolling; returns the parameters pEventCode, pCtrlIdent
- **evEventTag**
  sent when an embedded custom Html tag is read; returns the parameters pEventCode, pCtrlIdent, pName, pValue
- **evExecTag**
  sent when an embedded custom Html tag is executed; returns the parameters pEventCode, pCtrlIdent, pTagName, pTagValues
- **evHyperlink**
  sent when a hyperlink is clicked; returns the parameters pEventCode, pCtrlIdent, pHRef, pName, pTarget, pTitle
- **evImagePluginCreate**
  sent when an embedded image plugin is invoked, such as an embedded Jpeg object; returns the parameters pEventCode, pCtrlIdent, pType, pProperties, pWindowRef, pWidthRef, pHeightRef
- **evPluginDestroy**
  sent when an embedded plugin is destroyed; returns the parameters pEventCode, pCtrlIdent
- **evSetTitle**
  sent when the Html document is read; returns the parameters pEventCode, pCtrlIdent, pTitle
- **evXCompPluginCreate**
  sent when an embedded external component is invoked; returns the parameters pEventCode, pCtrlIdent, pComponentLib, pComponentCtrl, pProperties, pWindowRef, pWidthRef, pHeightRef

```
On evSetTitle
  Do $cwind.$title.$assign(pTitle)
On evHyperlink
  Do $cobj.$filename.$assign(pHRef)
On evHyperlink
  If pos(".lbs",pHRef)
    Calculate lOmnisLibPath as pHRef
    Do $cobj.$pathtoapi(lOmnisLibPath)
    Open library (Do not close others) {[lOmnisLibPath]}
  Else
    Do $cobj.$filename.$assign(pHRef)
End If
```

Supported HTML tags

The following HTML tags and their attributes are supported in the Omnis Document Viewer. Tags and attributes not listed here are *not* supported and ignored by the Document Viewer.

```
<!-- Comment -->
<a href name target title>…</a> clicks generate evHyperlink event. All attributes are passed to Omnis metho
<address>…</address>
<b>…</b>
<base href>
<basefont size>
<bgsound … > tag and all attributes are sent to evExecTag event
<big>…</big>
<blockquote>…</blockquote>
<body bgcolor text link vlink alink leftmargin topmargin>…</body>
<br>
<center>…</center>
<cite>…</cite>
<code>…</code>
<comment>…</comment>
<dd>…</dd>
<dir compact>…</dir>
<div align>…</div>
<dl compact>…</dl>
<dt>…</dt>
<em>…</em>
<font face size color>…</font>
<h1>…</h1> to <h6>…</h6>
<head>…</head>
<hr align noshade size width>
<html>…</html>
<i>…</i>
<img src height width align alt border hspace vspace> generates an evImagePluginCreate which is sent to the
<kbd>…</kbd>
<li type start>…</li>
<listing>…</listing>
<marquee align bgcolor height hspace scrollamount scrolldelay vspace width>…</marquee>
<menu compact>…</menu>
<nobr>…</nobr>
<ol compact start type>…</ol>
<p align>…</p>
<pre align>…</pre>
<s>…</s>
<samp>…</samp>
<small>…</small>
<strike>…</strike>
<strong>…</strong>
<sub>…</sub>
<sup>…</sup>
<table align bgcolor border bordercolor bordercolordark bordercolorlight cellpadding cellspacing hspace val
<td align bgcolor border bordercolor bordercolordark bordercolorlight colspan nowrap rowspan valign width>…
<th align bgcolor border bordercolor bordercolordark bordercolorlight colspan nowrap rowspan valign width >
<title>…</title> generates an evSetTitle event
<tr align bgcolor border bordercolor bordercolordark bordercolorlight valign>…</tr>
<tt>…</tt>
<u>…</u>
<ul compact type>…</ul>
<xmp>…</xmp>
```

Custom HTML tags

This section describes all the extended or custom tags supported in the Document Viewer. These tags are not standard HTML and are ignored by other browsers.

EVENT tag

The EVENT tag sends an evEventTag message to the $event() method of the Document Viewer object. Two parameters pName and pValue are sent. The values of these two parameters can be specified in HTML. For example:

```
<event name="your event name" value="your event value">
```

Both name and value can be preceeded by the keywords apipath or htmlpath. This tells the parser to convert the value to a full qualified api or html path based on the current document's location. For example, if the document path is "file:///c/docs/thedoc.htm", the following html:

```
<event name="openlib" apipath value="samplelib.lbs">
```

sends the following to your event method on the Windows platform:

```
pName = "openlib"
pValue = "c:\docs\samplelib.lbs"
```

XCOMP tag

The XCOMP tag allows you to embed Omnis built-in and external components within your HTML document. For example, you can embed a standard Omnis pushbutton or any external component like the Marquee control. When the parser encounters the XCOMP tag, an evXCompPluginCreate event is send to the $event() method of the Document Viewer object. When you create an HTML viewer control from the Component Store, it contains code to handle this event.

To embed external components you need to specify the library name for *componentlib* and the control name for *componentctrl*.

To embed built-in controls you must specify "internal" for the *componentlib* and the control constant, e.g. kPushbutton, for *componentctrl*.

You must also specify a width and height. Following this you can specify Omnis property names (exclude the $ symbol) with their values to setup the various properties of the control. For example, the following code references the Marquee external component:

```
<xcomp componentlib="Marquee Library" componentctrl="Marquee Control" width="300" height="20" align="middle
```

The following example, shows a built-in component with event method:

```
<xcomp componentlib="internal" componentctrl="kPushbutton" width="180" height="30" text="Click Me" name="Te
  <script language="xcomp" target="">
    ref.$methods.$add("$event")
    ref.$methods.//$event//.$methodtext.$assign("On evClick<br>OK message Button (Icon,Sound bell) {You cli
  </script>
</xcomp>
```

Any of the property names, if appropriate, can be preceded by the keywords apipath or htmlpath. This tells the parser to convert the value of the property to a full qualified api or html path based on the current document's location. See EVENT tag for an example.

In addition, you can specify horizontal and vertical spacing using the html attributes *hspace* and *vspace*.

When the current document is closed, the viewer generates an evPluginDestroy message to destroy the components. When you create a html viewer control from the component store, it already contains code to deal with this event.

Standard tags

The following standard HTML tags are interpreted by the Document Viewer parser in a special way.

BGSOUND tag

The bgsound tag is a standard html tag, but the viewer cannot play sounds. When the parser encounters this tag, an evExecTag is sent to the $event() method of the Document Viewer object. The pTagName parameter is "BGSOUND" and the pTagValues parameter is a two column list specifying the tag's parameter names in the first column and their values in the second column.

TITLE tag

The title tag is a standard html tag, and when the parser encounters this tag an evSetTitle event is generated. The pTitle parameter contains the title text which, for example, allows you to assign the document's title text to the current Omnis window title.

IMG tag

The img tag is a standard html tag, but the viewer itself can't draw images. It generates the evImagePluginCreate message asking you to create the appropriate image control. The pType parameter is one of the following "JPEG" or "GIF". When you create a html viewer control from the Component Store, it contains code to handle this event.

When the current document is closed, the viewer generates an evPluginDestroy message to destroy the component. When you create a Document Viewer control from the Component Store, it contains code to handle this event.

Icon DropList Control

See Color DropList Control.

Line DropList Control

See Color DropList Control.

Pattern DropList Control

See Color DropList Control.

Picture List Control

This is a Deprecated component; it should not be used for new applications and is only included for backwards compatibility.

The **Picture List** (Piclist) control can display a list of icons from an Omnis icon data file (Omnispic or Userpic) or the #ICONS system table in the current library. The icon IDs are specified in a list variable and assigned to the $dataname property of the piclist. You can specify the column in the list which contains the icon IDs using the $piccolumn property, which is set to first column (value=1) by default. The second column could contain another value associated with the icon which can be loaded when the user clicks on the list.

Rtf Viewer

This is a Deprecated component; it should not be used for new applications and is only included for backwards compatibility.

The **RTF Viewer** lets you display an RTF word processing file in a window. The RTF file is specified in the $filename as a pathname. The component has the following methods:

- $getselectedtext()
  $getselectedtext(&cText) returns the currently selected text in cText
- $pathtoapi()
  $pathtoapi(&cPath) converts cPath from the HTML syntax to the syntax for the current platform, and sets cPath to the result
- $pathtohtml()
  $pathtohtml(&cPath) converts cPath from the syntax for the current platform to the HTML syntax, and sets cPath to the result
- $printdocument()
  prints the document (no parameters)
- $startanimatescroll()
  $startanimatescroll(iHorzscrollunits,iVertscrollunits,iInterval) starts scrolling the document once every iInterval (milliseconds) by the specified units
- $stopanimatescroll()
  stops automatic scrolling of the document (no parameters)

Tab Bar Control

This is a Deprecated component; it should not be used for new applications and is only included for backwards compatibility.

The **Tab Bar** control (TabBar) displays a number of thumb tabs on which the user can click; in this respect, it is similar to the tab strip field. The tab bar is typically used with the Paged pane component to present a number of layers or pages in a window. For example, a preferences or options window could have separate pages for different options accessed by clicking the tabs in a tab bar.

The tab bar reports the evClick event and passes the pLineNumber parameter containing the number of the tab clicked on. The following method is the $event() method for a tab bar on a window containing a paged pane. The method detects the evClick on the tab bar and sets the currentpage of the paged pane.

```
On evClick
  Do $cinst.$objs.pagepane.$currentpage.$assign(pLineNumber)

# pLineNumber contains the number of the tab clicked
```

The $position property specifies the orientation of the tabs, either the top, bottom, left, or right. The left and right settings rotate the tabs and text labels and allow you to position the tab bar down the side of your window or paged pane area. The text on the tabs must use a True Type font if you wish to rotate the tab bar. Other significant properties include:

- **$nosoftab**
  The number of tabs.
- **$::currenttab**
  The number of the current tab.
- **$::selectedtabcolor**
  The color of the current tab, when enabled.
- **$::style**
  The style of the tabbar, either kDefaultWebTab, kSquareWebTab, or kTriangleWebTab
- **$tabenabled**
  If true, the tab with the current tab number is enabled; this can be set during $construct of the window to enable or disable specific tabs.
- **$tabtext**
  The text of the current tab.
- **$tabtip**
  The tooltip text of the current tab.

For example, a window could use a tab bar and paged pane to organize the different options available to customers, such as Accounts, Transactions, etc. The $event() method for the tab bar would detect the click and pass the number of the tab in the pLineNumber parameter. The following method first tests whether or not the user is logged on and if not switches to the logon page. Next, the method uses a Switch statement to branch according to the tab number passed in the pLineNumber parameter.

```
On evClick
  Do $cinst.$senddata(#NULL)
  If not(iLoggedOn)
    Do $cinst.$showmessage("Sorry, you must logon first","Error")
    Do $cinst.$objs.tab.$::currenttab.$assign(1)
    Quit method
  End If
  Switch pLineNumber
  Case 1
    Calculate iLoggedOn as kFalse
    Calculate iUserName as ""
    Calculate iPassword as ""
    Do $cinst.$objs.page.$currentpage.$assign(2)
    Do $cinst.$senddata(iUserName,iPassword)
    Do $cinst.$redraw()
  Case 2
    Do $cinst.$objs.page.$currentpage.$assign(3)
  Case 3
    Do method $showoneaccount
  Case 4
    Do $cinst.$objs.page.$currentpage.$assign(5)
  Case 5
    Do $cinst.$objs.page.$currentpage.$assign(6)
  Case 6
    Do $cinst.$objs.page.$currentpage.$assign(7)
  Case 7
    Do $cinst.$objs.page.$currentpage.$assign(8)
```

Timer Control

This is a Deprecated component; it should not be used for new applications and is only included for backwards compatibility.

The **Timer Control** provides a Second (S) or Millisecond (MS) timer. This is useful if you want to trigger a method or some other action after a specified period. When the time expires an **evTimer** event is sent to the component so its $event method can determine the action taken. The Second based timer control supports an optional visual countdown using $showcountdown plus extra font and style properties.

The Timer control has the following properties:

Property	Description
$autoreset	The timer will reset its parameters after returning from a evTimer
$reentrant	If true, the timer event can be generated while the previous timer event is being processed
$timeleft	The amount of time left until the timer will expire (if the timer is a millisecond timer then the value is an integer, otherwise it is a date value containing the time left)
$timervalue	The duration of the timer
$useseconds	If true, $timervalue is a value in seconds; otherwise it is a value in milliseconds

The Timer control has the following methods:

Method	Description
$resettimer	Stops and then restarts the timer using the current value of $timervalue
$starttimer	Starts the timer using the current value of $timervalue
$stoptimer	Stops the timer

Timer Object

The Timer control is also available as an External Object and Worker object allowing you to create an object variable that receives timer events. The Timer Object has the following properties:

Property	Description
$autoreset	The timer will reset its parameters after returning from a evTimer
$queueevent	If true, the event will be queued plus *the timer will need to be reset manually* regardless of $autoreset setting
$reentrant	If true,the timer event can be generated while the previous timer event is being processed
$timeleft	The amount of time left until the timer will expire (if the timer is a millisecond timer then the value is an integer, otherwise it is a date value containing the time left)
$timervalue	The duration of the timer
$useseconds	If true, $timervalue is a value in seconds; otherwise it is a value in milliseconds

The Timer Object has the following methods:

Method	Description
$resettimer	Stops, and then restarts the timer, using the current value of $timervalue
$starttimer	Starts the timer, using the current value of $timervalue
$stoptimer	Stops the timer
$timer	Method called when the timer has expired

Tray Icon

This is a Deprecated component; it should not be used for new applications and is only included for backwards compatibility.

The **Tray Icon** control is a non-visual external object which lets you add an icon to the Windows icon tray at the bottom of the screen (the control is not available for other platforms).

The mouse coordinates passed to the menu event are relative to the main Omnis Window, and these work with the PopupMenu command, for example. When Omnis is minimized, these coordinates will be larger than expected (due to how Windows 10 handles them), but they can still be used with PopupMenu to open the menu in the correct position.

Zoom Control

This is a Deprecated component; it should not be used for new applications and is only included for backwards compatibility.

The **Zoom** control lets you enlarge areas of the screen. The control supports dynamic updating via its own timers and various levels of enlargement.

Chapter 12—Window Programming

Window programming, Window classes and Window components are required for developing desktop or thick client applications only, and are therefore hidden in some editions of Omnis Studio, including the Community Edition. To create web or mobile apps, you need to create Remote forms using JavaScript component.

This chapter describes some of the more advanced properties of window classes, and the different programming techniques you can use in window classes, including Window methods, Field styles, Format strings, Input masks, Drag and drop, and creating your own HTML components for window classes.

Window Design Task

When testing a window class (using Ctrl/Cmnd+T, or the Test button in the Studio Browser, or via the context menu for a window class) Omnis switches to an instance of the design task, or the startup task if there is none.

There is an item in the config.json, tryDesignTaskWhenTestingWindow in the “ide” section, to control this behavior. When true (the default), Omnis looks at the design task name, and if it is the same as the startup task name, switches to the startup task, as in previous versions. If however, the design task name is different, Omnis switches to the first instance it can find of that task, but if there is none, it switches to the startup task as in previous versions.

When tryDesignTaskWhenTestingWindow is false, the behavior is the same as for previous versions: when testing a window, Omnis switches to the startup task of the library containing the window.

Window Methods

The following methods can be called within a window instance (open window) either via $cwind, $cinst or $iwindows.window-instance-name, in addition to the standard methods $canclose, $close, and $redraw.

Method	Description
$bringtofront()	$bringtofront([bRestoreIfMaximized=kFalse]) brings the window instance to the front. If bRestore the window to its previous size and position if it is currently maximized e.g. Do $iwindows.wTest.$
$beginanimations()	$beginanimations(iDuration [,iCurve=kAnimationCurveEaseInOut]) after calling this, assignments for iDuration milliseconds by $commitanimations() See Object Animation
$commitanimations()	$commitanimations() animates the relevant property changes that have occurred after the matc
$maximize()	$maximize() maximizes the window instance
$minimize()	$minimize() minimizes the window instance
$sendevent()	$sendevent(iEvent [,eventParameters...]) sends event iEvent (an ev... constant value) to the object name,value pairs, for example $sendevent(evClick,‘pLineNumber’,2). Returns kFalse if the event is Methods
$showmessage()	$showmessage(cMessage [,cTitle, iOptions=kMsgOK]) displays a message using the specified cMe of kMsg... constants).Returns true for OK or Yes,false for No or cancel.Use msgcancelled() to check
$showtoast()	$showtoast(cMessage,[cTitle, iStyle=kToastSuccess, iStack=kToastStackTopRight, iTimer=4000, bC notification using the title and message using the style See Toast Messages

Window Animations

Certain properties of window instances can be animated using the $beginanimations() and $commitanimations() methods, including $alpha, $left, $top, $width, and $height. So for example, when closing a window you could fade it out by setting $alpha under animation before closing it, or similarly you could enlarge a window under animation as you open it to create more impact (window animations work in a similar way to objects animations, which are described here: Object Animation).

Field Styles

If you are going to deploy your application on more than one platform, the objects in your application should use suitable text and font properties for each platform. You can achieve this using *Field Styles*. A field style is a style definition, similar to an HTML or document style, that you can apply to a window, remote form or report object. You can specify different appearance properties for all the supported platforms in Omnis, including Windows and macOS. Each field style contains:

- A set of Boolean flags, each of which identifies whether or not a particular standard property belongs to the style. The standard properties supported are: $align, $backcolor, $backgroundtheme, $backpattern, $bordercolor, $effect, $fontname, $fontsize, $fontstyle, $forecolor, and $textcolor.

- Custom properties. These are properties that are not in the set of possible standard properties. If the style contains custom properties, it can only be applied to a single type of object.
- For each platform, a set of property values. Each set contains a value for each standard property belonging to the style, and a value for each custom property.

Field Style Properties

The field style for an object is stored in its $fieldstyle appearance property; note that not all objects support the $fieldstyle property. Window objects also have the $fieldstylefocused property to allow you to style an object when it gets the focus, allowing you to override certain properties when a field has the focus such as the font style or color.

Creating Field Styles

You can create as many field styles as you like. They are stored in the #STYLES system table in the System Tables folder in a library. Having set up the styles in the style system table, you can copy the table to any library and use its styles throughout your whole application. When you copy an object from one library to another, its field style is also copied automatically, but only if there is not already one with the same name in the destination library.

To view the field styles system table

- Use the Class Filter (press F7/Cmnd-7 while the Studio Browser is on top) to make sure the system tables are visible in the Studio Browser
- Double-click on the **System Tables** folder
- Double-click on #STYLES

The #STYLES system table contains a number of styles that are applied to standard fields in Omnis by default. The icon in the left-hand column of the Styles grid indicates the control to which the custom properties in the style belong.

Printing styles

You can print a list of styles in the #STYLES system class by right-clicking on the class and selecting the Print Class option. The report shows a complete list of properties for each style in the #STYLES table.

Defining Field Styles

The Styles dialog lists all the styles in the current library. To define a style, you first select the standard properties that belong to the style. Then you set the values of these properties, for each platform.

To define a new style

- Click in the first empty line in the table and enter a name and description for the style
- With "All platforms" selected in the right-hand list, open the Property Manager, or bring it to the top (press F6/Cmnd-6)
- Set the $**has…** properties to kTrue, for each standard property to be included in the style. Note that the $**hascustom** property is only relevant when you are using custom styles, and it cannot be modified; custom styles are discussed below.

For example, if you want a style to set the font name and size only, enable only the $**hasfontname** and $**hasfontsize** properties.

- Go back to the Styles dialog and select a platform from the list on the right
- Bring the Property Manager to the top and enter a value for each property belonging to the style, e.g. set $fontname, $fontsize, and so on; the properties are located under the Text and Appearance tabs in the Property Manager
- Repeat this process for each platform
- When you have finished, click on OK in the Styles dialog

You can change an existing style by clicking on its name in the Styles dialog, selecting a platform, and editing its properties in the Property Manager.

Custom Styles

You can customize a style by dropping additional properties into the style.

To add a property to a style

- Open the Styles dialog, by double clicking on #STYLES in the Studio Browser
- Select a control on a design window
- Drag a property from the Property Manager, and drop it on to a style name in the Styles dialog

Alternatively, from the Property Manager:

- You can right-click on a property name and select the "Add To Style As Custom Property…" option and select the style name from the popup, after which the #STYLE system table can be opened

Note that multi-value properties, such as the column properties in a data grid, cannot be assigned to custom styles. In this case, the "Add To Style As Custom Property…" option will be disabled, and attempting to drag to the #STYLES window will fail.

When you add a property to a style, as above, this will set the **$hascustom** property of the style to kTrue. You can use the Property Manager to assign a value to the property, for each platform, in the same way that you assign values to the standard style properties.

Once you have added a property to a style, the style is linked to that type of control. You can no longer assign this style to controls of a different type.

You can remove the custom properties from a style by Right-clicking on the style in the Styles dialog, and selecting "Remove Custom Properties".

Listing custom styles

The $customlist() method returns a single column list containing the names of the custom properties of the style for any given platform. For example, the following method returns a list of custom properties for the style called Custom under Windows:

```
Do $clib.$fieldstyles.Custom.$platforms.kMSWindows.$customlist() Returns myList
```

Applying a Style to an object

The field style for an object is stored as a property. You can set this under the Appearance tab in the Property Manager.

- Open your window, remote form or report class
- Click on an object and view its properties in the Property Manager
- On the Appearance tab set the $fieldstyle property to the required style; the current style (if any) is selected in the dialog by default

The properties that you have enabled using the **has…** properties, and the custom properties if present, will override the properties in the object. For example, if you have set $hasfontname and $hasfontsize in your style definition, these properties will apply to an object with that $fieldstyle, whereas the other text properties will remain unaffected. Once you apply a field style to an object you can no longer set the properties belonging to the style; these become disabled in the Property Manager.

The library preference $styleplatform controls which set of platform characteristics defined in the style is used on the current machine. This defaults to the current platform.

Field Styles Notation

The Omnis notation allows you to manipulate field styles. The notation allows you to:

- Assign custom style properties
- Query custom style properties
- Add custom style properties
- Remove custom style properties
- Assign all properties of a style
- Assign all properties for a platform

Assigning custom style properties

To assign custom style properties you use the notation:

```
omnis Do $clib.$fieldstyles.StyleName.$platforms.kMSWindows.$thepropertyname.$assign(TheValue)
```

Querying custom style properties

To query custom style properties you use the notation:

```
Calculate TheValue as $clib.$fieldstyles.StyleName.$thepropertyname
```

Adding custom style properties

To add custom properties to a style you use the $add() method passing in a reference to the property of a window object or remote form object.

```
Set reference objectProperty to $clib.$windows.WindowName.$objs.ObjectName.$thepropertyname.$ref
Do $clib.$fieldstyles.StyleName.$add(objectProperty)
```

Note: The style will be linked to the object type, and additional custom properties can only be added if they belong to the same object type.

Removing custom style properties

You cannot remove individual custom properties from a style. You have to remove all custom properties at one time. To remove custom properties from a style you calculate the $hascustom notation to kFalse.

```
Do $clib.$fieldstyles.StyleName.$hascustom.$assign(kFalse)
```

Assigning all properties of a style

You can assign all properties of a style from another style in a single statement. You do this by setting up two item references to the two styles, and calculating one style from another.

```
Set reference srcStyle to $clib.$fieldstyles.TheSrcStyleName
Set reference dstStyle to $clib.$fieldstyles.TheDstStyleName
Calculate dstStyle as srcStyle
```

Assigning all properties for a platform

You can assign the properties of a style for a single platform from another style, or from a different platform from the same style. To do this you set up two item references referencing the platforms of the styles, and calculate one from the other.

```
Set reference srcStyle to $clib.$fieldstyles.StyleName.$platforms.kMacintosh
Set reference dstStyle to $clib.$fieldstyles.StyleName.$platforms.kMacOSX
Calculate dstStyle as srcStyle
```

Copying styles

The $copystyle() method enables you to copy styles between libraries (or the same library). $copystyle(rItem[,cNewName]) copies the style to rItem, which is an item reference to the library, and renames the style if cNewName is supplied. The method returns kTrue if successful. For example:

```
omnis Do $clib.$fieldstyles.TestStyle.$copystyle($libs.TargetLib,"myNewStyle")
```

copies the TestStyle from the current library to TargetLib and renames it to myNewStyle. If the new name is not supplied, the original name is maintained. If the style already existed under that name, it will be overwritten.

Using the style() function

In addition to using field styles for styling fields or text in your windows, you can use the *style()* function which can insert one or more style characters or "text escape" characters (represented by an Omnis constant) into a text calculation. Note you can also use text escapes and the style() function in text and fields in reports. You can use the style() function in the calculation of the following window fields or objects:

- Text object and Text type of Shape field (in the text itself, must be enclosed in square brackets)
- Headed List Box
- Combo box (list portion only)

Plus the following window components when their $styledtext property is enabled:

- List Box
- Dropdown List
- Tree List (style() can be used in the list definition)

The syntax of the function is:

```
style(style-character[,value]) `
```

Valid constants for the *style-character* are listed under 'Text escapes' in the Constants pane of the Catalog (press F9, click on Constants, then 'Text escapes'). The following constants are available:

*style-character*text escape	Description and possible *value*
kEscAdjustPos	Changes the position of the text using x.y *value*
kEscAngle	Rotates the text using the *value* kAngle0, kAngle90, or kAngle270
kEscBmp	Inserts an image specified by icon ID in *value* (you can include icon size constant, e.g. 1756+k16x16)
kEscColor	Changes color of text specified by rgb() *value* or a color constant under the Colors group in the Catalo
kEscCTab	Inserts a Center tab at the specified pixel position in *value*
kEscLTab	Inserts a Left tab at the specified pixel position in *value*
kEscRTab	Inserts a Right tab at the specified pixel position in *value*
kEscStyle	Changes the style of the text using a font style constant or sum of constants

Depending on the style character, you may also need to specify a *value*, which can be a numeric value or a constant, such a color constant. In its simplest form you can add the style() function to the text property of a text label, for example, the following text added to a text label will display an 'info' icon in front of the text (note the use of square brackets to evaluate the function):

```
[style(kEscBmp,1010)] Note:
```

You can use this function to format the columns in a headed list box field (added to the list definition for the headed list), for example, you could insert an icon by specifying its Icon ID, change the text color, or italicize the text in a particular column. The following example formats the columns in a headed list box by giving lCol1 a blue spot icon, lCol2 is red and lCol3 is italic:

```
Do con(lCol1,style(kEscBmp,1756),kTab,lCol2,style(kEscColor,kRed),kTab,lCol3,style(kEscStyle,kItalic))
```

Note that the alignment escapes (kEscCTab, kEscLTab, and kEscRTab) are not designed to work with the headed list box. The headed list box method $setcolumnalign() can be used to set the alignment of a column. The constants kEscAngle, and kEscAdjustPos cannot be used in lists or grids.

Format Strings and Input Masks

A *format string* is a set of characters or symbols that formats the data in a field for display, regardless of how the data is stored. The string is stored in the **$formatstring** property for the field. An *input mask* formats data as you enter it into a field, and is stored in the **$inputmask** property. On a window, only the **Masked Entry Field** allows a formatting string or input mask. When a user enters data into a field controlled by an input mask, Omnis rejects any text that does not conform to the format you've specified in the mask. Report data fields also support format strings.

To enter a format string for a field, you need to specify the type of data represented in the field, that is, its **$formatmode**: this property can be Character, Number, Date, or Boolean. You can enter a format string manually or use one from the dropdown list in the format string dialog: the default formats in this dropdown are stored in the appropriate system table.

Character Format Strings

To format a text field, you have to set its $formatmode property to Character. Character format strings have one or two sections. The first section contains the value display format; the second section contains the format to display for NULL or empty values. When you click on the $formatstring property, a dialog appears that lets you select a format. The dialog is the same for all the different formats.

You can enter a format directly into the Text Display Format field, either from the keyboard or using the buttons in the dialog. Alternatively, you can click on the down arrow in the Text Display Format field and select a format from the #TFORMS system table. The character formatting strings for the current library are stored in #TFORMS. You can edit #TFORMS by double-clicking on it in the Browser.

You can use the following symbols in character formats:

Symbol	Description
@	represents a single character or space
&	represents a single character but not a space
U	forces all characters in the field to upper case
L	forces all characters in the field to lower case
<	fills placeholders from left to right for left adjustment of the field; must be leading characters in field
X	truncates the value if it exceeds the format length. It truncates the front of the string; use the sequence <X to trunca
P	character fill; Px fills the front of the string with the character x to make the string the required length
;	section separator

Example character format strings

Format string	ANT	adder	Antelope	NULL or Empty
@	ANT	adder	Antelope	
U	ANT	ADDER	ANTELOPE	
L’Text: ‘& | Text: ant | Text: adder | Text: antelope | | | Px&&&&&&&& | xxxxxANT | xxxadder | Antelope | | | <Px&&&&&&&& | ANTxxxxx | adderxxx | Antelope | | | X&&&& | ANT | dder | lope | | | <X&&&& | ANT | adde | Ante | | | &;’Null text value’	ANT	adder	Antelope	Null text value

Number Format Strings

To format a numeric field you have to set its $formatmode property to Number. Number formats can use the following symbols in a format string.

Symbol	Description
0	zero; displays a digit; displays leading or trailing zeros for the format length; rounds to number of decimal pl
#	a digit as for 0 but does not display leading or trailing zeros
?	a digit as for 0 but displays a space for leading or trailing spaces for the format length
.	decimal placeholder
	percentage placeholder
E-, E+, e-, e+	displays the number in scientific notation
$, -, +, (,)	display exactly as you type them in
P	character fill; Px fills the front of the string with the character x to make the string the required length
;	section separator
D	D (or d) can be used in place of . forcing Omnis to add at most the decimal places specified in the format, w trailing decimal point

The Numeric format string contains up to four sections: which format positive values, negative values, zero or empty values, and NULL values respectively. An empty format section consisting of two contiguous semicolons will cause the positive format section to be used. Additionally, if the format string contains less than four sections, the positive section will be used for the unspecified sections. Null values will only be formatted using the NULL section.

Example numeric format strings

Format string	1234.47	-1234.47	0 or Empty	N
0	1234	-1234	0	
0.0	1234.5	-1234.5	0.0	
#,##0.00	1,234.47	-1,234.47	0.00	
#,##0;(#,##0)[red]	1,234	(1,234)	0	
0;(0);'Zero';'Nil'	1234	(1234)	Zero	N
0.00E+00	1.23E+03	-1.23E+03	0.00E+00	
+Px#,###,###;-Px#,###,###	+xxxx1,234	-xxxx1,234	+xxxxxxxxx	

The number formatting strings for the current library are stored in #NFORMS. You can edit #NFORMS by double-clicking on it in the Browser.

Date Format Strings

To format a date field you have to set its $formatmode property to Date. The display formats of all date and time fields are controlled by date format strings. #FD is the date format string which is used to display short dates, #FT is the date format string which is used to display short times, and #FDT is the default date format string which is used to display long dates.

Date format strings contain twenty special characters that denote the positions where the string displays the year, month, day, hour, minute, second or hundredths of second. All other characters in the date format string display unchanged (note, for example, the colons in the sample strings below). The Date codes item on the Constants tab in the Catalog contains a list of all the special date format characters. There are options to display the hour in 24 or 12 hour format with an AM/PM position.

N is the character for displaying minutes; M and m indicate the month.

Using the date and time of 20 minutes past 1 p.m. on the 12th of January 1994, a date time value displays as:

- 12 JAN 98 13:20 if #FDT is 'D m Y H:N'
- 12 JAN 98 1:20 PM if #FDT is 'D m Y h:N A'
- 12th 01 1998 13:20:00.00 if #FDT is 'd M y H:N:S.s'

The date formatting strings for the current library are stored in #DFORMS. You can edit #DFORMS by double-clicking on it in the Browser.

Boolean Format Strings

To format a boolean field you have to set its $formatmode property to Boolean. Format strings for boolean fields contain up to three sections: the first formats True values, the second formats False values, and the third formats NULL or Empty values. You can use the following formatting symbols:

Symbol	Description
t	displays "T" or "F" for true or false values
T	displays "True" or "False" for true or false values
y	displays "Y" or "N" for true or false values
Y	displays "Yes" or "No" for true or false values
1	displays "1" or "0" for true or false values
O	the letter "O"; displays "On" or "Off" for true or false values

Example Boolean format strings

Format string	1	0	NULL or Empty
T	True	False	
'True';'False'	True	False	
T;Y;'Null Boolean'	True	NO	Null Boolean

The boolean formatting strings for the current library are stored in #BFORMS. You can edit #BFORMS by double-clicking on it in the Browser.

Input Masks

Input masks control the format of data entered by the user. The input field for a field is stored in its $inputmask property. When you click on the $inputmask property, a dialog appears that lets you select a mask.

You can enter a mask directly into the Input Mask field, either from the keyboard or using the buttons in the dialog. Alternatively, you can click on the down arrow in the Input Mask field and select a mask from the #MASKS system table. The input masks for the current library are stored in #MASKS, which you can edit by double-clicking on it in the Browser.

An input mask can contain a number of characters together with literal display characters. The literal characters are presented to the user when the mask is used for data entry in order to provide context to the surrounding mask placeholder characters. The mask characters can either consist of placeholders or mask control characters. Placeholders are replaced by user characters of the appropriate type during data entry.

You can use the following mask placeholders:

Placeholder	Meaning
#	any digit
@	any character
a	any letter
A	any uppercase letter
n	alphanumeric
N	alphanumeric, upper-cased
“ABC”	any character from list, i.e. either A, B or C
“A-D”	any character from A to D inclusive

Mask control characters control how the mask is presented to the user and how the data is saved to the underlying field. You can use the following control characters:

Symbol	Meaning
^	stores literal characters in underlying field
\C	displays next character literally
>C	uses following character to prompt user
>>	displays default prompt characters

By default, the underscore character is used at data entry to represent placeholder characters yet to be entered. You can configure this character on a per placeholder basis using the ‘>’ symbol. When the character sequence ‘>>’ occurs at the start of the input mask, the default numeric prompt will be a hash sign; other placeholders are displayed as an ampersand.

The following table contains some example input masks, together with the string that is initially displayed to the user and an example value that can be entered to satisfy the mask.

Input mask	Initial display	Example value
(###) ###-####	(___) ___-____	(717) 321-8745
>>(###) ###-####	(###) ###-####	(717) 321-8745
aa ## ## ## a	__ __ __ __ _	xy 12 34 56 z
>>aa ## ## ## a	@@ ## ## ## @	xy 12 34 56 z
>?AA >*## ## ## >?A	?? ** ** ** ?	XY 12 34 56 Z
E\nter digit #	Enter digit _	Enter digit 1
>>aaaaaaaa	@@@@@@@@	Antelope
>>aaaaaaaa	@@@@@@@@	Baboon
>¿‘0-5”	?	4

Note in the above example “E\nter digit #” you need to add a \ before the “n” character to force it to appear rather than it being interpreted as an alphanumeric placeholder.

By default, literal characters occurring in the input mask are simply used to aid data entry. They are not saved to the underlying variable or field. Therefore, when performing queries on the saved data the user must remember not to search for the literal characters. In the first example above, the string ‘7173218745’ would be saved. To show this data correctly, you must add a display format to the entry field.

To enable the user to save literal characters to the underlying variable field, you can put a circumflex character ‘^’ in the input mask. In the first example above, an input mask of ‘^(###) ###-####’ would cause the string ‘(717) 321-8745’ to be saved. In this case it would be inappropriate to place a display format on the associated entry field.

Format String Notation

The text, number, Boolean, and Date format groups are represented in the Omnis notation in their own notation groups, and stored as part of the library preferences, lib.$prefs:

- **$textformats**
  group of text formats
- **$numberformats**
  group of number formats
- **$booleanformats**
  group of Boolean formats
- **$dateformats**
  group of data formats

The standard group notation applies to these groups including $makelist(). Each line in these groups has the $text property containing the corresponding the format.

The following examples load a format from each respective group and format a string using the *format()* function:

```
Calculate mask as $clib.$prefs.$textformats.4.$text
# text format is '('@@@@')' @@@@@@ 'Ext.'@@@'
Calculate strvar as '01728652200221'
Calculate fmtStr as format(mask,strvar)
# fmtStr contains ‘(01728) 652200 Ext.221’
Calculate mask as $clib.$prefs.$numberformats.4.$text
# number format #,##0.00
Calculate numvar as 1589663
Calculate fmtStr as format(mask,numvar)
# fmtStr contains ‘1,589,663.00’
Calculate mask as $clib.$prefs.$booleanformats.4.$text
# Boolean format is y
Calculate boolvar as kTrue
Calculate fmtStr as format(mask,boolvar)
# fmtStr contains Y
Calculate mask as $clib.$prefs.$dateformats.4.$text
# date format is H:N:S
Calculate dat as #D
Calculate fmtStr as format(mask,dat)
# fmtStr contains 15:51:31
```

Remote masked entry fields

The different values for $formatstring and $inputmask are stored in a range system tables (#TFORMS, #NFORMS, #DFORMS, #BFORMS for Character, Number, Date, Boolean formatting strings, and #MASKS for input masks). Omnis stores the property value as an index into the appropriate system table. This means that changing the entry in the system table changes the format/mask stored and used by the object. For remote form masked edit fields, this does not apply, rather the mask or format is stored with the object, and a change to the system table on the Server at runtime will not affect the remote masked entry field.

Drag and Drop

Drag and drop is a powerful feature that lets the user copy data and objects from one field to another, or from one window to another. For example, in a human resources application you could build a list of employees and allow the user to select certain employees and drag them onto a print button to print those employee details; in a stock control system, the user could add items to a dispatch note by dragging the items from a stock list into the dispatch window; and so on.

The drag and drop capabilities of a field are properties of the field itself. Windows also have some drop properties. You can set these properties under the Action tab in the Property Manager, or you can use the notation. The field properties are

- **$dragmode**
  sets whether or not the data or whole object is dragged and/or duplicated: includes kNoDragging (the default), kDragData (drags the data only), kDragDuplicate (drags a copy of the object), kDragObject (moves the object without copying)

- **$dragrange**
  limits the scope of where a field can be dragged to: includes kRangeAll (can be dragged anywhere in the application), kRangeTask (within the current task), kRangeSubwindow (within a subwindow if the field is in a subwindow), kRangeWindow (within the current window only)
- **$dragiconid**
  sets the icon for the object while it is being dragged
- **$dropmode**
  determines what types of object or objects the field will receive: includes kAcceptAll (all types of Omnis control, but not system files), kAcceptButton, kAcceptComboBox, kAcceptDroplists, kAcceptEdit, kAcceptGrid, kAcceptList, kAcceptNone, kAcceptPicture, kAcceptPopMenu, kAcceptOperatingSystem

For a field which is not in a subwindow, kRangeSubwindow is equivalent to kRangeWindow. The drag range is ignored when the drag mode is kDragObject since a field can be moved only within its own window or subwindow.

Drag and Drop Events

Having set the drag and drop properties of fields and/or windows, you need to write event handling methods for these objects to handle events when dragging and dropping occurs. Drag and drop actions generate four events, in the order

- **evDrag**
  the mouse is held down in a field and a drag operation is about to start, the event parameters are: pEventCode, pDragType, pDragValue. It is sent to the field being dragged.
- **evCanDrop**
  a drag operation has started to test whether the field or window containing the mouse can accept a drop, the event parameters are: pEventCode, pDragType, pDragValue, pDragField. It is sent to the field might receive the drop.
- **evWillDrop**
  the mouse is released at the end of a drag operation, the event parameters are: pEventCode, pDragType, pDragValue, pDropField. It is sent to the field being dragged.
- **evDrop**
  the mouse is released over the destination field or window at the end of a drag operation, the event parameters are: pEventCode, pDragType, pDragValue, pDragField. It is sent to the field being dropped on.
- **evDragFinished**
  sent to the dragged field after a drag and drop operation has been completed or cancelled. Its only event parameter is the event code

If a field can accept the object or data that you are currently dragging onto it, it will become highlighted and the appropriate event messages are sent to the field. Depending on the drag and drop mode, evDrag and evWillDrop are both sent to the dragged field, and evDrop is sent to the drop field. For the kDragObject and kDragDuplicate modes, the move or duplicate is not performed if you discard the evDrop message.

The event parameters are

- **pDragType**
  the drag mode of the field being dragged
- **pDragField**
  a reference to the field being dragged
- **pDragValue**
  the object or data being dragged: text, numbers, list data, and so on
- **pDropField**
  a reference to the destination field

All the drag and drop events supply pDragType and pDragValue event parameters. Initially pDragType contains the drag mode of the field being dragged, but you can change it in any of your event handlers. If pDragType is changed by evDrag, the subsequent evCanDrop, evWillDrop and evDrop will see the changed value, but changing it does not affect the drag mode. To avoid confusion with built-in drag operations it is recommended that your drag types are all greater than 1000.

Using Drag and Drop

Consider a window in which the user selects a substring of text in one field fDrag and drags it onto another text field fDrop, which will then highlight the inserted string. First you must set the drag and drop mode of the fields, either in the Property Manager or using the notation

```
Do Win1.$objs.fDrag.$dragmode.$assign(kDragData)
Do Win1.$objs.fDrop.$dropmode.$assign(kAcceptEdit)
```

You must also set up a variable name in the $dataname property for each field, perhaps String1 and String2.

You can trap the events in the field methods, but it may be more convenient to handle them all in the window $control() method, in the case when all the fields in the window have consistent drag and drop handling. The code line

```
Quit event handler (Pass to next handler)
```

at the start of each field method will pass all events to the window $control() method. In the window $control() method, you can add event handlers to detect the drag and drop event messages evDrag, evDrop, evCanDrop, and evWillDrop, with the following structure.

```
# $control() method in window
On evDrag
  # do this

On evDrop
  # do this

On evCanDrop
  # do that

On evWillDrop
  # do the other
```

The evCanDrop event is sent frequently during a drag operation, so this handler must be short and efficient: it should *not* do anything to change the appearance of the user interface, such as displaying a message or opening or closing a window.

You could choose to handle evDrop only, which provides the value of the dragged substring in the parameter pDragValue, and ignore the other events. When the mouse is released over the fDrop field, triggering evDrop, you can use the *mouseover()* function to return the position of the mouse pointer in the text string. The $mouseevents library preference must be turned on for your library to send and receive mouse events. You can also use the string functions *con()* and *mid()* to insert the dragged string into the fDrop field at the right place. You can highlight the inserted substring using the properties $firstsel and $lastsel.

```
# $construct() method for the window
# declare variable Pos of type Number
On evDrop
  Calculate Pos as mouseover(kMCharpos)
  Calculate String2 as con(mid(String2,1,Pos),pDragValue,mid(String2,Pos+1,len(String2)-Pos))
  Do $cobj.$firstsel.$assign(Pos)
  Do $cobj.$lastsel.$assign(Pos+len(pDragValue))
  Redraw (Refresh now) {fDrop}
```

Another frequent use of drag and drop is moving selected lines between lists. Consider two fields lDrag and lDrop that use the list variables List1 and List2. lDrag should have its $multipleselect property set and a drag mode of kDragData, and lDrop should have its $dropmode property set to kAcceptList.

When the drop occurs, pDragValue has a copy of List1 and not just the selected lines: these can be merged with List2 and removed from List1.

```
# $control() method for the window
# declare variable iList of type List
On evDrop
  Set search as calculation #LSEL
  Set current list List2
  Calculate iList as pDragValue
  Merge list iList (Use search)
  Set current list List1
  For each line in list (Selected lines only) from 1 to #LN step 1
    Delete line in list ## remove dragged lines
  End For
  Redraw lists (All list)
```

Having merged the dragged lines into List2, you can sort the list.

Alternatively, you might wish to insert the line or lines at a specific place in List2. In this case, you need to use the *Insert line in list* command to insert each required line at the mouse pointer position in List2 using the *mouseover(kMLine*) function

```
Set current list List1
Calculate InsertPoint as mouseover(kMLine)
For each line in list (Selected lines only,Descending) from 1 to #LN step 1 \
     ## descending order, so as to insert in ascending order
  Load from list
  Delete line in list
  Set current list List2
  Insert line in list {InsertPoint}
  Set current list List1
End For
Redraw lists (All list)
```

A drag mode of kDragObject can be useful to give users the chance to rearrange fields on a window.

Operating System Files

You can drop external system files onto window class fields. For example, you can drop a file into a node in a tree list. Note that any folders are included in the list of dropped objects, with a size of zero.

There is an example app to demonstrate system file drag and drop on the Omnis GitHub repo at: https://github.com/OmnisStudio. Search for Omnis-SystemDrop. The same app is available in the **Samples** section in the **Hub** in the Studio Browser.

Notes for existing users

Support for dragging and dropping *operating system files* and *file data* (in the thick client) was simplified and improved in Studio 10.2. The old-style Windows file drag and drop is no longer supported, but can be enabled using the "classicwindowssystemdragdrop" item in the "windows" section of config.json.

Drop mode

To accept dropped files or file data from outside Omnis, $dropmode for a field must be set to kAcceptOperatingSystem. Note that kAcceptAll does not include system files (kAcceptOperatingSystem); kAcceptAll means accept drops from all types of Omnis control.

Drop mode flags

Window controls that have the $dropmode property (as well as the thick client window itself) have a $osdropflags property to control what is dropped (data or files). This is the combination of a number of constants that can be used to specify what is dropped:

- **kOSDROPincludeData**
  If set in $osdropflags, pDragValue will include the data for objects dropped from the operating system if the data is available and kOSDROPwithoutDataIfOsDropLimitExceeded allows.
- **kOSDROPfilesOnly**
  If set in $osdropflags, objects dropped from the operating system must all be files. Note that on macOS, data provided by file promises may not be accepted, because the file containing the data may only become available when dropping the object(s).
- **kOSDROPwithoutDataIfOsDropLimitExceeded**
  If set in $osdropflags, $clib.$prefs.$osdroplimit can be exceeded (in which case data is not included in pDragValue for evDrop)

On conversion, objects that were set to kAcceptFileData are now set to kOSDROPincludeData.

Drop data limit

The library preference $osdroplimit ($clib.$prefs) sets the maximum number of bytes of dropped data that can be included in pDragValue for evDrop when $osdropflags contains the flag kOSDROPincludeData; it defaults to 100000000 (100MB). The setting kOSDROPwithoutDataIfOsDropLimitExceeded specifies if evDrop still occurs when the limit is exceeded.

Note that combining kOSDROPincludeData and kOSDROPwithoutDataIfOsDropLimitExceeded with a suitable drop limit provides a good way of accepting files of arbitrary size (where the data is too large to be read into the drag value) and also accepting other non-file objects.

Note that if you change $osdroplimit, any windows relying on its value need to be closed and re-opened.

Event Parameters

There have been some changes with the parameters for the evDrop event: pDragType was previously set to kDragFiles but is now set to kDragOperatingSystem. While kDragFileData is renamed to kDragOperatingSystemData_OBSOLETE, and resolves to the same value as kDragOperatingSystem.

The pDragValue list uses the same column names as previous versions (for compatibility), and has an additional column called **isfile,** a Boolean, which is true if the dropped object is a file.

Notes that **filedata** is always a binary value containing the contents of the object, and **filesize** is a 64 bit integer.

When dropping email from macOS Mail, pDragValue has the same content (with these additional columns) as previous versions.

File extension (macOS)

For mac OS, when a file is dragged a fourth column pFileextmac has been added to the pDragValue parameter which returns the macOS file extension, or is empty if the data is from an unknown application.

Toast Messages

Toast messages are small notification type messages that that can be “popped up” in your application to alert the end user about something. You can open toast messages in your desktop apps, via a window instance ($cinst) using the $showtoast method.

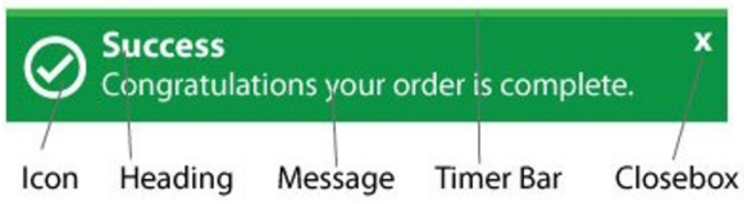
There is an example app called **Toast Messages** in the **Samples** section in the **Hub** in the Studio Browser. The same app is available on the Omnis GitHub repo at: https://github.com/OmnisStudio; search for Omnis-Toasts.

Toast Properties and UI

Toast messages have a title, message and an icon, and can be positioned in the top-left, top-right, bottom-left, or bottom-right relative to the Omnis application window (not the desktop) under Windows, or the whole screen on macOS. They will close automatically after 4000 ms or a specified time.

Toast messages are non-modal, and therefore they are outside the scope of other Omnis window stacks and do not interfere with evToTop message processing, nor do they change the end users current window or current focused object.

They have the following UI layout:



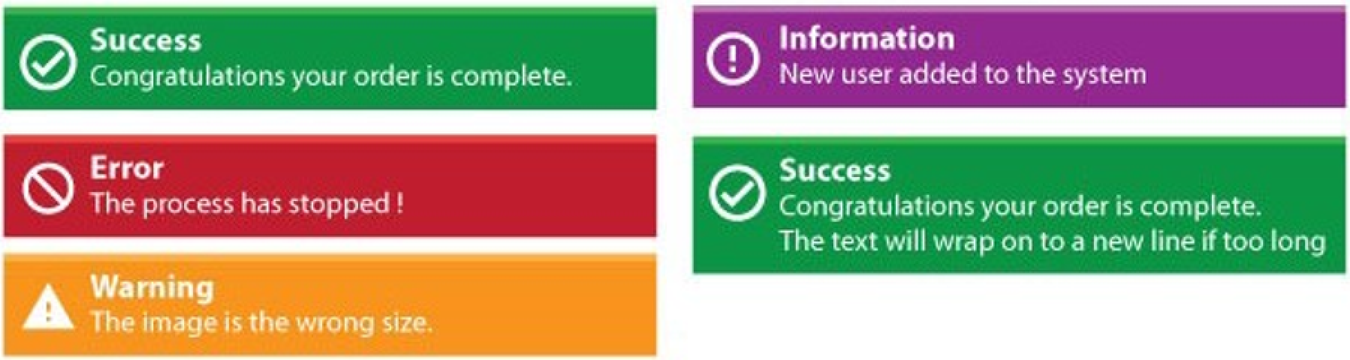
Success
Congratulations your order is complete.

Figure 162:

The following are some example toast messages:

Toast messages have close boxes so they can be dismissed before the default time expires. They have a set of predefined colors and default width with the option to override these colors by setting some new appearance theme settings. The message box has a progress timer bar to show how long remains before the message box is dismissed.

Toast messages can be stacked up to 6 levels deep. When a message is manually closed or expires, it is removed from the stack and other on-screen messages adjust keeping a stacked appearance.



Success
Congratulations your order is complete.

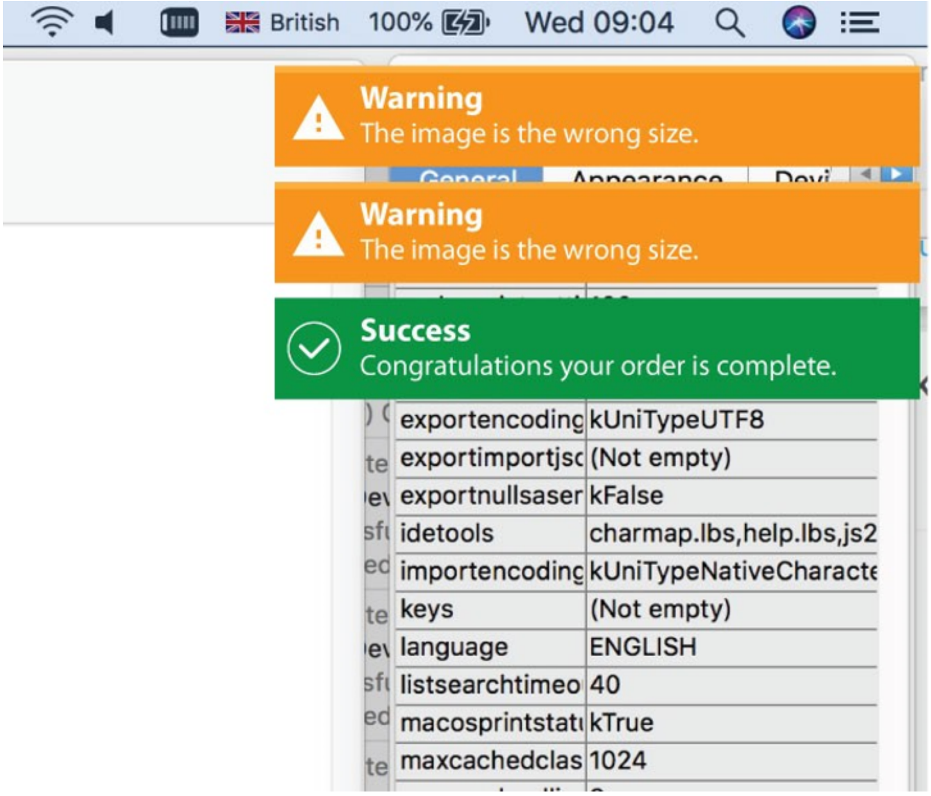
Information
New user added to the system

Error
The process has stopped !

Success
Congratulations your order is complete.
The text will wrap on to a new line if too long

Warning
The image is the wrong size.

Figure 163:



Warning
The image is the wrong size.

Warning
The image is the wrong size.

Success
Congratulations your order is complete.

exportencoding	kUniTypeUTF8
exportimportjsc	(Not empty)
exportnullsaser	kFalse
idetools	charmap.lbs,help.lbs,js2
importencoding	kUniTypeNativeCharacte
keys	(Not empty)
language	ENGLISH
listsearchtimeo	40
macosprintstatı	kTrue
maxcachedclas	1024

Figure 164:

Showing Toast Messages

Toast messages are opened in the context of the current instance (window or object instance) using the **$showtoast()** method, which has the following parameters.

- $cinst.$showtoast(cMessage [, cTitle, iStyle=kToastSuccess, iStack=kToastStackTopRight, iTimer=4000, bClearStack=kFalse, cContext=”]) adds a new toast to a stack.

Parameter	Description
cMessage	the message text; must be supplied
cTitle	Title text, or leave empty ‘’ for no title | | iStyle | kToastSuccess (the default), kToastError, kToastWarning, kToastInformation | | iStack | the position of the message stack relative to the application window or screen on macOS, a constant: kToastStackTopRight (the default), kToastStackTopLeft, kToastStackBottomRight, kToastStackBottomLeft, kToastStackCenter | | iTimer | number of milliseconds the toast will be displayed (default is 4000, as set in the ‘toast’ section of the appearance.json theme file)
bClearStack	if kFalse (the default) new toast messages are stacked with previous messages, otherwise if passed as kTrue, all previoustoasts in the stack will be cleared
cContext	text passed into $toastnotificationclicked when the content of the toast is clicked (blank by default): see below.

For example:

```
$cinst.$showtoast('Success','Congratulations your order is complete', kToastSuccess )
$cinst.$showtoast('Error','Problems with your connection.', kToastError, kToastTopLeft )
```

Note: Attempting to add a toast to the same stack with an identical title and message will reset the existing toast timer and not add a new toast.

Notification Clicks

The startup task method $toastnotificationclicked() method allows you to determine when a toast notification message has been clicked. The $showtoast() method has an optional parameter pContext, which can be passed into $toastnotificationclicked when the content of the toast is clicked, and after querying pContext your code could take some specific action, such as opening a window.

If you return kTrue from $toastnotificationclicked, the toast is closed immediately. $toastnotificationclicked is not called if clicking the toast close box.

Toast Message Colors

You can override the default colors of the four toast styles, the onscreen delay, and the default width in the ‘toast’ section of the appearance.json file.

```
toast
{
  toastsuccessbackgroundcolor : color,
  toastsuccesstextcolor : color,
  toastsuccessicon : number,
  toasterrorbackgroundcolor : color,
  toasterrortextcolor : color,
  toasterroricon : number,
  toastwarningbackgroundcolor : color,
  toastwarningtextcolor : color,
  toastwarningicon : number,
  toastinformationbackgroundcolor : color,
  toastinformationtextcolor : color,
  toastinformationicon : number,
  toastdefaultdelay: 4000,
  toastdefaultwidth: 400
}
```

Example

The following example code opens different toast messages:

```
If (toastType>kToastInformation)
  Calculate toastType as kToastSuccess
End If
Calculate type as toastType
Switch type
  Case kToastSuccess
    Calculate title as "Success"
    Calculate message as "Congratulations. Logon complete."
  Case kToastError
    Calculate title as "Error"
    Calculate message as "Sorry your details are incorrect. Please check your deails and try agin."
  Case kToastWarning
    Calculate title as "Warning"
    Calculate message as "Please check your settings"
  Case kToastInformation
    Calculate title as "Information"
    Calculate message as con('Great news. All formst have been submitted, saved and has passed ',      styl
End Switch
Calculate toastType as toastType+1
```

Window Messages

The **$showmessage()** method opens a message in a window instance, which you can use as an alternative to the OK message command (this is similar to the $showmessage() method which is available in remote form and remote task instances). The $showmessage() method is also available for menu, toolbar, report, object, and table instances. The method has the following definition:

- **$showmessage**(cMessage[,cTitle,iOptions=kMsgOK])
  displays a message using the specified cMessage, cTitle and iOptions (a sum of kMsg... constants). Returns true for OK or Yes, false for No or cancel. You can use msgcancelled() to check for cancel.

The supported constant values are:

Constant	Description
kMsgOK	Display an OK message (the default)
kMsgYesNo	Display a Yes/No message
kMsgNoYes	Display a No/Yes message
kMsgCancelButton	Add a cancel button to the message
kMsgIcon	Display an operating system specific icon with the message
kMsgSoundBell	Sound the bell when the message is displayed

If you mix kMsgYesNo, kMsgNoYes and kMsgOK, kMsgYesNo has precedence over kMsgNoYes. kMsgNoYes has precedence over kMsgOK.

HWND Notation

All window instances and their objects, except background objects, have the $hwnd and $framehwnd properties. In addition, window instances have the $toplevelhwnd property. These three properties all identify child windows or parts of a window. Each window object has a $framehwnd, which is the outermost enclosing child window of the object. Each window object also has an $hwnd, which is the child window which typically contains the main information displayed by the object. $hwnd is always contained in $framehwnd, and in many cases $hwnd and $framehwnd are the same child window.

For example, in the following field $hwnd is not the same as $framehwnd: $hwnd refers to the client window excluding the title window, and $framehwnd refers to the frame window.

For a window instance, $toplevelhwnd is the outermost enclosing child window of the window instance, that is, it corresponds to the complete window, including title bar and sizing border, if present. $framehwnd of an open window instance is the window contained in the $toplevelhwnd; it excludes the window title bar and sizing borders. $hwnd of an open window instance is contained in $framehwnd, together with the window menu bar, toolbar and status bar, if present. For example, when you use the notation

```
Do $cwind.$hwnd Returns lvNumber
```

$hwnd, $framehwnd, and $toplevelhwnd return a number. The number is a unique identifier that represents the child window.

$hwnd, $framehwnd, and $toplevelhwnd can also return an item reference to the child window, for example

```
Set reference myRef to $cinst.$hwnd.$ref
```

For $hwnd and $framehwnd, the item reference supports the following properties: $left, $top, $width, $height, $clientleft, $clienttop, $clientwidth, and $clientheight. $toplevelhwnd supports the following properties: $left, $top, $width and $height.

A child window can have a client area and a non-client area. The non-client area of the window can contain features such as the window border and scroll bars. Sometimes the non-client area is empty, such as in a borderless entry field with no scroll bars. Consider an entry field with a 2 pixel inset border. Its client area sits inside the non-client area, as follows.

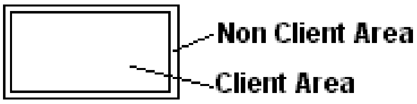


Figure 165:

The entry field with a 2 pixel inset border may be 100 pixels wide and 50 pixels high. The client area would be 96 pixels wide and 46 pixels high. Therefore

```
$width = 100 $clientwidth = 96
$height = 50 $clientheight = 46
```

$top and $left are the coordinates of the child window, relative to the child window which contains it (in the case of $toplevelhwnd, these coordinates are relative to the area in which Omnis displays window instances).

$clientleft and $clienttop are the coordinates of the client area of the child window. These always return the value zero.

The properties of $hwnd and $framehwnd are not assignable. For $toplevelhwnd, $left, $top, $width and $height are assignable.

Screen Size

There is a property of $toplevelhwnd, called $screen, that allows you to track the *location* and *dimensions* of the screen, as the window changes position. This could be useful if, for example, a window in Omnis is located on a second monitor and you want to determine its width and height in order to resize or reposition the window.

The value of $left, $top, $width and $height of the screen can be obtained by creating an item reference to this property, e.g.

Set reference toplevelitemref to $cwind.$toplevelhwnd.$ref

```
Set reference screenref to toplevelitemref.$screen.$ref
```

This is only implemented for macOS and Windows. Other platforms (Linux) will return (0,0,1,1) for (left, top, width, height).

macOS

The screen dimensions on macOS will take account of the menubar and position of the dock and only return the visible screen area. This can be controlled using the **excludeDockFrom$screen** item in the 'macos' section of config.json, which is true by default, which means the $screen property excludes a visible macOS dock from its coordinates.

Windows

Under the Windows OS all Omnis windows are contained within the main Omnis window. Therefore, on Windows the identifier for a window's screen will be the screen containing the main Omnis window.

If the main Omnis window is displayed across multiple screens then the identifier for the screen is that screen which contains the largest area of the main Omnis window. The screen properties will provide the area of the main Omnis window which intersects the visible area of the screen.

Using HWND

Sometimes it is important to know the exact size of the client area. For example, if a window has a toolbar on the left and you want to create controls right-justified down the right edge f the window, $cwind.$width would not be good enough, as it includes the width of the toolbar. This would cause you to add controls (using $add) too far to the right.

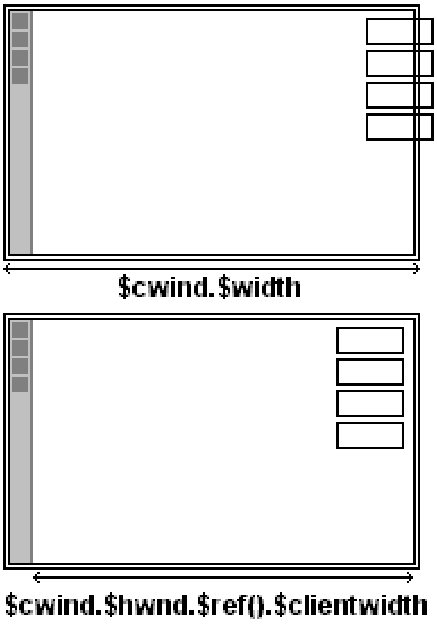


Figure 166:

In the first example you call $add() to add objects and the $left for the objects would be the width of the window less the $width of the objects you are adding. As $width included the toolbar space, the objects would be added too far to the right.

In the second example, using the $hwnd of the window, you get the exact width excluding the toolbar width, allowing you to right justify the controls correctly.

Enter Data Mode

You can place window instances in enter data mode on a case-by-case basis. A window instance has the $enterable attribute; if true, the window is in enterable mode. Enterable means the window is in 'enter data mode' so data can be typed into the entry fields and any OK and Cancel buttons are enabled. Normally windows with modeless enter data are always enterable and other windows are enterable when there is an executing *Enter data* command on the method stack. When you set $enterable to kTrue for a window instance it is never changed automatically by Omnis at an *Enter data* command, therefore

```
Do $cinst.$enterable.$assign(kTrue)
```

in the $construct() method of the window is equivalent to putting the window in modeless enter data mode.

It is possible to go into enter data mode without the top window (or any window) being enterable. Sometimes this might be desirable, but beware Omnis provides no protection against this situation.

If a window is not enterable for Enter data it is also not enterable for Prompted find.

Floating Edges for Windows and Fields

Window classes and all field types have the $edgefloat property which affects how the object is resized and/or moved when its container window/field is resized. The default value of $edgefloat is kEFnone. For a window class, the $edgefloat property affects how the edge or edges of the window are resized or repositioned when the main Omnis application window is resized, or when the monitor resolution is changed. For fields and other objects, the $edgefloat property affects how a field is resized or moved when its container window is resized. You can apply $edgefloat properties to fields within other container fields, such as a tab pane or shape field, in which case their floating edge or positioning properties are relative to the area within the container field.

The $edgefloat property for a window class or field can have one of the following settings (an Omnis kEF… constant):

Constant	Description
kEFall	All the edges of the window/field can float; this means the window or field will move as the main Om effect, the window or field will remain fixed relative to the bottom-right corner of its parent window o
kEFbottom	The bottom edge of the window/field will move with the main Omnis window or container window
kEFleftRight	The left and right edges of the window/field will move with the main Omnis window or container win unchanged
kEFleftRightBottom	The left, right, and bottom edges of the window/field will move with the main Omnis window or cont
kEFnone	No floating edges; the window/field is unaffected when you resize the main Omnis window or contai
kEFright	The right edge of the window/field will move with the main Omnis window or container window
kEFrightBottom	The right and bottom edges of the window/field will move with the main Omnis window or containe
kEFrightTopBottom	The right, top and bottom edges of the window/field will move with the main Omnis window or cont
kEFtopBottom	The top and bottom edges of the window/field will move with the main Omnis window or container v unchanged

In addition, you can set a field's $edgefloat property to one of the kEFposn… or positioning constants which affect how a field is positioned (and resized) relative to the edge of its container window class or container field, such as a tab pane.

Constant	Description
kEFposnBottomToolBar	Places field at the bottom of the window or container field
kEFposnClient	Field resizes to fit the area that is available around the field
kEFposnHorzHeader	Places field in the horizontal header or top of the container field or window
kEFposnJoinHeaders	Field is located where the horizontal and vertical headers meet
kEFposnLeftToolBar	Places field at the left-hand edge of the window or container field
kEFposnMainHeader	Places the field in the main header of the container field or window
kEFposnMenuBar	Places the field at the top of the window or container field; such a field will be placed above fields and kEFposnHorzHeader
kEFposnRightToolBar	Places field at the right-hand edge of the window or container field
kEFposnStatusBar	Places field in status bar position at the bottom of the window
kEFposnTopToolBar	Places field at the top of the window or container field
kEFposnVertHeader	Places field in the vertical header or left side of the container field or window

Note that some of these positioning constants take precedence over others. For example, a field with kEFposnMenuBar will always be positioned at the top of a window or container field. If you add a field with kEFposnTopToolBar positioning it will appear under a field with kEFposnMenuBar positioning, if there is one, otherwise it will appear at the top of the window.

The following window shows the different kEFposn… settings for fields within a window class. Fields within a container field, such as a tab pane, behave in the same way. Note the kEFposnClient setting will make the object expand to fit the available area in the window or container field.

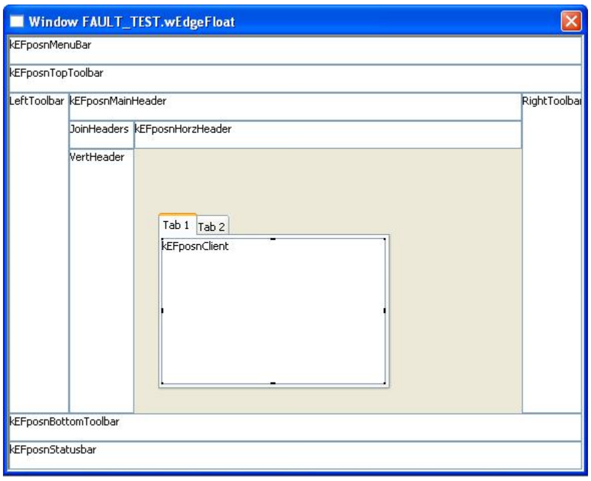


Figure 167:

Window Minimum Size

The **$minwidth** and **$minheight** properties allow you to set a minimum width and height for a window. This can be useful if you want to stop the end user making a window containing many floating fields too small, to preserve its layout, for example.

$minwidth and $minheight are the minimum values to which the $width or $height properties of a window can be set, either programmatically or by the user resizing the window. A value of zero (the default) means that no minimum is specified.

Creating a right floating window

The following example shows how you can use $edgefloat, in conjunction with some code in the $construct() method of a window, to create a window that will attach itself to the right-hand side of the main Omnis window (or desktop under Mac OS), even when the Omnis window is resized. Set $edgefloat to kEFleftRightBottom, and use the following code in the $construct() methods of the window.

```
Set reference item to $cinst.$toplevelhwnd.$ref
Calculate item.$left as $root.$modes.$width - item.$width
Calculate item.$top as 0
Calculate item.$height as $root.$modes.$height
```

If you resize the Omnis window (under Windows), or change the monitor resolution (Mac OS), the window remains attached to the right-hand side.

Window Fonts

If you are developing an application for a cross platform environment, you may want to set up the system font tables to allow the fonts used in your application to map correctly across the different platforms. There is a system font table for window classes and report classes for each platform supported in Omnis.

The fonts in the window font table will appear in the $font property for window objects. So even if you are developing an application for a single platform, you may still want to edit the window font table to add fonts to those already available for window objects.

Font table	Description
#WIWFONTS	Window font table for Windows OS
#UXWFONTS	Window font table for Unix
#MXWFONTS	Window font table for macOS

To view the window fonts system table

- Use the Browser Options dialog (press F7/Cmnd-7 while the Browser is on top) to make sure the system tables are visible in the Studio Browser
- Double-click on the System Tables folder
- Double-click on #WIWFONTS or the window font table for your platform

The #WIWFONTS system table contains a list of fonts that are available in Omnis by default. Each row in the font list displays the corresponding font for each platform supported in Omnis. To change the font mapping, replace the name of a font either by typing its name or selecting it from the list of fonts. To add a font, click in the next available line in the font list and add the name of the font. Add a font name for each platform.

The font table editor loads or creates a font table for each platform (corresponding to each column in the editor) and allows you to edit them all simultaneously. Therefore when you edit the window font table for the first time, and click OK to finish editing it, a new system table is added to your library for each platform supported in Omnis, other than your current platform.

Window and Font Scaling

Computer monitors are increasingly available in a variety of screen resolutions, therefore your applications must be capable of scaling to fit different display devices. There are some library preferences to control the scaling of windows and fonts for applications displayed on monitors that have a different resolution from the standard 96 DPI for PCs or 72 DPI for Macs.

- **$designdpi**
  The DPI of the monitor which the library was designed on. This will default to 96 for PCs or 72 for Macs, if it is currently not set. This value is used to scale the application accordingly when $designdpimode is on. For example, if you have an application that was designed on a 120 DPI machine, you should set $designdpi to 120. When such an application is opened on a 96 or 72 DPI monitor, and $designdpimode is on, the windows and fonts will scale down.

- **$designdpimode**
  Turns designdpi mode on or off, one of three values: kDPIoff (the default) no scaling occurs, which equates to the behavior of previous versions of Omnis Studio; kDPIframeOnly means that window and control frames will be resized, but not fonts; kDPIall specifies that all window and object frames, as well as fonts, are scaled.

When you assign these preferences you must close and reopen your library for them to take effect.

Note that the $left, $top, $width and $height of windows and objects are also affected as appropriate when $designdpimode is on and an application is scaled.

Menu and Toolbar Fonts (Windows)

You can scale menu and toolbar fonts when using design DPI scaling. Specifically, window menus and window toolbars, under Windows, scale using the design DPI preferences and settings for the library.

There are some entries in the **config.json** file that can be used to control scaling of menus and toolbars for cases where Omnis cannot easily determine a library to be used as the source of the scaling settings. These entries are:

- defaultMenuDesignDPIMode and defaultMenuDesignDPI
  in the “windows” section (these only apply on the Windows platform: note that menu items are never scaled on macOS)
- windowToolbarDesignDPIMode and windowToolbarDesignDPI
  in the “ide” section
- dockingAreaDesignDPIMode and dockingAreaDesignDPI
  in the “ide” section

The syntax for these entries is

- the mode entries: “kDPIall”, “kDPIoff” or “kDPIframeOnly”
- the DPI values: 3 comma separated DPI values

This corresponds to the syntax of the library preference (even in the case of menus where only the Windows platform value is used). Typically, you would set these to the same value as those in the library preferences.

Background Themes

Background themes allow you to specify the Aqua theme for the background of an object. $backgroundtheme is a new property of a window object, window background object, a window, and some remote form objects and allows you to set the background theme. It can have the following values:

Constant	Description
kBGThemeNone	No background theme; you can use the other background properties of the object ($forecolor and so o
kBGThemeParent	parent object theme; if this results in kBGThemeNone, the object uses the parent’s appearance proper
kBGThemeWindow	window background theme
kBGThemeContainer	container background theme (usually kBGThemeWindow)
kBGThemeTabPane	tab pane background theme
kBGThemeTabStrip	tab strip background theme
kBGThemeControl	control background theme
kBGThemeMenubar	menu bar background theme
kBGThemeMenu	menu background theme

Note that if you use this property on platforms other than macOS, the background will be filled with the following system colors:

Constant	Description
kBGThemeWindow kBGThemeContainer kBGThemeTabPane kBGThemeTabStrip	kColor3DFace
kBGThemeControl	kColorWindow

Constant	Description
kBGThemeMenubarkBGThemeMenu	kColorMenu

Since $backgroundtheme uses sensible values for platforms other than macOS, you can use the theme to obtain good cross-platform results. The following are some additional recommendations:

- Always set the window’s theme to kBGThemeWindow.
- When designing windows using container objects such as scroll boxes, in order to create sizeable panes, set the container’s theme to kBGThemeParent, unless there is a good reason for the container to use a different background.
- For label background objects set the background pattern to transparent (pattern 15).
- When using edit fields as labels, set $backgroundtheme to kBGThemeParent and set $effect to kBorderNone.

Theme Fonts

Theme fonts allow you to specify a font from the current Aqua theme. They are only available on macOS. You can use Theme Fonts in the $fontname property of a Field Style. The following Theme Fonts are available:

ThemeLabel
ThemePushButton
ThemeApplication
ThemeSystem
ThemeEmphasizedSystem
ThemeSmallSystem
ThemeSmallEmphasizedSystem
ThemeMenuTitle
ThemeMenuItem
ThemeMenuItemMark
ThemeMenuItemCmdKey
ThemeWindowTitle
ThemeUtilityWindowTitle
ThemeAlertHeader
ThemeViews

Note that not all theme fonts on macOS support text underline.

Window Style

You can set the style or type of a window using the $style property. The window $style can be one of the following: kDialog, kNoframe, kPalette, kSimple, or kTitle (the default).

Window Title Colors (macOS)

On macOS, you can use the **coloractivecaption** and **colorinactivecaption** items in the ‘system’ section of appearance.json to set the colors for window title bars. If either of these is set to kColorDefault, the system default colors are used.

In addition, the **macoscaptiontextappearance** item in the ‘system’ section of appearance.json specifies the text color for captions (the window title); this is an integer: 0 system default, 1 dark text, 2 light text.

Drawer Windows (macOS)

Drawer and Sheet windows are available under macOS only. In general, their behavior is ignored under the other platforms and they behave like standard windows. For example, you can specify that OK message boxes are opened as sheet windows: under macOS they will open by overlaying the parent window, but under all the other platforms they will open as normal by popping up over the top of the current window.

A *drawer* is a window that slides out from a parent window and while open remains attached to the parent window. For example, in an appointment calendar application a drawer window could pop out from the main window to display details about each appointment. In Omnis, drawer windows behave in the standard way as defined by the macOS operating system.

When you open a window from inside the current window, the parent window, you can specify that the child window is opened as a drawer either at the top, bottom, left, or right of the parent. The drawer position is selected as one of the window position constants when using the $open() method.

```
# button method inside a parent window
On evClick
  Do $clib.$windows.wDrawer.$open('*',kWindowDrawerDefault)
  # other positions include kWindowDrawerBottom, kWindowDrawerLeft, kWindowDrawerRight, kWindowDrawerTop
```

Window instances have the property $isdrawer which tells you whether the instance is a drawer window or not.

Sheet Windows (macOS)

Window classes have the property $usesheets which enables sheet windows under macOS; on other platforms, windows are opened as normal. If this property is kTrue, and the Window is the top most window (excluding palette windows), any message boxes, such as standard OK and Yes/No messages, and file selection dialogs will be opened as a sheet window in the top window. In this case, message and file selection sheet windows are application modal, that is, the use cannot click on any other Omnis windows while the sheet window is open.

It is also possible to open user windows as sheet windows, by using the constant kWindowSheet with the $open() method, for example:

```
Do $clib.$windows.wMyWindow.$open('*',kWindowSheet,$iwindows.wMyParentWindow.$ref)
```

The sheet window will be attached to the specified parent window. If no parent window is specified, the sheet window is attached to the top user window.

If $clickbehind is kTrue, it will be possible to click on other Omnis windows with the exception of the parent window. If $clickbehind is kFalse, the sheet window becomes application modal (open user windows only) and it will not be possible to click on other Omnis user windows.

Window instances have the property $issheet which tells you whether the instance is a sheet window or not.

Simple Style Windows

The $growbox property is assignable for simple style windows ($style = kSimple), as this controls whether the window has a sizing border. This will allow you to make simple style windows look more alike on macOS and wWindows.

For a window with $style set to kSimple, irrespective of the setting of $growbox, you can always resize the design window. On Windows, there is a wider sizing border in design mode, even if $growbox is kFalse. When you open the window, the open window is only resizeable if $growbox is true, and on Windows it only has a wider sizing border in this case.

The net effect is that for kSimple style windows with $growbox kFalse, there is no wide sizing border on either platform for the open window, making them appear more alike.

Palette Windows

It is recommended that you do not use Palette type windows under macOS due to various constraints with window ordering and their conflict with the window "bringtofront" mechanism in Omnis. You can use one of the other window types, by setting the $style property of the window, such as the kSimple type.

Window Transparency

Window classes now have the $alpha property which sets the transparency of the window and all its controls (an integer from 0 to 255, with 0 being completely transparent and 255 opaque). In addition, the majority of the Window class components have the $alpha property which means you can set the transparency of individual window components.

Note $alpha was available for window classes in versions prior to Studio 8.1 but for macOS only.

Disabling the Focus on Fields

Most window fields have the $disablefocus property which specifies whether or not a field gets the focus when the user tabs through the fields in a window. This is useful, for example, with fields that do not show the focus, such as the trans button: setting $disablefocus to kTrue still allows the user to click on the button, but prevents tabs appearing to go nowhere.

Lookup Windows

In the situation where there may be limited space on your window for a large list field, therefore you might want to place the list on a separate *lookup window*. You can force such a window to open when the user needs to look up the data, and close it as soon as a line is chosen. As a further refinement you can allow the user to enter some data directly into a field and not popup the window, or if the field is left empty popup your lookup window containing a list of possible choices.

An entry window wBookings for the BOOKINGS file, for example, might have the foreign key BkCuId to the primary key CuId in the CUSTOMERS file. The BkCuId field method will check if the code entered by the user is valid and allow the user to choose from a list of customer names if it is not. The important point here is that method execution must be held up until the user has made a choice. These are the methods to implement this: they will be described together since they interact.

```
# $event() method for wBookings field BkCuId
On evAfter
  Do CheckCustCode Returns Valid ## check on code entered
  If not(Valid) ## if invalid or no code
    Open window instance {wCustList} ## open lookup window
    Enter data ## until item is selected
    Close window instance wCustList
    Calculate BkCuId as CuId ## set foreign key ..
    Redraw { BkCuId } ## and show value
    Queue set current field {BkCuId} ## reposition cursor ..
    Queue tab ## to next field
  End If
  Quit event handler (Discard event)
```

The Customers List window wCustList is a Simple or NoFrame style window filled by a list box. The list is defined and built as the window opens. There’s no need to show CuId in the list box but it must be in the list.

```
# $construct() method for wCustList
Set current list cCustList
Define list {CuId, CuLname, CuCountry}
... build list from Omnis or SQL data

# $event() method for list box field
On evDoubleClick ## Event Parameters - pRow ( Itemreference )
  Load from list ## transfer list line values to CRB
  Queue cancel ## terminate enter data mode
  Quit event handler (Discard event)
```

When wCustList opens, the list is built. At this point *Enter data* is necessary to halt execution of the method until the user has chosen from the list. When the list box receives a double-click, *Load from list* transfers the list line data to the CRB. *Queue cancel* now terminates the enter data state so that execution resumes and closes the window. BkCuId is set from CuId entered from the list and the field is redrawn. The cursor will still be in BkCuId so *Queue set current field* and *Queue tab* can be added to place the cursor at the next field in the tabbing order.

This enter data state is needed whether or not the parent window has $modelessdata set.

Timer Methods and Splash Screens

A splash screen makes a more friendly introduction to an application than presenting the user with a blank screen and a menu bar. It involves opening an introductory window, usually called wAbout, from the $construct() method in the startup task of your library. You can keep the About window on screen either for a predetermined time or until the user clicks on it. The wAbout window contains a button area field to detect clicks and the following methods

```
# $construct() method for wAbout
Set timer method (8 sec) {Timer Control}

# $event() method for the button area field
On evClick
  Do method Close Window

# Close Window method
```

```
Clear timer method
Close window {wAbout}

# Timer Control method
Do method Close Window
```

When the $construct() is called, the *Set timer method* command sets a time delay in seconds and nominates a method, called Timer Control, which is run at the end of the delay. The Timer control method then calls Close Window which clears the timer and closes the window. If the window has been clicked on before the end of the delay, the button area method calls the Close Window method which closes the window immediately.

The button area should have its $noflash property set to kTrue to avoid flashing when the user clicks on it.

Pictures

You can place button area fields over a graphic, which you can paste onto your window or load into a Picture field. If the library preference $sharedpictures is set to kSharedPicMode256Color or kSharedPicModeTrueColor, pictures are converted to a format accepted under Windows or macOS. The recommended shared picture mode is kSharedPicModeTrueColor. kSharedPicMode256Color is provided for backwards compatibility with earlier versions of Omnis Studio.

Window Status Bars

A *window status bar* is an area at the bottom of a window in which you can display data, text, help messages, progress or thermometer bars, and so on; for example, there is a status bar at the bottom of the Studio Browser. A status bar is a property of the window itself which you enable in the Property Manager. You can set how many panes should appear in the status bar, and the size and style of each pane.

To enable a window status bar

- Open your window in design mode
- Click on the background of the window to show its properties, or press F6/Cmnd-6 to bring the Property Manager to the top
- Set the **hasstatusbar** property to kTrue
- Click on the Appearance tab in the Property Manager and set the **statusedge** property: it can be flat (the default), plain, inset, or chisel border style

To set the number of panes in the status bar and their style you need to edit the properties of the status bar in the Property Manager.

To set the number of panes in the status bar

- Click on the status bar and bring the Property Manager to the top

The **panecount** property specifies the number of panes in the status bar. The **helppane** property specifies the pane in which any help messages should appear, held in **helptext** for menu lines, for example. On the Text tab you can set the **font** and **fontsize** properties for the whole status bar.

To change the properties of individual panes you should click on the pane and edit its properties in the Property Manager.

To change the properties of a pane

- Click on a pane in the window status bar and click on the Pane tab in the Property Manager

You can change the pane's border, alignment, and width in pixels. The **sizing** property sets the pane to fixed or elastic when the window is sized at runtime. The minimum size of an elastic pane is the size of the pane in design mode: it cannot be made smaller in runtime.

The height of the status bar changes to accommodate the status bar font size with a two-pixel buffer above and below. In design mode you can change the width of a pane by dragging the handle that appears in the selected pane.

Every window instance contains the $statusbar property containing the window status bar in runtime. The $hasstatusbar property lets you hide and show the status bar at runtime. The $statusbar property also contains a group $panes containing the panes in the status bar numbered consecutively from the left. For example, pane 2 is $iwindows.WindowName.$statusbar.$panes.2. Each pane has width, text and appearance properties which you can set at runtime. For example

```
# declare item references to the panes
Set reference Pane1 to $cinst.$statusbar.$panes.1
Set reference Pane2 to $cinst.$statusbar.$panes.2
Set reference Pane3 to $cinst.$statusbar.$panes.3
Do $cwind.$statusbar.$panes.$remove(Pane3) ## removes the third pane
Do Pane2.$text.$assign('Click Save button to save your work')
Do Pane3.$sizing.$assign(kElastic)
Do Pane1.$hasborder.$assign(kFalse)
```

The $align property for a pane specifies whether to position the pane either after the previous left-hand pane or before the right-hand pane.

Progress Bars

In runtime, you can make a pane into a progress bar by enabling its **isprogress** property. If you want to view the properties of the status bar in runtime you can view it in the Notation Inspector under $iwindows. The properties of the status bar on a window instance are

- **min** and **max**
  sets the minimum and maximum value for the progress bar
- **isprogress**
  enables the pane as a progress bar
- **value**
  sets the current value on the progress bar

When $isprogress is set, $min and $max default to 0 and 100 respectively, but if you set them after setting $isprogress, your values will override the default settings. For example, to set $max for the second pane:

```
Do Pane2.$isprogress.$assign(kTrue)
Do Pane2.$max.$assign(200) ## default for $min is zero
```

The defaults for $min and $max are useful for percentages, for showing the percentage completed for an operation. The following method sets up a progress bar in the second pane and uses the default values for $min and $max:

```
Set reference Pane1 to $cinst.$statusbar.$panes.1
Set reference Pane2 to $cinst.$statusbar.$panes.2
Do Pane1.$hasborder.$assign(kFalse)
Do Pane1.$text.$assign("Doing Loop")
Do Pane2.$isprogress.$assign(kTrue)
# now set max if required e.g. Do Pane2.$max.$assign(maxvalue)
Do Pane2.$backcolor.$assign(kRed)
Calculate Pane2.$value as Pane2.$min ## resets value of pane2
Repeat
  Calculate Pane2.$value as Pane2.$value+1
Until Pane2.$value>=Pane2.$max
Do Pane1.$text.$assign("Ready")
Do Pane2.$isprogress.$assign(kFalse)
```

You can add an icon or picture from the USERPIC.DF1 data file or #ICONS to the progress bar, either from the Property Manager or with a command. For example, to have a show of hands as your bar add the line:

```
Do Pane2.$iconid.$assign(1072)
```

As a further refinement, you can add a '% Done' message to the progress bar using the current $value of the pane inside the loop.

```
Repeat
  Calculate Pane2.$value as Pane2.$value + 1
  Calculate Pane2.$text as con(rnd(((Pane2.$value/Pane2.$max)*100),0),"% Done")
Until Pane2.$value = Pane2.$max
Calculate Pane2.$text as "Finished!"
```

True Color Shared Pictures

Omnis Studio supports true color (24 bit) shared pictures. These are implemented using the free source for PNG and ZLIB.

PNG or "Portable Network Graphics", is a standard picture format, with a portable free source code implementation. ZLIB is a compression library, which also has a portable free source code implementation. You can find out more about PNG at: www.w3.org.

The $sharedpictures library preference has three values, which can be set using the following constants:

- **kSharedPicModeNone**
  do not use shared pictures.
- **kSharedPicMode256Color**
  use the 256 color shared pictures from earlier releases.
- **kSharedPicModeTrueColor**
  use true color shared pictures.

If you use shared pictures, you should use true color since this will probably result in smaller stored images, and more realistic colors.

The data file browser has a hierarchical menu from which you can choose no shared pictures, the old 256 color shared pictures, or true color shared pictures; this affects how pictures are converted when reorganizing.

Window picture fields and background pictures have the $cachepicture property, which defaults to kTrue. When kTrue, and a shared picture is displayed, Omnis keeps both the shared picture data, and a copy of the decompressed native OS picture - this uses more memory but results in faster drawing.

The reorganize data command has a new checkbox which only applies when the convert to shared option is checked. It indicates convert to true color shared pictures.

The Edit>>Paste From File dialog allows direct pasting of PNG files.

Note that conversion of images to true color loses color depth, unless you convert on a machine running in true color mode.

- **pictconvto**(*Character SrcFormat,Binary Src,Character DstFormat*)
  Converts the supplied binary data (with or without our internal header) from the supplied source format to the supplied destination format (on macOS, all bitmap data formats are supported, including PNG, TIFF, BMP, JPEG, and GIF). For example:

```
Do pictconvto("PNG",myPngData,"JPEG") Returns myJpegData
# This converts the PNG data in myPngData to JPEG
```

- **pictconvfrom**(*Character SrcFormat,Binary Src*)
  Converts from the raw data and the specified format to a picture value, which can be used in various Omnis fields. For example, the following code lets you read a JPEG file from disk and display it:

```
ReadFile ("C:\MYFILE.JPG") returns myJpegData
Do pictconvfrom("JPEG",myJpegData) Returns myJpegData
Redraw {JPEG_CONTROL}
```

- **pictformat**(*Binary Src*)
  Returns a character string which contains the format of the picture data supplied. For example:

```
Do pictformat(myJpegData)    ## will return "JPEG"
```

- **pictconvtypes()**
  Returns a single column list, which contains all the picture conversion types registered with Omnis. The values are: CS24, PNG, BMP, JPEG

The picture formats are as follows:

- **CS24**
  Omnis colour shared picture format (16 million colours) , including the internal Omnis header.

- **PNG**
  PNG format (Raw, as written on disk)

- **JPEG**
  JPEG format (Raw, as written on disk)

In the following example, the 24 bit colour shared images in the background picture objects of a remote form class are converted to PNG.

```
Set reference fref to $root.$libs.THIN.$remoteforms.rfBooks.$bobjs
Set reference curBOBJ to fref.$first
While curBOBJ
  If curBOBJ.$objtype=kBackpicture
    If pictformat(curBOBJ.$picture)="CS24"
      # 24 Bit picture
      Calculate picture as curBOBJ.$picture
      Calculate pictPNG as pictconvto("CS24",picture,"PNG")
      Calculate curBOBJ.$picture as pictPNG
    End If
  End If
  Set reference curBOBJ to fref.$next(curBOBJ)
End While
Save class {THIN.rfBooks}
```

Chapter 13—Unicode

Omnis Studio fully supports Unicode, which means you can expand the market for your Omnis applications by supporting the majority of world languages and the display of special characters, including scientific and mathematical symbols.

In previous versions of Omnis Studio, we provided a Unicode and non-Unicode version of the development kit, but from Omnis Studio 5 onwards only the Unicode compatible version was provided. The Unicode version of Omnis Studio is available for Windows, macOS, and Linux, and will allow you to localize your applications and deploy them to virtually any market, anywhere in the world.

You should also refer to the Localization chapter for information about localizing and deploying your desktop applications for non-English speaking markets.

What is Unicode?

Unicode provides a mechanism for representing characters or symbols used in many of the languages in the world, as well as scientific and technical environments. The Unicode standard is maintained by the Unicode Consortium (www.unicode.org) who set the standards for Unicode and promote its worldwide use. They define Unicode as: *"a character coding system designed to support the worldwide interchange, processing, and display of the written texts of the diverse languages and technical disciplines of the modern world."* In the context of client-server and Internet-based computing, Unicode allows the seamless exchange and processing of character data across different platforms, software products and programming environments.

The Unicode consortium provides information and resources concerning Unicode, including the standard definition and maintenance, character code tables, a locale identifier repository, and lists of Unicode enabled products. The last major version update of Unicode was version 9 which is capable of representing over 100,000 different characters, used in many different languages throughout the world. Many operating systems and software products have adopted Unicode, which is now universally accepted as the standard for character representation. For example, the latest versions of Windows and macOS, as well as all varieties of Linux, support Unicode. All web standards, such as the latest versions of HTML, XML, and JSON support Unicode, as well as the latest versions of Internet Explorer and all Mozilla-based browsers. In addition, SQL databases such as the most recent versions of Sybase, Oracle, and DB2 support Unicode.

Together with the display of multiple languages in Omnis, the use of Unicode encoding affects the sort order of dynamic data, for example, in list variables, as well as the querying and retrieval of data from Unicode compatible Server databases.

DAMs

The DAMs provided with Omnis Studio (from version 5.0 onwards) are able to function in Unicode or 8-bit compatibility mode. This means that after converting your existing libraries, it is possible to continue interacting with non-Unicode databases.

In 8-bit compatibility mode, all DAMs:

- Return non-Unicode character data types via the $createnames() and $coltext attributes
- Bind outgoing character variables using the database's non-Unicode data types
- Convert all data inside outgoing character bind variables to single-byte characters
- Define incoming character columns using the database's non-Unicode data types
- Convert all data inside incoming character bind variables from bytes into characters

Switching to 8-bit compatibility mode

To switch to 8-bit compatibility mode, there is a session property $unicode which should be set to kFalse from its default value of kTrue. This implementation allows multiple Unicode and 8-bit session objects to exist side by side if required.

Character Mapping

This section is applicable to session objects operating in 8-bit compatibility mode only.

When reading data from a server database, Omnis expects the character set to be the same as that used in an Omnis data file. The Omnis character set is based on the macOS extended character set, but is standard ASCII up to character code 127. Beyond this value, the data could be in any number of different formats depending on the client software that was used to enter the data.

When assigned, the $maptable session property identifies files containing translation tables for 8-bit character codes read into and sent out of Omnis. For example, suppose you are working with a database that stores EBCDIC characters. In order to accommodate this database, you should create an '.IN' map file that translates EBCDIC characters to ASCII characters when Omnis in reading server data and a matching '.OUT' file that reverses the process by converting ASCII to EBCDIC characters when Omnis is sending data to the server.

Under Windows and Linux, Omnis uses the same character set as under macOS, so in the general case, mixed platform Omnis applications should have no need for character mapping. However, if the data in a server table was created by another software package, running under Windows for example, the characters past ASCII code 127 would appear incorrect when read using Omnis. In this situation the $maptable property should be used to map the character set.

There are two kinds of character maps: IN and OUT files. IN files are used to translate characters coming from a server database into Omnis. OUT files are used to translate characters that travel from Omnis back to a server database.

The Character Map Editor

The Character map editor is accessed via the Add-On tools menu item and enables you to create character-mapping files. You can change a given character to another character by entering a numeric code for a new character. The column for the Server Character for both .IN and .OUT files may not actually represent what the character is on the server. This column is only provided as a guide. The Numeric value is the true representation in all cases.

To change a character, select a line in the list box and change the numeric code in the Server Code edit box. Once the change has been recorded, press the Update button to update the character map. You can increase/decrease the value in the Server Code edit box by pressing the button with the left and right arrows. Pressing the left arrow decreases the value, pressing the right arrow increases the value.

The File menu lets you create new character map files, save, save as, and so on. The Make Inverse Map option creates the inverse of the current map, that is, it creates an ".IN" file if the current file is an ".OUT" character map, and vice versa.

Using the Map Files

Establish the character mapping tables by setting the session property $maptable to the path of the two map files. Both files must have the same name but with the extensions .IN and .OUT and be located in the same folder. The $maptable property establishes both .IN and .OUT files at the same time. For example:

```
Do SessObj.$maptable.$assign('C:\Program Files\Omnis Software\ Charmaps\pubs') Returns #F
```

In this example, the two map files are called “pubs.in” and “pubs.out”.

The session property $charmap controls the mode of character mapping that is to be applied to the data. Set the character mapping mode using a command of the form:

```
Do SessObj.$charmap.$assign(pCharMap) Returns #F
```

The potential values for the character mapping mode parameter pCharMap are:

- **kSessionCharMapOmnis**
  Use the internal Omnis character set.
- **kSessionCharMapNative**
  This is the default and specifies that the client machine character set is to be used.
- **kSessionCharMapTable**
  Use the character mapping table specified in the $maptable property. If the $maptable property is not set and the application attempts to assign kSessionCharMapTable this fails.

If you wish to use the character mapping tables defined using the $maptable property, you must set $charmap to kSessionCharMapTable.

Interpreting 8-bit Data

This section is applicable to the MySQL, PostgreSQL and Openbase DAMs which interface with their respective client libraries using the UTF-8 encoding.

When operating in Unicode mode, it is possible to receive mixed 8-bit and Unicode data, since UTF-8 character codes 0x00 to 0x7F are identical to ASCII character codes.

Where this data was created using the non-Unicode version of Omnis however, it is possible that the data may contain ASCII extended characters. In this case, the Unicode DAM will encounter decoding errors, mistaking the extended characters as UTF-8 encoded bytes.

This issue was not a concern for the non-Unicode version of Omnis Studio since extended characters were always read and written as bytes, irrespective of the database encoding.

In order to avoid problems when upgrading to the Unicode version of Omnis Studio, it is advisable to convert tables containing ASCII extended characters to UTF-8. This process is simplified where the database character set is already set to UTF-8 (as is often the case with MySQL). All that is required is to read and update each row in the table and repeat this for all tables used by the application. In so doing, Omnis will convert the 8-bit data to Unicode and then write the converted Unicode data back to the database.

In order to facilitate this within the DAM, the session property $validateutf8 is provided. When set to kTrue (the default), any fetched character data is validated using the rules for UTF-8 encoding. Where a given text buffer fails validation, it is assumed to be non-Unicode data and is interpreted accordingly. When written back to the database, all character data will be converted to UTF-8. Such updates will result in frequently accessed records having their contents refreshed automatically.

By setting $validateutf8 to kFalse, validation is skipped and the DAM reverts to the previous behavior, in which case extended ASCII characters should be avoided.

Aside from the issue of UTF-8 encoded data, the DAMs provided with Studio 5.0 are able to retrieve non-Unicode data from non-Unicode database columns in either Unicode or 8-bit compatibility mode. The DAM knows the text capabilities of each character data type and assigns encoding values to each result column accordingly.

The difference in behavior when using 8-bit compatibility is that in compatibility mode, it is also possible to *write* data back to non-Unicode columns.

In Unicode mode, the DAM assumes that it will be writing to Unicode compatible data types and this will cause data insertion/encoding mismatch errors if the clientware tries to insert into non-Unicode database columns.

Character Mapping in Unicode Mode

Character mapping to and from the Omnis character set is also possible where session objects are operating in Unicode mode. This was previously removed from the Unicode DAMs since it provided compatibility between the various 8-bit character sets. Where Unicode DAMs encounter 8-bit data however, it is necessary to indicate the character set used by the data. For this reason the session $charmap property can be used to indicate that fetched 8-bit data uses either:

- **kSessionCharMapRoman**
  Use the Mac Roman character set to interpret the characters
- **kSessionCharMapLatin1**
  Use the Windows/Linux character set to interpret the characters

Fetching Data to a File

The $fetchtofile() method has the iEncoding parameter, as follows:

```
Do StatementObj.$fetchtofile(cFilename [,iRowCount=1] [,bAppend=kTrue] [,bColumnNames=kTrue] [,iEncoding=kU
```

where iEncoding is an optional parameter specifying the type of encoding to be used. It should be one of the Unicode type constants and defaults to kUniTypeUTF8. The corresponding Unicode Byte Order Marker (BOM) is written to the beginning of the file when the file is empty or when bAppend is set to kFalse.

Server Specific Programming

Certain DAMs, namely DAMORA8 and DAMODBC, also provide session properties which allow mixing of Unicode and 8-bit data when the DAM is operating in Unicode mode.

Oracle DAM

This section summarizes recent changes made to the Unicode Oracle Object DAM designed to enable insertion and retrieval of mixed ANSI and Unicode character types.

In the case of Oracle 8i and later, these data types are:

Type	Description
CHAR	Fixed single-byte character data, limited to 2000 bytes.
NCHAR	Fixed multi-byte character data, limited to 2000 bytes.(1000 UCS-2 encoded characters)
VARCHAR2	Varying length, single-byte character data, limited to 4000 bytes.
NVARCHAR2	Varying length, multi-byte character data, limited to 4000 bytes.(2000 UCS-2 encoded characters)
CLOB	Character Large Object- single-byte character data.
NCLOB	National Character Large Object- multi-byte character data.
LONG	Varying length, single-byte character data, limited to 2GB.Supported for backward compatibility only.

By default, the Unicode Oracle DAM maps all Omnis character data to the NVARCHAR2 and NCLOB data types, dependent on the field length of the Omnis bind variable. However, the Oracle DAM provides session properties which affect the Omnis to Oracle data type mappings:

- **$nationaltonvarchar**
  If set to kTrue, Character and National data types are treated differently when being inserted to VARCHAR2 / NVARCHAR2 columns. The *National* character subtype will be used with Unicode data, whilst the *Character* subtype will be reserved for non-Unicode data.
- **$nationaltonclob**
  If set to kTrue, large Character and National data types are treated differently when being inserted to CLOB / NCLOB columns. The onus is upon the developer not to put Unicode characters into Character subtypes when using these properties; otherwise data insertion/encoding mismatch errors will occur.
- **$maxvarchar2**
  Sets the byte limit above which Omnis character fields will be mapped to CLOB/NCLOB data types as opposed to VARCHAR2 / NVARCHAR2 columns. The maximum value is 4000 bytes.
- **$longchartoclob**
  If set to kTrue (the default), Omnis large character fields > $maxvarchar2 in byte length will be mapped to the CLOB/NCLOB data type. If set to kFalse, the LONG data type is used.

Reading Unicode and Non-Unicode Data

The Oracle DAM automatically detects the data type of retrieved character columns and converts the data accordingly. There is no need to modify any properties in order to retrieve mixed ANSI and/or Unicode Data.

ODBC DAM

The ODBC DAM provides the $nationaltowchar session property.

By default, Omnis Character and National fields are mapped to the SQL_WCHAR, SQL_WVARCHAR and SQL_WLONGVARCHAR data types. By setting $nationaltowchar to kTrue only National fields will be mapped to these types (to the equivalent server data types) and Character fields will be mapped to SQL_CHAR, SQL_VARCHAR and SQL_LONGVARCHAR as determined by the Omnis field length. Character fields mapped in this way are subject to data loss/truncation where such fields contain Unicode characters. When setting this property, please note that Unicode data types usually have precision limits half that of their corresponding ANSI data types. For example, this is 8000 for the SQL Server VARCHAR() data type but 4000 for NVARCHAR(). $nationaltowchar affects both the text returned by the $createnames() method and the binding of input parameters.

Character Normalization

Originally, Unicode was a 16-bit character set. It has subsequently been extended to include code point values up to and including U+10FFFF. It is not expected that it will be extended any further. Windows and macOS still represent Unicode character strings using arrays of Short (16-bit) integers. This is not a problem, because the UTF-16 standard allows code points U+10000 and greater to be represented by pairs of 16-bit values (each member of the pair occupies space in the 16-bit range that is not used for code points). This representation is referred to as a *surrogate pair*.

Internally Omnis uses UTF-32 to represent code points, that is, each code point occupies 32 bits, and the value of each code point is between 0 and U+10FFFF inclusive. This allows for straightforward processing of character strings, since every code point occupies the same space in memory.

Unicode allows a significant number of characters to be represented by more than one sequence of code points. For example, consider the letter E with circumflex and dot below, a character that occurs in Vietnamese (Ệ). This character has five possible representations in Unicode:

- U+0045 Latin capital letter E U+0302 combining circumflex accent U+0323 combining dot below
- U+0045 Latin capital letter E U+0323 combining dot below U+0302 combining circumflex accent
- U+00CA Latin capital letter E with circumflex U+0323 combining dot below
- U+1EB8 Latin capital letter E with dot below U+0302 combining circumflex accent
- U+1EC6 Latin capital letter E with circumflex and dot below

A character represented by more than one code point is referred to as a *composite character*. A character represented by a single code point is referred to as a *pre-composed character*.

As far as the end-user is concerned each of these representations usually needs to be treated identically. This leads to some interesting consequences for Omnis. These are discussed in the following sections. Note the term *end-user character* means the character that the end-user is working with – in the example above, the end-user character is Ệ.

Normalization of a Unicode character string converts the string into a standard, defined format. Once normalized, a Unicode character string has only one possible representation, thereby making it possible to compare it with other character strings, and produce results useful to the end-user. The Unicode standard recommends two forms of normalization. These are:

- Canonical decomposition, referred to as NFD:
  Pre-composed characters are replaced by their equivalent composite characters; Composite characters are replaced with a single fixed composite representation.
- Canonical decomposition followed by canonical composition, referred to as NFC:
  After carrying out NFD, all composite characters are replaced with their pre-composed equivalent, where one exists.

Omnis provides two functions to normalize character strings:

- **nfd(string)** carries out canonical decomposition on the string and returns the normalized string.
- **nfc(string)** carries out canonical decomposition followed by canonical composition on the string and returns the normalized string.

These functions are **not** available in client-side web client methods.

Comparing Text

Omnis uses two types of comparison for character strings:

- Comparison of the UTF-8 values of the strings. This is called *Character comparison*.
- Comparison according to the rules for the locale specified via the localization data file; prior to comparison, the input data is normalized. This is called *National comparison*. National comparison is more likely to produce results that the end-user would expect. Note that upper casing used in conjunction with national comparison may not have an effect, since sometimes the rules for the locale ignore the case of the characters.

The natcmp() function uses national comparison. Note that natcmp() is **not** available in client-side web client methods.

Omnis compares text for many different reasons, and in many different places. Key areas are:

- Sorting lists
- Searching lists
- Manipulating data file indexes
- Expressions, for example the test on an if statement

Omnis supports two types of character variable – character and national.

Sorting Lists

When using the character type, Omnis uses character comparison.

When using the national type, Omnis uses national comparison.

Searching Lists

When using the character type, Omnis uses character comparison.

Searches that directly use a character column of national type use national comparison.

Other searches, for example searches using a calculation, will behave as if they are operating on normal character data. However, you can use natcmp() as part of the calculation, in order to use national comparison.

Manipulating Data File Indexes

Indexes for national fields use national comparison.

Expressions

To ensure the correct behavior of expressions that test the value of character variables, you must either normalize their value first using nfc() or nfd(), or you must use the natcmp() function.

Drawing Text

Depending on the font and operating system you use, different representations of the same end-user character may not always be drawn in the same way. The same applies if you try to use strings that require surrogate pairs. Generally speaking, you will get the best results if you normalize the text using nfc(), as the issues generally occur with composite characters.

Entering Text

Wherever possible, you should use the nfc() normalization form for data that is to be edited. If composite characters are present in the data, multiple left or right arrow key presses are required to skip a composite character, and also clicking and selecting in the text will highlight an area which when copied to the clipboard might not exactly contain what appeared to be highlighted.

Omnis performs NFC normalization on character data pasted from the clipboard when running in the thick client (runtime); no normalization occurs when pasting characters into a remote form when using the web client.

Character Translation

The following functions allow you to translate a specified character in a string to its Unicode value and to allow the reverse.

- *unicode*(string,position[,returnhex])
  returns the Unicode value of the character at the specified position in the string. The first position in string is 1. If Boolean returnhex is true (default false) it returns a hex string representing the value, of the form 'U+h'.
- *unichr*(num1[,num2]...)
  returns a string formed by concatenating the supplied Unicode character codes. Each code is either a number or a string of the form 'U+h',where h is 1-6 characters representing a hexadecimal value.

These functions are available in client-side methods as well as the thick client, but will generate an error if used in the non-Unicode version of Omnis.

Unicode Clients

Locale Identifier

The *locale()*function returns the Locale Identifier (LCID) for the current client machine/operating system. As well as the language of the machine, the Locale Identifier specifies the decimal, thousand and list separators, currency values, units of measurement, date formats, and character sort order. The Locale is specified at the operating system level and is in the form language_country, where *language* is the ISO639 language name, and *country* is the ISO3166 country name. For example, the Locale for the UK is "en_GB". On macOS, there may be other information, such as a script code, between the language and country (this is because macOS uses ICU locales).

Unicode Data Handling

The *uniconv()* function allows you to translate Unicode character data from one type to another. The syntax is:

```
uniconv(srctype,src,dsttype,dst,bom,errtext)
```

The function converts *src*, and stores the result in *dst*. It returns zero for success, or a non-zero error code together with error text in *errtext*. *Src* and *dst* are either binary or character variables, depending on the values of the *srctype* and *dsttype*.

srctype and **dsttype** are one of the kUniType... constants (see below).

Bom is Boolean: if true, **dst** has a Unicode Byte Order Marker (BOM) if relevant for the destination type.

The kUniType... constants are as follows:

- **kUniTypeAuto**
  The source encoding is automatically detected from the conversion source; possible encodings are identified by the remaining kUniType... constants (allowed only for the source type).
- **kUniTypeUTF8**
  The data is stored in a binary variable and contains Unicode character data encoded using UTF-8
- **kUniTypeUTF16**
  The data is stored in a binary variable and contains Unicode character data encoded using UTF-16LE if the machine is little-endian, or UTF-16BE if the machine is big-endian. Useful when writing cross-platform code that interacts with the OS.
- **kUniTypeUTF16BE** or **kUniTypeUTF16LE**
  The data is stored in a binary variable and contains Unicode character data encoded using UTF-16BE (big-endian) or UTF-16LE (little-endian)
- **kUniTypeUTF32**
  The data is stored in a binary variable and contains Unicode character data encoded using UTF-32LE if the machine is little-endian, or UTF-32BE if the machine is big-endian. Useful when writing cross-platform code that interacts with the OS.
- **kUniTypeUTF32BE or kUniTypeUTF32LE**
  The data is stored in a binary variable and contains Unicode character data encoded using UTF-32BE (big-endian) or UTF-32LE (little-endian)

- **kUniTypeNativeCharacters**
  The data is stored in a binary variable and contains a stream of bytes, where each byte is a character in the Latin 1 character set for the machine (Ansi on Windows, MacRoman on macOS, ISO-8859-1 on Unix

- **kUniTypeCharacter**
  The data is stored in a character variable. Note – this constant has been moved since the last Unicode build, so you need to re-enter it in your code.

- **kUniTypeAnsi…**
  The data is stored in a binary variable, and contains character data where each byte is encoded using the specified ANSI code page. A range of constants are provided to cater for most world or regional languages, including Cyrillic, Greek, Hebrew, Arabic, Thai, and so on

- **kUniTypeISO8859…**
  The data is stored in a binary variable, and contains character data where each byte is encoded using the specified ISO 8859 code page.

There are two sys() functions to assist OEM conversion when using the uniconv() function.

- sys(218) modifies OEM conversion to map CR to CR and LF to LF.

- sys(219) reverts to the original mapping for the OEM code page.

Formfile

The $filereadencoding and $filewriteencoding properties have been changed. In previous versions of Omnis Studio, the Formfile component defined kFFEncoding… constants. *These constants should not now be used,* and you are advised to use the kUniType… constants to identify the file encoding. Formfile has been extended, so that you can use any of the kUniType… constants except kUniTypeCharacter for the $filereadencoding property, and any of the kUniType… constants except kUniTypeAuto and kUniTypeCharacter for the $filewriteencoding property.

In addition, there is also a kUniTypeBinary constant to identify files that are to be treated as raw binary data.

Code that uses the old kFFEncoding… constants should continue to work.

Fileops

The Fileops component has two methods, $readcharacter() and $writecharacter() which allow you to read and write Unicode character data from and to a file.

- $readcharacter(encoding,variable)
  reads all data from a file containing character data into *variable*; *encoding* is one of the kUniType… constants (listed above), identifying the encoding of the file.

- $writecharacter(encoding,variable)
  replaces the contents of the file with the character data stored in *variable*; *encoding* is one of the kUniType… constants, identifying the encoding of the file.

For $readcharacter, specify the encoding as any kUniType… constant except kUniTypeBinary and kUniTypeCharacter.

For $writecharacter, specify the encoding as any kUniType… constant except kUniTypeAuto, kUniTypeBinary and kUniTypeCharacter.

Note the $readcharacter() and $writecharacter() methods use the kUniType… constants and not the kFFEncoding… constants which should not now be used.

Mixing Char & Binary data

You cannot concatenate a Character variable to a Binary in the Unicode version of Omnis Studio. The correct method is to use $readfile to read the file into a Binary variable, and then parse the binary variable. Assigning Character to Binary and vice-versa is likely to cause problems, including data corruption, and should therefore be avoided.

Import/Export and Report File Encoding

There are a number of Omnis Prefences ($root.$prefs) that control the encoding of import text files, export files, and report data written to text files and the port. These are:

- **$importencoding**
  The encoding used for imported data when importing from port, or when the import file does not have a Unicode Byte Order Marker (BOM). Any of the kUniType... constants, except kUniTypeAuto, kUniTypeCharacter, kUniTypeBinary and the kUniTypeUTF32... values.
- **$exportencoding**
  The encoding used for exporting data and printing to port or text file. Any of the kUniType... constants, except kUniTypeAuto, kUniTypeCharacter and kUniTypeBinary.
- **$exportbom**
  If true, and the $exportencoding preference identifies a Unicode encoding, a Unicode BOM is output at the start of the output file.

The default value of the $importencoding and $exportencoding is kUniTypeUTF8, but you can set them using the **Preferences** option in the Options menu in the bottom-left corner of the Studio Browser. You can can also set the corresponding "importencoding" and "exportencoding" items in the "prefs" group in the Omnis configuration file (config.json) using the **Edit configuration** option in the same menu.

In a multi-threaded server, there is a separate value of each of these properties for each thread.

Omnis Data File Conversion

Omnis datafiles are supported for backwards compatibility only in legacy Omnis applications, and therefore they should not be used for new applications.

WARNING: YOU SHOULD MAKE A SECURE BACKUP OF YOUR OMNIS DATA FILES BEFORE CONVERTING THEM IN THE UNICODE VERSION OF OMNIS STUDIO (note all versions after Omnis Studio 5 are Unicode based and will convert Omnis datafiles to Unicode automatically).

When you access an Omnis data file you are asked to confirm that you want to convert the data. After you select Yes, Omnis displays a dialog which offers two types of conversion:

- **Full**
  whereby a full conversion of the Character based data in you Omnis data file takes place. The existing indexes are dropped and a new index of your data is built
- **Quick**
  whereby the indexes are dropped and rebuilt, but the Character data in you Omnis data file is not converted. This is OK for files containing only 7 bit data: Omnis does not check that the file contains only 7 bit data, so it's your responsibility to know whether or not it is safe to run this conversion process.

The full data file conversion mechanism converts the data in your Omnis data file and rebuilds the indexes. When data file conversion takes place, all data marked as Character is converted, including any characters >= 128. Note that in the case where character data is stored in a binary or external file, for example, text stored in a document file, conversion of this data *does not* take place.

Testing Data File Conversion

Omnis Studio can perform a full conversion of Omnis data files to Unicode, as described above. If this is the first time you have used the Unicode version of Omnis, we suggest that you *make a secure copy/backup of your Omnis data file* and convert one of the copies using the 'Full' conversion mechanism. We suggest that you check the results of the full conversion carefully, making sure that the Character data has converted successfully and that the indexes have been rebuilt successfully.

You may want to perform some regression tests on your application and data – you should normally do this with a new version of Studio, but when converting to Unicode Omnis, and converting your data files, you need to be especially sensitive to possible data file and indexing issues.

Data File Commands

The *Open data file* and *Prompt for data file* commands have an existing option called "Convert without user prompts". If this is checked, and the new "Full Unicode conversion" option is checked, no dialogs are displayed and your data is converted to Unicode using the Full conversion process.

Chapter 14—Localization

Note that the following section describes a localization method that can only be used for desktop applications, that is, apps containing window classes, so the following information is not relevant for some editions of Omnis Studio, including the Community Edition. For all new libraries that use the JavaScript Client, we urge you to use the localization method described in the Localization chapter in the Creating Web & Mobile Apps manual.*

If you are developing desktop applications for an international market, you may want to translate the text and labels in your libraries into another language or support multiple languages. Omnis provides tools for localization of your desktop applications using the **String Table** tab in the Catalog window and via the **String Table editor** in the **Add-ins** submenu of the **Tools** menu.

As well as translating your Omnis libraries, you can translate most of the text and strings that appear in the Omnis Runtime environment (the Omnis.exe) and the Omnis Web Client. You can localize various language dependent strings in the Omnis executable itself, such as the days of the week.

Right to Left Data Entry

In addition to the localization or translation functionality in Omnis Studio, you can force data entry fields to display their data from right to left, for example, to support text entry on Arabic machines. The $righttoleft property allows data in single- and multi-line edit fields to scroll from the right to the left. When this property is enabled for a multi-line field, the vertical scrollbar is displayed on the left of the field.

Localizing Your Libraries

Omnis contains an external package called 'StringTable' that contains a special String label object and a set of functions that allow you to dynamically change the language of text labels in your Omnis application.

You can use the String Table Editor in Omnis to create tables containing a matrix of strings or words for any number of languages. String tables and the functions in the StringTable package allow you to load the text for fields, buttons, and text labels when the window is opened, and then change the language of the text while the window is open.

The String Table Editor allows you to translate a whole list of strings (stored in the first column of your string table) into one or more languages automatically. For example, you could add English labels to a window or remote form, add the text for these labels to a string table, and translate all the labels to French, German, and Italian, with a single mouse click. The String Table Editor uses the translation tools provided by Google Translate™ to translate the text in your string tables automatically.

Using String Labels

First you need to create your window or form in your application that you wish to display in multiple languages. If you are enabling an existing window for dynamic label and text translation, you will need to replace existing text labels with the special 'String Label' external component from the 'StringTable' package. You can use standard fields and buttons.

Locating the StringLabel component

- **For window classes**
  the StringLabel object is located in the 'Background Components' group in the Component Store. The StringTable component is pre-loaded for window classes so you don't need to load it by right-clicking on the Component Store.
- **For remote forms (using the Web Client plug-in only)**
  the StringLabel object is located in the 'WEB Background Objects' group in the Component Store. If the object is not shown in the Component Store, right-click the Component Store, select the 'External Components…' option, and load the String Label object (FORMSTRG) from the 'Form Background Components' node in the list.

To create String labels

- Open your remote form and open the Component Store (press F3)
- Click on the ‘Background Components’ or ‘WEB Background Objects’ group in the Component Store toolbar and drag the String Label component onto your window or remote form
- Open the Property Manager (F6/Cmnd-6), click on the Custom tab, and enter a suitable label in the $rowid property; the row ID can be a number or string
- Create the other labels for your window or form using the String Label component, and assign unique labels in the $rowid property for each one

The string label object can display single or multiple lines of text.

It is also possible to translate the text for pushbuttons, check boxes and the contents of lists that may appear in your window or form. To identify these objects in the string table and your Omnis code you should make sure the objects have a suitable text string specified in the $name property. The text in the $name property of an object should be used as the row ID in the string table and also used in your code to reference the text.

Editing String Tables

Having created the string labels and buttons in your window or remote form, you need to create a String Table that contains all the alternative text for each object in the different languages you wish to support.

Note for existing users

In previous versions of Omnis Studio, the first column of a string table was labeled “ID” but in Studio 5 onwards it is called “STRINGID”. This is because “ID” is the ISO 639 code for Indonesia, and the Omnis localization features use ISO 639 codes to identify the language columns, so “ID” could no longer be used. Old string tables, where column one is called “ID” will usually still work, unless of course you are using ID to represent Indonesia in one of the language columns.

To create a string table

- Open the String Table Editor from the Tools>>Add-Ons menu
- Click on New to clear the table and create a new file; you can click on Save to save the new string table; the file should have the .stb file extension and can be located in the same folder as your library

Note: Your Omnis library cannot be called “STRINGTABLE”.

The first column in the new string table is called STRINGID: do not rename this column since this will contain the object rowIDs used to identify the strings in the table.

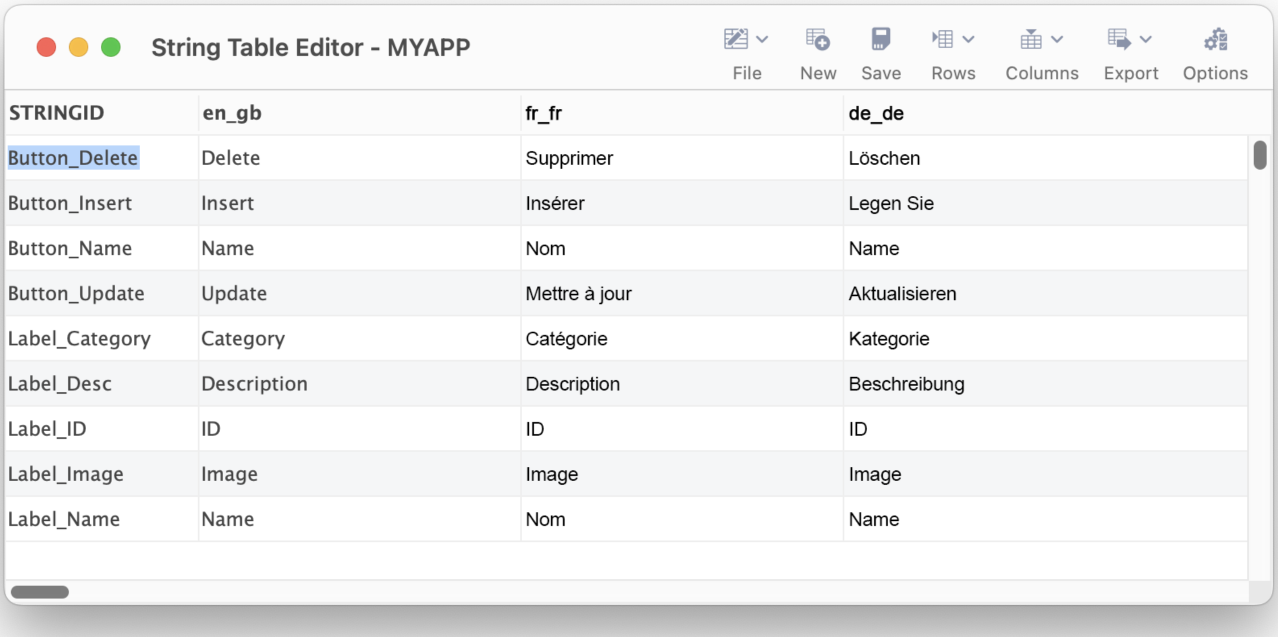
- The second column is *en_gb* by default for British English. If you want a different main language, use the Rename Column option from the Columns menu; name the column using a combination of the two-letter ISO 639 *language code* and the ISO 3166 *country code,* which you can select from the code lists, e.g. select *en_us* for American English (note language codes are lower case)
- Next you need to add a row for each string label or button you have used in your window(s) and/or remote form(s) in your application, entering the row IDs (the exact string) and button names you have used to identify these objects in the STRINGID column

You can find specific strings in Omnis using the **Find strings…** option (right-click on the string table name in the Catalog). You can drag a string from the Find window into the STRINGID column to enter the exact string to replace, and avoiding any mis-typing.

You can tab to the end of the row to create a new row or click the Add Row button and enter the Row ID for the next table entry; continue adding rows for each label or object you wish to translate.

- Add a column for each language you wish to support in your application, using the appropriate language and country codes to name each column, e.g. the third column could be named *fr_fr* to support French, the fourth column could *de_de* to support German, and so on (note lower case codes)
- Enter the translated string for each language

The following example shows a String Table for a remote form that has a number of string labels and pushbuttons. The English strings are in the second column, while the French and German strings were added to subsequent columns.



STRINGID	en_gb	fr_fr	de_de
Button_Delete	Delete	Supprimer	Löschen
Button_Insert	Insert	Insérer	Legen Sie
Button_Name	Name	Nom	Name
Button_Update	Update	Mettre à jour	Aktualisieren
Label_Category	Category	Catégorie	Kategorie
Label_Desc	Description	Description	Beschreibung
Label_ID	ID	ID	ID
Label_Image	Image	Image	Image
Label_Name	Name	Nom	Name

Figure 168:

- Then you need to **Save** and close the string table
- Finally, click on the background of your remote form, open the Property Manager (F6), and set the $stringtabledata property to the name of your string table; you can click on the property dropdown and navigate to your string table

Under certain circumstances you can use the STRINGID column in your string table as your “default” language, for example, if you have used the exact label names in the STRINGID column rather than some other rowID. In this case, you need to specify the Locale of your STRINGID column in the Translate dialog, using the two-letter ISO 639 language code, to identify its language.

Accessing String Tables via the Catalog

String Tables can be accessed and edited via the String Table tab in the Catalog (F9) window. Your own string tables for windows and remote forms will appear in the Catalog window when the window or form is the top design class.

String Table Functions

The StringTable package contains a number of functions that allow you to load string tables, load text items from a table, and replace the text in the windows in your application. The following methods are available:

- **$colcnt()**
  StringTable.$colcnt([cTableName]) returns the number of columns in string table cTableName, or an error code which is less than zero
- **$getcolumnname()**
  StringTable.$getcolumnname([cTableName]) returns the current column name for the string table specified by cTableName, or an error code which is less than zero
- **$getcolumnnumber()**
  StringTable.$getcolumnnumber([cTableName]) returns the current column number for the string table specified by cTableName, or an error code which is less than zero
- **$gettablelist()**
  StringTable.$gettablelist(lList) populates a single column lList with the loaded string table names; define lList to have a single character column before calling this method
- **$gettext()**
  StringTable.$gettext(cRowID) returns the text from the cell specified by cRowID for the current column, or an error code which is less than zero

- **$loadcolumn()**
  StringTable.$loadcolumn(cColumnNumber|Name,cTableName,cPathname) loads column cColumnNumber|Name from string table at cPathname into table cTableName. Returns kStringTableOK or an error code which is less than zero

- **$loadexistingtablefromlist()**
  StringTable.$loadexistingtablefromlist(cTableName,lList) replaces an existing string table with the content of a list. Returns kStringTableOK or an error code which is less than zero

- **$loadlistfromtable()**
  StringTable.$loadstringtable(cTableName,cPathname) loads string table from file cPathname, and gives it the name cTableName. Returns kStringTableOK or an error code which is less than zero

- **$loadstringtable()**
  StringTable.$loadstringtable(cTableName,cPathname) loads string table from file cPathname, and gives it the name cTableName. Returns kStringTableOK or an error code which is less than zero

- **$loadtablefromlist()**
  StringTable.$loadtablefromlist(cTableName,cPathname,lList) creates a string table from a list. Returns kStringTableOK or an error code which is less than zero

- **$removestringtable()**
  StringTable.$removestringtable(cPathname) deletes the string table file specified by cPathname. Returns kStringTableOK or an error code which is less than zero

- **$rowcnt()**
  StringTable.$rowcnt([cTableName]) returns the number of rows in string table cTableName, or an error code which is less than zero

- **$savestringtable()**
  StringTable.$savestringtable(cTableName) saves the string table specified by cTableName. Returns kStringTableOK or an error code which is less than zero

- **$setcolumn()**
  StringTable.$setcolumn(cColumnNumberOrName) sets the current column. Returns kStringTableOK or an error code which is less than zero

- **$unloadall()**
  unloads all string tables from memory.

- **$unloadstringtable()**
  StringTable.$unloadstringtable(cTableName) unloads the string table cTableName from memory. Returns kStringTableOK or an error code which is less than zero

The following functions apply to string tables for remote forms and are available for execution in server and client methods: when used on the server they will use the correct language for the task instance.

- **stgettext()**
  stgettext(id) returns the string with the specified *id* from a string table, or empty if the lookup fails. *id* can be prefixed with 'TABLENAME.', or must be an id for the string table of the current form. It is preferable to use stgettext() rather Stringtable.$gettext() since the latter one does not rely on the task instance.

- **stgetcol()**
  stgetcol(table) returns the name of the current column for string table *table* (for iOS Plug-in only)

- **stsetcol()**
  stsetcol(table,col) sets the current column for lookups from string table *table* to the column with name col and returns Boolean true for success (for iOS Plug-in only)

Programming String Tables

The following method loads a string table with the name 'Lang.stb' and sets the column containing the English text as the current column.

```
# custom method $loadStringTable
Calculate lvpath as sys(10)
Do FileOps.$splitpathname(lvpath,lvdrive,lvdirname,lvfilename,lvfileext)
Calculate lvpath as con(lvdrive,lvdirname,"Lang.stb")
```

```
Do StringTable.$loadStringTable(iTableName,lvpath) Returns ln
If ln
  OK message Error (Icon) {The Language String Table "Lang.stb" could not be loaded.}
  Quit method ln
End If

# Select the English Language
Do StringTable.$setColumn("English") Returns ln

Quit method ln
```

Having loaded the string table and specified the current column, the following method can be used to load the text or string values into the appropriate field labels, buttons, and lists in a data entry window. Note that the IDs for each object are stored in custom constants that are defined in the the Startup_task in the library.

```
# custom method $loadFields
# Define the button and group box descriptions using $getText
Do StringTable.$getText(kPrintButton) Returns lval
Do $cwind.$objs.PrintButton.$text.$assign(lval)
Do StringTable.$getText(kLanguageButton) Returns lval
Do $cwind.$objs.LanguageButton.$text.$assign(lval)
Do StringTable.$getText(kEmpTitle) Returns lval
Do $cwind.$objs.stEntry_1022.$text.$assign(lval)
```

The next section of the method defines the dropdown lists for Sex (male/female) and Marital Status by loading the relevant entries from the current string table.

```
Set current list iSex
Define list {iField}
Do StringTable.$getText(kMale) Returns iField
Add line to list
Do StringTable.$getText(kFemale) Returns iField
Add line to list
Calculate iSex.$line as 1
Set current list iStatus
Define list {iField}
Do StringTable.$getText(kMarried) Returns iField
Add line to list
Do StringTable.$getText(kSingle) Returns iField
Add line to list
Calculate iStatus.$line as 1
```

Within your application you need to provide some way for the user to select and change the language setting. This could be done using a separate window containing a list of available languages and a button to set the selected language. The following method gets all language column names from the Lang String Table and puts them into an Omnis list.

```
# custom method $loadList
Set current list iLang
Define list {iLangField}

# Save the current column number
Do StringTable.$getColumnNumber() Returns ln

# Build a list of all column names in the String Table.
Do StringTable.$colCnt() Returns maxc

# Start from column 2 as column 1 is reserved for String Table IDs
For lcount from 2 to maxc step 1
  Do StringTable.$setcolumn(lcount)
  Do StringTable.$getcolumnname() Returns iLangField
  Add line to list
End For
Do StringTable.$setColumn(ln)
Calculate iLang.$line as pLine
```

When the user selects a language from the list, they should click a button with the following method which changes the language of the text on itself and the data entry window.

```
On evClick ## Event Parameters - pRow( Itemreference )
  Set current list iLang
  Calculate lval as ((iLang.$line)+1)
  # need to offset by 1 since col1 in string table has the rowID
  Do StringTable.$setColumn(lval)
  Do StringTable.$getText(kLangTitle) Returns lval
  Do $cwind.$objs.stLang_1016.$text.$assign(lval)
  Do StringTable.$getText(kSetLanguage) Returns lval
  Do $cwind.$objs.SetLang.$text.$assign(lval)
  Do $root.$iwindows.stEntry.$loadFields
  # this line runs the $loadFields method again (see above) to change the objects in the data entry window
  Do method $loadList (iLang.$line)
  # this line reloads the language list in the new language
  Do method $translateList
  Do method $redrawAll
```

The following method translates the language list depending on the Language selected. The code uses the String Table column headings to look up corresponding IDs from within the table itself.

```
# custom method $translateList
Set current list iLang
Calculate ln as iLang.$line
Calculate lrowcnt as iLang.$linecount
For lcount from 1 to lrowcnt step 1
  Do StringTable.$getText(iLang.[lcount].iLangField) Returns iLang.[lcount].iLangField
  Calculate iLang.$line as lcount
End For
Calculate iLang.$line as ln
```

Finally you need a method to redraw all the fields and text labels on any open windows or forms.

```
# custom method $redrawAll
Do $iwindows.$first() Returns ref
While ref
  Do StringTable.$redraw(ref.$hwnd)
  Do $iwindows.$next(ref) Returns ref
End While
```

Localizing Remote Forms

The String Label object is available for remote forms, and together with string tables, allows you to localize your applications running in the Omnis Web Client. See the previous section for details about how to create string labels and string tables; the technique is the same for remote forms.

In Omnis Studio 5, remote form classes have a new property called $stringtabledata. The contents of this property is the data, in list format, from a standard Omnis string table. To populate $stringtabledata in the IDE, you can click on the droplist button on the property in the Property Manager, and select a string table file. When you press OK, Omnis takes a copy of the string table data and stores it in the class. If you cancel the dialog, Omnis asks if you wish to clear the data from the class, which is a convenient way to clear the contents of $stringtabledata.

When the web client creates an instance of the remote form class, on the client, it automatically loads the string table data as a client-side string table, and sets the string table name to the remote form class name.

The column names in the string table must be:

- STRINGID for column 1 – this is the standard for string tables, and the name cannot be changed.
- A two character ISO 639 language code for the second and subsequent columns; for example, en for English, de for German, fr for French.

When the client loads a new string table (because a remote form is being instantiated in some way), it uses the operating system locale of the client to locate the string table column to use for lookups. You can override this behavior by setting the remote task instance property $stringtablelocale to a two character ISO 639 language code to use instead of the client operating system locale, but you can only do this in $construct of the remote task. If there is no column in the string table for the desired locale (specified in $stringtablelocale), lookups default to using column 2.

The $rowid property of a remote form string label object can refer to one of the Web Client string tables. If $rowid is not prefixed with a table name (remote form name), then the string must be in the string table for the remote form containing the object.

Any character string property of a remote form control can be specified as $st.id (note that $st is not notation; it is just a special prefix recognized by the client). This tells the client to lookup id in the string tables for the client, and set the property value to the result of the lookup. The $rowid rules regarding the presence of a table name prefix also apply in this case. Note that when you get a property set in this way, the result is the result of the lookup, not $st.id.

There is a new remote form property called $stringtabledesignform which allows you to access the string table in the Catalog (F9) windows while designing the remote form.

There are three new "Client String Table" functions, available for execution in client methods only:

- stgettext(id)
  returns the string with the specified *id* from a string table,or empty if the lookup fails. *id* can be prefixed with 'TABLENAME.',or must be an id for the string table of the current form.
- stgetcol(table)
  returns the name of the current column for string table *table*
- stsetcol(table,col)
  sets the current column for lookups from string table *table* to the column with name *col* and returns Boolean true for success.

Remote tasks: $construct()

There is also a new column, "ClientLocale", in the row variable parameter passed to $construct() of the remote task. This contains the locale for the client, as returned by the locale() function; the first two characters are the ISO 639 language code of the client.

Remote Menu Lines

You can set the text for a remote menu line to $st.id. The lookup occurs when the menu is built on the client, before any event processing for the menu.

Tooltips

You can use the $st. lookup notation to lookup text from a string table to fill out the $tooltip property for a field.

Decimal point character

The decimal point character for display and entry on the JavaScript Client is the character for the current locale (either set by the current locale of the browser, or overridden using $ctask.$stringtablelocale).

Multi-threaded Language Separators

The main Omnis App Server thread and any other server thread(s) can now have their own values for decimal point, thousand separator, and import dp, which are stored in the $separators Omnis root preference. This allows you to support multiple languages in a single app running on the Omnis App Server.

If you call $separators from an Omnis App Server thread, the new values for the function parameter and import separators are ignored – you can only set these two separators when running in the main thread.

In addition, once you have started the server with the *Start server* command, subsequently changing the language only affects the decimal point, thousands separator and import decimal place for the main thread.

Localizing Omnis

In order to provide a completely foreign version of your Omnis application, you may need to translate various resources that appear in Omnis itself (rather than strings that appear in your libraries, as described in the previous section). This applies equally to applications that run on the desktop using the Omnis executable (Runtime), and applications that run on the Internet using the Omnis Web Client: you can translate string resources in the Omnis runtime and in the web client. This method may also be useful if you wish to replace any references to "Omnis" in the Runtime with the name of your own product, e.g. the 'Hide/Quit Omnis' option in the macOS version of Omnis.

Omnis Studio 5 introduces two new String Tables that allow you to translate:

- Various built-in string resources in the Omnis executable, and
- String resources in the Web Client.

The new String Tables can be accessed via the String Table tab in the Catalog (F9) window and are edited using the String Table editor.

In addition, the Localization Library (omnisloc.lbs) is still available in the Local folder for you to translate further string resources in the Omnis executable.

Localizing Built-in Client Resources

The built-in resources comprise the string resources, and the strings in dialogs, in the Omnis core, externals and external components. Omnis has two new string tables used for the translation of built-in and web client resources. The new tables are located in the Local folder of the main Omnis tree, and are:

- **studio.stb**
  containing translations of text strings for the fat client, comprising dialogs, components, and so on
- **client.stb**
  containing translations of text strings for the Omnis Web Client

These string tables can be edited using the String Table Editor. However, there are some special rules regarding their structure: the STRINGID column is the exact value of the built-in resource text and cannot be changed; the other columns contain the translations of the built-in resource text, and they are named using a 2 character ISO 639 language code, in lower case.

Usually, there is no column for "en" (English), because the built-in resources are English. If you are using a mixture of languages in the built-in resources, then you can add a column for "en". As a result, you will have cases where the STRINGID and language column for the built-in resource would have the same value. To avoid duplicating the value in the language column, you can enter "*" for that column, meaning the string does not need translating from its STRINGID.

When reading a string resource, Omnis looks up the string value in the string table, and if present, it replaces the string value with the translation. If no translation is available, the string remains unchanged. The string value lookup is case-sensitive, allowing the translation to contain the correct case for the translated text.

Omnis loads the studio.stb string table when it starts up, and when $root.$prefs.$language changes, and the appropriate language version of the resources is loaded. The column used for translations is the ISO 639 language component of the locale stored in the current language record. This is also stored in a small text file (locale.txt) in the Local folder of the Omnis tree, to cater for the fact that the string table is loaded sooner in the life of Studio than the localization data.

When a web-based client connects, Omnis sends client.stb to the client; this is handled via the cache, so it will not always be sent. The client loads client.stb, and uses the column corresponding to the client locale to obtain translations (the client locale can be specifically set using $cinst.$stringtablelocale for the remote task instance, or it can be allowed to default to the locale of the client machine).

Editing the Built-in Client Resources

You can open and edit the studio.stb and client.stb string tables via the Omnis Catalog (F9) window. You can click on the String Table tab in the Catalog, and right-click on the string table you wish to edit.

Representing carriage return, linefeed, and tab

You can represent cr (carriage return), lf (linefeed), and tab in strings in the studio.stb file using the escape sequences: <cr>, <lf> and <tab>. The Find Strings dialog automatically generates these escape sequences.

Local Language

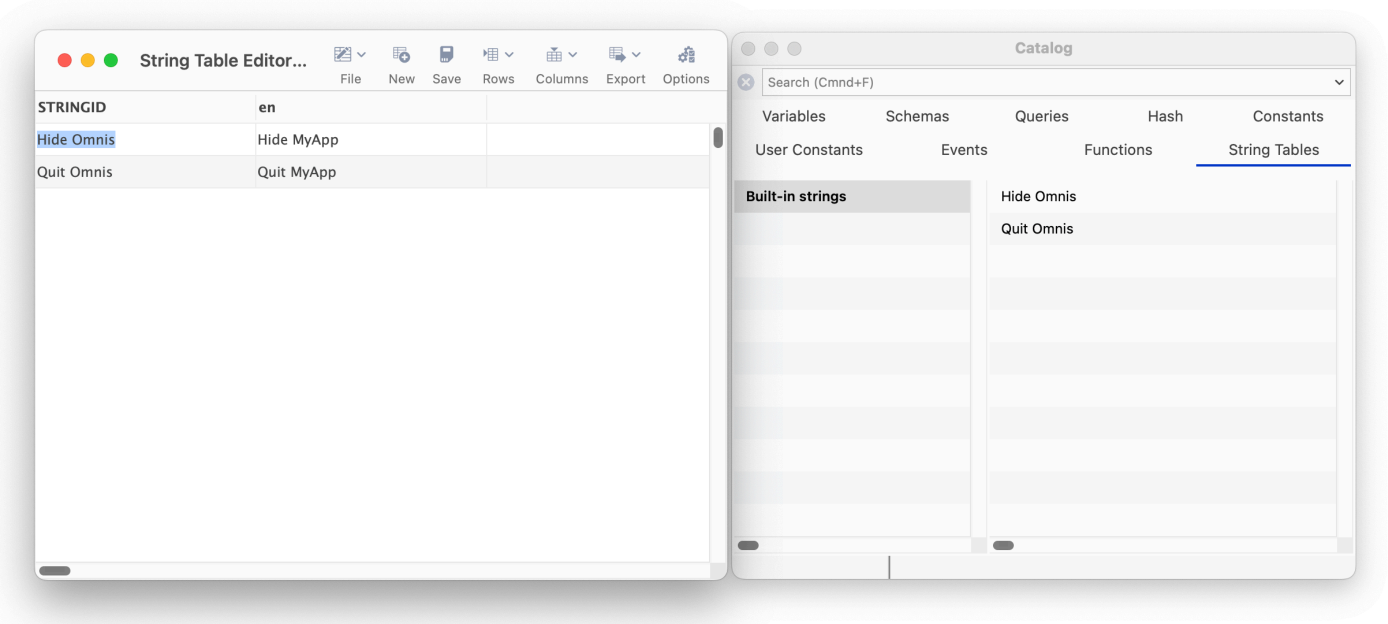
The locale() function for the fat client now has an optional Boolean parameter, which when passed as kTrue, causes it to return the locale field value for the current $root.$prefs.$language.

Changing the System Menu Options in macOS

You can change the **Hide Omnis** and **Quit Omnis** options in the Omnis Studio runtime on macOS by adding strings to the Studio String Table (studio.stb). In addition, you can localize items in the **Preferences** and **Services** menus.

If you have renamed the Omnis app package on macOS, to match your product name, you may also like to change the Hide Omnis and Quit Omnis options in the main application menu to reflect your product name. To do this:

- Open the Omnis **Catalog** (press F9) in Omnis Studio for macOS
- Select the **String Tables** tab, right-click on the 'Built-in strings' table (studio.stb) and select the **Edit** option
- Enter the Strings **Hide Omnis** and **Quit Omnis** in the STRINGID column and enter the alternative strings for each option in the 'en' column.



STRINGID	en
Hide Omnis	Hide MyApp
Quit Omnis	Quit MyApp

Figure 169:

You can find specific strings in Omnis Studio using the **Find** strings... option (right-click on the string table name in the Catalog). You can drag a string from the Find window into the STRINGID column to enter the exact string to replace, and avoiding any mis-typing.

Existing users should note that the Translate button has been removed from the String Table editor since automatic translation is no longer supported.

Localizing the Omnis Runtime

Developers and distributors in non-English speaking countries may need to localize the Omnis program (runtime) itself. You can localize the following Omnis internal items:

- The names of the days of the week
- The names of the months of the year
- Separator characters
- The text for Yes/No, OK/Cancel, True/False, Am/Pm and On/Off
- The national sort ordering
- The date ordinal suffixes

Storage of Localization Data

All libraries share the same set of data, stored in an Omnis data file or *localization database* called OmnisLOC.DF1, located in the Omnis local folder.

OmnisLOC contains a data slot for configuration data; each record in that slot contains a complete set of data corresponding to a particular language. It also contains a data slot with a single record, which identifies the current language, that is, the current set of configuration data.

Overriding the Language

You can override the current language set in the localisation data file by setting an item in the Omnis config.json file, which can be used with the Linux headless server. The entry is called "language" and is located in the "defaults" section. It defaults to empty, which means the setting in omnisloc.df1 will be used. To override the language setting in omnisloc.df1, you can add the name of a language in omnisloc.df1 to the language item.

The Localization Data

The following items are stored for each language.

Days of Week

This comprises 2 strings for each of the 7 days of the week, allowing for a full name such as Wednesday, and an abbreviated name such as Wed.

If a day of the week item is empty, Omnis will read its value from the operating system.

Months of Year

This comprises 2 strings for each of the 12 months of the year, allowing for a full name such as August, and an abbreviated name such as Aug.

If a month item is empty, Omnis will read its value from the operating system.

Separators

These comprise the following:

- The decimal point used for all numeric fields. If this item is the character '0' (zero), Omnis will read the decimal point character from the operating system.
- The thousands separator used for numbers. If this item is the character '0' (zero), Omnis will read the thousands separator character from the operating system.
- The function parameter separator
- The decimal point used when importing data
- The field separator used when importing
- The sequence used for quoting names in the notation

Standard Text Strings

These comprise the strings for Yes and No, OK and Cancel, True and False, AM and PM, and On and Off. If the value of the AM or PM text is the single character '0' (zero), Omnis will read the text from the operating system.

Locale: National Sort Ordering

The ISO 639 language code in the Locale field is used to define the sort ordering for National fields. The "Use Locale For Defaulted Items" check box below the Locale field determines the locale used for language items that have been left empty, so that they get a default value from the system:

- If the Use Locale check box is checked, default values come from the locale stored in the language record.
- If it is not checked, default values come from the operating system default locale on the client machine.

Date Ordinal Suffixes

You can localize the date ordinal suffixes which are the strings that can be appended to a day number to result in language specific days such as 1st, 2nd, 3rd, 4th, etc. You can edit the date ordinal suffixes field on the Text Strings tab of the localization library language settings window.

For English, the value of the date ordinal suffixes field is:

```
th$1st$2nd$3rd$21st$22nd$23rd$31st$
```

If you leave the field empty, then no suffix is applied, otherwise, the string before the first $ is the default suffix, and the remaining $-separated strings are a day number followed by its suffix. There must be a trailing $.

To see the suffix used for a particular date you can use the function call dat(date,'d').

The natcmp() function

The natcmp() function lets you compare two values using the national sort ordering.

```
natcmp (value1, value2)
```

Omnis converts both values to strings before doing the comparison.

Omnis uses the same rules for comparing the strings as it does for normal strings, except that it performs the comparison using the national sort ordering.

natcmp() returns 0 if the strings are equal, 1 if value1 > value2, and -1 if value 1 < value2.

User Interface

The Omnis preferences accessed from the IDE Tools>>Options menu line let you assign a new language from the dropdown list in the $language property. The current language is shown in the **$**language property. The language must already be defined in the localization data file.

The new language assigned in the $language property applies straight away; Omnis does not need to be restarted. However if the localization database is shared by several users, then the new language setting only affects them when they have restarted.

An Omnis library, OmnisLOC.LBS is provided that lets you create and edit language information. To use it:

- Take a backup of the OmnisLOC.DF1; you may prefer to work on the backup copy rather than the live copy, in which case you should make a working copy as well as a backup copy
- Open the OmnisLOC.LBS library, found in the Local folder in the main Omnis folder. You are prompted for the location of the localization data file and a localization menu is installed on the IDE menu bar, to the right of the Tools menu
- Select Current Language to display the language in use
- Select Language Records to create a new set of language information, or to edit an existing one. This displays a dialog containing a set of tabbed panes and the standard Omnis Insert, Edit, Find, Next and Previous buttons

You use the Next and Previous buttons to move through the records in the data file, the Find button to locate a particular language record, and Edit to modify data already present in the data file.

Two Insert buttons are available. Insert lets you create a brand new record, while Insert CV lets you make a copy of an existing language record and edit that. This is particularly useful for cases where there are only minimal differences between two language records. To use Insert CV:

- Display the language record you want to copy
- Click on Insert CV

A new record is created. Remember to edit the language name as well as the specific internal data.

- When all the data is input, click on OK to store it and close the library
- If you were working on a copy of the data file, move it back to the local folder
- Close the OmnisLOC library

Any fields that are left blank will default to a single space. Some of the fields on the General tab are limited in terms of which characters can be used; for example trying to define a letter as a decimal separator is not allowed, and will generate an error message.

Setting the Locale

In the localization library (omnisloc.lbs), the sort order field is now labeled Locale. There is a new check box below the Locale field, "Use Locale For Defaulted Items". This determines the locale used for language items that have been left empty, so that they get a default value from the system:

- If the Use Locale check box is checked, default values come from the locale stored in the language record.
- If it is not checked, default values come from the operating system default locale.

Notation

There is no requirement to manipulate localization data at runtime, so the localization notation is minimal.

- **$root.$prefs.$language**
  returns the name of the language Omnis is currently using; you can assign the name of the new language (note this takes immediate effect in your copy of Omnis, but any shared users will need to restart)
- **$hascurrlangnationalsortorder**
  a property of a data file, for example
  $root.$datas.DataFile.$hascurrlangnationalsortorder
  if true the sort order matches that for the current language, and false otherwise

Every data file stores its national sort order. When you create a new data file, Omnis stores the national sort order for the current language in the data file.

$hascurrlangnationalsortorder is assignable, but you cannot set it to kFalse only kTrue. When set to kTrue Omnis drops all of the indexes from the data file, changes the sort order to that for the current language, and rebuilds all of the indexes.

Chapter 15—Version Control

The Omnis VCS is not available in some editions of Omnis Studio, including the Community Edition, so in this case it will not appear in the Studio Browser.

This chapter describes how you can use the **Omnis VCS** (Version Control System) to control Omnis application development in a team environment. In such an environment, with several people working on the same application at the same time, perhaps in different locations, you need to ensure that only one person can change a particular component at a time and that any modifications are not lost or overwritten. The Omnis VCS is built into the Studio Browser and therefore provides tight integration with the whole Omnis IDE: specifically, the Omnis VCS allows you to control the development of your Omnis applications (classes and libraries), or any other project involving many different files such as web or Intranet applications, and it allows you to build your Omnis application for deployment to your customers or organization.

The Omnis VCS lets you revise Omnis library files and other application components systematically. Apart from the Omnis libraries and classes within those libraries, you may have your own externals, text files, web components, Html files, and so on, that are all necessary to the running of the application. The Omnis VCS can handle all these types of files and components. The repository for the VCS would typically be a SQL database within your own domain, or on a remote, cloud-based server.

In this chapter, the term *component* is used to refer to all types of files and different components stored on disk. With regards to the Omnis VCS, a *non-Omnis component* is any disk file or external component other than an Omnis class.

Converting pre-Studio 10 VCS repositories

When using the VCS in Studio 10, you must create a new VCS repository due to the changes in the Omnis language syntax resulting from the new code editor. You are advised to open and convert your library, then check the conversion logs to look at any possible issues in your code (any conversion issues are shown in the Find and Replace log, and written to a log file in the 'conversion' folder in the logs folder). Then when you are satisfied your library and its code are OK, you can check the classes in your library into a new VCS repository created in Studio 10.

Pre-Studio 5 VCS Repositories

There is no change in the structure of VCS repositories from Omnis Studio version 5 to version 6, but if you are upgrading to Omnis Studio 6/8 or above from a version prior to version 5, you cannot use your old VCS repositories. In Omnis Studio version 5 there were a number of significant changes to the structure of the VCS which means repositories created in versions of Omnis Studio prior to version 5 will not work with Omnis Studio 6/8 or above.

Existing VCS repositories created in versions of Omnis Studio prior to version 5 must therefore be re-created in Omnis Studio 6/8 or above to ensure that they are in the new format. To do this, you must do a build of your existing project (or projects) using your old version of Omnis Studio, create a new VCS repository in Omnis Studio 6 or above, and check in your project(s) and components into the new Studio 6/8 repository. You will also need to setup all user accounts and preferences in the new repository.

Project Branching

Project branching was deprecated in Studio 8.1 and you are no longer recommended to use branching in VCS projects. For backwards compatibility in branched projects only, support for branching is available by setting the **enableBranching** item in the 'vcs' section of the config.json file to True.

In addition, sys(236) returns true if VCS branching is enabled. If sys(236) returns false on a branched project, the VCS will display the default branch data.

Overview

To place Omnis libraries under version control you check them into the Omnis VCS from the Libraries tree in the Studio Browser, or for non-Omnis components from the File Browser in the VCS itself. All the components you check into the VCS are kept in a *project*. The VCS stores each project in its own *repository* database, which can be a server database within your own domain or a remote location ***To use the VCS you must first create a database session to store your project: see below.***

The Omnis VCS provides all the functionality to set up, manage, and use version control, including:

- Creating a database session and VCS repository
- Checking in Omnis libraries and other non-Omnis components
- Creating a project
- Managing and supervising users
- Building projects and libraries for distribution
- Managing projects and granting user privileges
- Setting VCS options

The Omnis VCS is permanently available in the Studio Browser (unless you're using the Community edition). The **IDE Options** option in the **Hub** in the Studio Browser lets you hide or show the VCS; if for some reason the VCS is not visible in the Studio Browser you can enable it in the Hub.

To use the VCS, click on the VCS option in the Studio Browser. The Session Manager and the Open Session options are shown in the Studio Browser which allow you to setup your VCS repository.

The VCS tree displays all your open projects and works in a very similar way to the Libraries and SQL Browser tree. The VCS context menu lets you open and close sessions, and perform user administration functions. When you select a project in the VCS, the Browser shows the contents of that project.

You can create a VCS repository in an Omnis data file, but this is only provided for backwards compatibility with legacy applications, therefore this option should not be used for new applications or projects.

Setting up a Project

This section is for the lead developer or manager in the development team and describes how you can setup a VCS repository and create a new project containing all your application components.

To use the VCS, you need to connect to a database via a session, set up the Supervisor user, and create a VCS repository. You can open a VCS session using the SQL Browser, but you should use the VCS session browser for convenience.

You can create a VCS repository on a server database such as PostgreSQL, available in all versions of Omnis Studio, or using one of the many other leading databases such as Oracle, Sybase, MySQL, and so on. You can access a repository using a native DAM for the chosen database, or you can use ODBC.

Once you have connected to your database, the VCS operates in the same way, regardless of the database and the location the repository is stored in. To setup a new project, you need to:

- Define a new database session, and create a new repository
- Sign in as Supervisor (default password is password), then create and define users
- Check in your libraries, classes and other components (the check in process creates a new project for you) and assign access privileges to components for individual users
- You also need to tell the members of your development team their usernames and passwords, and provide them access to the VCS repository

Sybase Repositories

If you are using Sybase for your VCS repository, you must make sure the transaction log has enough capacity to handle your transactions. See the Sybase documentation for details on the transaction log. You can use the command

```
dbcc checktable (syslogs)
```

while you are running the VCS against your repository to check the status of the transaction log. You must also set the option "select into/bulkcopy" to false with the following remote procedure call

```
sp_dboption <db>, "select into/bulkcopy", false
```

where <db> is your Sybase database name. The VCS issues an error message and aborts logon if the target database has the "select into/bulkcopy" option set.

Creating a session

To create a new VCS session, you can either use the default session, called 'VCS_Session', or create a new one.

To create a VCS session

- Select **VCS** in the Studio Browser and click on the **Session Manager** option
- Click on the **New Session** option

The Modify Session dialog opens which allows you to enter the details of the new session.

To duplicate a VCS session

If you have a session that is a suitable template for a new one

- Select the existing session and select **Duplicate** from the Sessions menu

In either case, a new session appears in the VCS Session Browser. You can now click on your session and modify it.

To modify a VCS session

- Select the session in the Browser and click the **Modify Session** option

or

- Double-click on a session to modify it

The Modify Session dialog lets you modify the details of the selected session; it is identical to the session definition dialog in the SQL Browser.

Note that to make a session usable with the VCS, you must select VCS from the Session type dropdown list.

The information you need to supply depends on the database you want to use, but would normally include hostname, username and password, plus you need to select the correct DAM for your chosen database, which will depend on your version of Omnis Studio.

- When you have modified the session, click on OK to close the Modify Session dialog

To open an existing VCS session

- Select the **VCS** option in the Studio Browser, click on the **Open Session** option and select the session from the list of VCS sessions
- Enter the Username and Password to open the session

Logon at Startup

You can specify that a VCS session logs on automatically when Omnis starts. On the Modify session dialog there is a 'VCS' tab that allows you to enter the VCS username and password which will be used to log onto the VCS automatically when you start Omnis. If this is an existing session, there is a button which allows you to verify the username/password are correct for the session. Then on the Session Definition tab you need to enable the 'Logon at Startup' option.

Time Settings

The VCS stores times in UTC (Co-Ordinated Universal Time) but displays times using your local time zone, which is set automatically.

Signing in to the VCS for the first time

When you create and/or open a VCS session, you must log on to your database and sign into the VCS as the Supervisor user. The VCS logs on to your database and checks for a repository. When you open a VCS session for the first time, a VCS repository will not be found, and the VCS prompts you to add certain VCS resources or tables. If you click on the No button, the log on process is aborted. If you agree, certain VCS tables are installed so the repository is available for use, and a user called Supervisor, with password "password", is set up automatically.

Once logged on to the repository, the Supervisor has access to an option from the VCS context menu titled "Remove Repository", this option will remove all the tables created by the VCS.

To sign into the VCS for the first time

When you log on to your VCS repository, the Sign in window appears. When you logon for the first time you need to sign in as Supervisor.

- Enter the user name "Supervisor" and the password "password"

Both the username and password are *case sensitive*, so make sure both words are in the correct case, otherwise you will not be able to logon.

- Click on OK

To ensure a secure system, you should change the Supervisor name and password. You can do this later, but to do it now

- Select the VCS option in the Studio Browser and click on the User Admin option

The User Administration window lets you add and remove users, and initially displays only one defined user, Supervisor. The columns in the list show details about each user, in this case the Supervisor, including the user's name, password, phone extension, department, and status.

- Change the "Supervisor" username to your name and enter a new password
- Enter any more information you wish to store, and click on the Finished button

As a Supervisor, you can allow other users to have Supervisor status so they can create and delete users as well. You will also need to set up the regular users who will check components in and out, and set up preferences.

Adding and Removing Users

User administration involves managing the users of a project and assigning them privileges for the components in the project. Before developers can start using the project, you need to add each one as a project user and grant the right privileges for the components they need to change.

Once you sign into the project as Supervisor, you can add, alter and remove users of the project. The User Administration window displays a list of existing users. The Supervisor can add any number of additional users and assign the following privileges which determine the extent to which a user can access components:

- **Observer** can see components only, therefore user cannot check components out or in
- **Participant** user can check components out and in, but cannot perform user admin
- **Supervisor** can do everything, including adding or deleting users and changing user details, and checking components out and in

Only a Supervisor can see a user’s password. Only the first Supervisor user can grant Supervisor status to other users.

To add a new user

- Click on the User Admin option and click on the Add User button, or you can right-click on the list of users and select Add User
- Add the new user information, including the username and password; you can add the phone extension number and department name for the user as well
- Change the type of user as appropriate; a new user is set to Participant by default
- Click on the Finished button and you are prompted to Save the Changes

To change an existing user’s definition

- Select the user in the list and change the user details,
- Click on the Finished button and you are prompted to Save the Changes

Do not change the status of the Supervisor user, particularly if you have only one user with Supervisor status. If you change the status of the Supervisor user you will no longer be able to manage the project and its users.

To delete a user

- Select the user in the list and click on the Delete button
- Click on the Finished button and you are prompted to Save the Changes

Checking in Components

The final step in setting up your project is to check the components in your application into the VCS. You can check in complete Omnis libraries to include all the classes in those libraries, you can check in individual classes from any number of different libraries, and you can check in a whole folder hierarchy containing all the necessary files for your application. Once you have checked all the components in, other users can build a local working version of your library or project using the Build Project option.

Creating a New Project

When you check in your library classes and other components for the first time a new project is created for you automatically. Using this check-in method creates a new project with the same name as your library. Alternatively, you can create a new project using the ‘New Project’ option in the VCS window, although this is not necessary if you start by checking components into the VCS.

Checking in Omnis Libraries and Classes

All developers working on a project should have access to all the Omnis classes in your library. Therefore, you should check in all the necessary classes in your library, including any system tables, superclasses, and task classes. The system tables contain information about the fonts, display formats, and so on, used in your library. In particular, if the objects in your library use field styles you should remember to check in the #STYLES system table. You can use the Class Filter option (press F7/Cmnd-7) in the Studio Browser to make sure all the classes in your library are displayed. In future, if you change the system tables in your local library you must remember to check them back into the VCS along with any other classes you may have changed. You should not check in the #DEBUG system table class that may appear in Omnis; this exists temporarily for internal use only.

To check an Omnis library or individual classes into the VCS

- Open the Studio Browser and your library containing the Omnis classes you want to check in to the VCS

If you want to check in individual Omnis classes

- Display the classes in your library, and drag and drop your library or selected classes onto the VCS node in the Studio Browser tree (or if you have created a project manually, you can drop the components on the project name)

or you can

- Select your library or individual classes in the Studio Browser, right-click on the object(s) and select the Check-in option from the context menu

Whichever method you choose, the Check in components dialog appears. This dialog lets you set the check in options for the selected components. If you check in a library all the classes in the library are checked in automatically.

- Select the Add new component radio button, and type “Initial check in” or something similar in the Check In Notes field

When you check in classes or libraries *for the first time* they are set to read-only, so they are protected before the first build has been performed. You can control this behavior on the Check in tab in the VCS Options (right-click on the VCS node in the Studio Browser).

Version Numbers

The version string for each component is currently set to “1”. You can enter a special version in the Version field if you want to use a different numbering scheme, although this is not recommended. All the components in the VCS have both a version and a revision number, each of which is an integer value. The version indicates the major revision or release of a component and typically applies to all the classes in the library, so it’s effectively the version of the library. The revision number indicates a relatively minor change to the component. When you first check in a component, you can assign it a version; the VCS automatically sets this to 1 and the revision number to 0. Each time you check in a component, the VCS assigns a new revision, but the version doesn’t change unless you change it explicitly in the check in dialog.

- Click on the Continue button to start checking in

A progress bar shows the number of components added. When all the components are checked in, the project appears in the VCS tree, with the same name as your library.

Prompt for Options and Notes

The VCS will prompt you if you have not activated the Options and Notes tab when checking in/out. You can manage this behavior on the VCS Options, Check Out & Check In tabs using the new “Prompt for Options and Notes” option.

Checking in file system folders

The VCS checks in external components *without the file system folders* being created within the repository. The old behavior (pre-Studio 10) can be restored by unchecking the Check-In preference “Ignore file system folders for external components”.

Checking in non-Omnis Components

You can use the Omnis VCS to control most other types of non-Omnis components such as external components (DLLs or plug-ins), documents, Web pages, PDF files, Omnis data files, and so on. You can use the Omnis VCS to manage any type of project containing any number of files, including ones that don’t contain Omnis libraries or classes.

File Browser

To check in non-Omnis components, you use the File Browser available within the VCS itself.

Important Note: You can check Omnis library files into the VCS using the File Browser, but this is only appropriate if you want to manage the library as a discreet disk file. However, if you want to access all the classes in the Omnis library, you should check the library into the VCS as separate classes, as described in the previous section, and not using the File Browser.

To check non-Omnis components into the VCS

- Select your project under the VCS node in the Studio Browser (not in the right-hand pane)
- Click on the File Browser option
- Locate the files or folder you want to check into the VCS using the Select Files/Folder button

From thereon, the check in process for non-Omnis components is exactly the same as for Omnis classes, that is, the Check in components dialog appears which lets you set the check in options for the selected components.

If you check in a folder, all the files and *all subfolders and files* within that folder are checked into the VCS. In this case, you could, for example, check in a complete folder & file structure containing your own Help system.

Viewing the Contents of a Project

To see the contents of a project

- Select the project name in the VCS tree, or double-click on the project icon in the VCS Browser

You may find the Details view most useful when viewing the components in a project, since that view shows information such as the version, the status, which user has checked each component in or out, and so on. You can change the view using the View menu on the main Studio Browser menubar. In addition, you can sort the contents of your project by clicking on one of the column headers in the VCS Browser.

Project Folders

You can create *folder classes* in an Omnis library to allow you to store and organize the classes within the library. When you check a library into the Omnis VCS these folder classes are copied into the VCS project. It is also possible to create a folder class directly within a VCS project via the hyperlink option ‘New Folder’ available at the VCS project level. This will allow you to organize the classes in your VCS project, but will also allow you to organize non-Omnis objects such as external components.

Depending on how a project folder is created and what it contains, it can have one of three possible states:

- a “*normal“* folder is one that is generated within an Omnis library and contains only Omnis classes
- an “*external“* folder is one that is generated from within the VCS
- a “*hybrid“* folder is one that contains both Omnis classes and non-Omnis Objects

Building a Project

When you build a project, *normal* folders will be built in the destination library. *External* folders will be built into the file system using the folder name as a directory appended to the build path with any non-Omnis Objects built there. A *hybrid* folder will build a folder class in the Omnis library along with any Omnis classes inside it as well as building to the file system.

Moving classes & components between folders

If you move a non-Omnis object to a folder class that was previously generated from an Omnis library (a *normal* folder), that folder will become a *hybrid* folder. The same thing will happen when you move an Omnis class to an *external* folder.

Note: Once a folder has become a *hybrid*, it will remain one: Therefore empty folder classes may be generated when building a project even if you have deleted all classes from the folder.

Hiding and Showing Project Folders

It is now possible to hide the folders within a VCS project thereby giving you a flat view of all the components in your project. This may be useful when you want to quickly locate a class rather than navigating the entire folder structure in your project.

To hide or show the folders in your project, click on the Hide / Show folders hyperlink option in the VCS Browser.

Assigning Component Privileges

When you have checked in all the necessary components into your project, you need to assign access privileges to each component for each user.

To grant component privileges

- Select the set of components for which you want to grant the privileges
- Click on the Privileges option or right-click on the component(s) and select the Privileges option from the context menu

The Assign Component Privileges window lets you assign privileges for specific components to particular users.

- Select one or more users and one or more components
- Select one of the levels of privilege, either Read/Write, Read Only or Info Only, and click the Set button to grant the privileges

Alternatively, you can check the ‘Assign privileges when checking in for the first time’ option under the Check In tab on the VCS Options dialog to ensure that all users are granted access to all components in your project automatically when you first check in the components.

For more details about assigning privileges to components see the *Managing Components* section.

Using the VCS

This section is for developers who wish to check out components from the VCS and start working on their application. Lead developers or managers (Supervisors) wishing to setup a VCS repository and create a new project should read the beginning of this chapter.

Once the Supervisor has created a project, set up the users, checked in classes and granted access to components, users or developers can start using the VCS. Assuming you have ‘Participant’ user status, you can:

- Sign in to the VCS
- Check out one or more components that you need to work on
- Check those components back into the VCS once you have finished with them

Signing in to the VCS

To sign in to the VCS as a user

- Click on the VCS option in the Studio Browser
- Click on the Open Session option and select the appropriate project in the list
- Enter your user name and password

If you don’t have a valid user name and password, you can sign in as a temporary user by checking the Observer check box. An observer can view information about the components stored in a project, but cannot check components in or out, or perform any other VCS tasks. Alternatively, you can check the Create User check box to create a new user and password, which signs you in with Participant status. The Supervisor can change your user status at a later stage if required.

- Click on OK to sign in to the VCS

Checking Out or Copying Out Components

When you *check out* a component into a local library, it becomes locked in the VCS, preventing other users from checking it out. A locked component is shown in the VCS with a lock icon. Alternatively, if you *copy out* a component it is not locked in the VCS and other users can check it out if they wish. When copying out a component you can change it in your local library, but you cannot check it back in to the VCS. In practice, copying out is a convenient way of viewing components locally without locking them in the VCS.

You can check out or copy out multiple components at the same time, but they must all be Omnis classes or all non-Omnis components during a single check out process. Components may have the same or different target libraries or folders.

To Check out or Copy out components

- Display the components in your project by double-clicking on the project in the VCS Browser, or selecting the project name in the VCS tree
- Select the components in the VCS Browser that you want to check out or copy out
- Select the Check Out option, or right-click on the component(s) and select the Check out option in the context menu

or for Omnis classes only

- Drag the classes from your VCS project and drop them on the appropriate library in the Libraries node in the Studio Browser
- In the Check out dialog, select either Check Out or Copy Out and whether the component should overwrite an existing one or prompt for a new name (you can change the default for both these options in the VCS Options)
- You can add a description for the checking or copying out process

You can add a separate note for each component by clicking the Expand Notes button.

- Click on Continue

If the VCS cannot locate the original library for an Omnis class, it will prompt you to select a target library. You should select a library, or skip this component. The VCS may prompt you to locate a library for further classes. For non-Omnis components the VCS prompts you for a destination folder.

If you check out an Omnis class that has a superclass or belongs to a design task, read-only copies of the superclass(es) and design task are copied out, assuming the ‘Automatically copy out related components’ preference is enabled in the VCS options.

In the Libraries tree, Omnis classes are shown as checked out with a lock icon.

Check Out from Find & Replace Log

You can check out a class from the VCS directly from the Find & Replace log window, assuming the class is not already checked out. You can select a line or multiple lines in the Find log, right-click on the selection, and select the Check Out option. You will be prompted to log onto the VCS if required and the Check-out dialog will be displayed containing the selected classes ready to be checked out.

Checked Out Classes

The “Checked Out Classes” option in the main VCS browser opens a list showing all the classes that are checked out by the current user, or for the Supervisor user, the window shows a list of checked out classes for all users.

Updating Properties

When checking out a class, you can update the destination library properties with the properties that are held in the VCS for that project. This is especially useful for situations in which there are multiple developers working on the same project. You can enable this feature by checking ‘Update the destination library with preferences from the VCS’ option on the Check Out tab of the VCS Options.

Superclass and Design Task Names

You can maintain the superclass or design task library name for checked out classes by enabling ‘Maintain superclass / design task library name’ option on the Check Out tab of the VCS Options. In this case, the superclass (or design task) library name is *not* removed from checked out classes. This is necessary when the superclass exists in a separate library and there is a class with the same name as the superclass in the current library.

Showing Checked Out Classes in the Studio Browser

There is a hyperlink option in the Libraries view of the Studio Browser “Show Checked Out” to display only checked out classes in the current local library. Once enabled you can click the option again to show all classes.

Checking in or Unlocking Components

When you have finished making changes to a class or component in your local library, you need to check it back into the VCS. This unlocks the component in the current VCS repository and allows other users to check it out and make changes or propagate your changes into their local libraries.

Checking in a component releases its lock and makes it available to other users for checking out. The VCS stores the changes in its repository and updates the revision number. You can see the current version and revision by clicking on a component and clicking the Information option when the component is selected.

You can also unlock a component without checking in a new version to make it available for other users. You could do this if you decide you don’t want to make any changes to a component after all, or if someone else needs to make immediate changes to a component that you have checked out and is currently locked. In the latter case, you can unlock the component, and check it out again when the other user has made their changes. You can unlock selected components by clicking the Unlock option when the component is selected.

To check in Omnis classes

- Select the component(s) in your local library in the Studio Browser
- Drag the selected component(s) on to the appropriate project in the VCS Browser

or

- Right-click on the component(s) and select the Check In option from the context menu

To check in non-Omnis components

- Assuming your project is selected in the VCS, click on the File Browser option
- Locate the folder containing the files your want to check into the VCS

For all types of component, the Check in window appears which lets you set the check in choices for the selected components, as already described in the Setting up a Project section.

You can select one of the following check in modes

- **Add new revision**
  adds the component by incrementing the revision number by one, unlocks the component, and updates the check in date, time and other status information
- **Add new component**
  adds the component as a new one, setting the version to 1 and revision to zero, by default

The After Check in option lets you decide whether to

- **Keep checked out**
  adds the component to the VCS and locks it; in effect, this allows you to backup your components but keeps the component(s) checked out and available for you to make further changes
- **Delete local copy**
  adds the component to the VCS and deletes the local copy

You can enter a new version in the Version field or accept the default, and regardless of the check in mode, the VCS sets the version to that number.

Showing Checked Out Classes

You can view only the checked-out classes in a VCS project. To show the Checked Out classes in your project, click on the Show Checked Out hyperlink option in the VCS Browser. To show all classes click on the Show All Classes option.

This option may be useful when you want to quickly identify all the classes that are checked out from a project, and you can combine this option with the ability to hide/show project folders, as above, to view all checked out classes within folders.

When only the Checked Out classes are showing you can select one or more classes, right-click on them and select the Check-in option. You can also run the Class Comparison tool from the same context menu. If there are classes that belong to more than one library, or if a library is not open, these options are disabled.

Building Projects

The Omnis VCS lets you build a project on your local workstation, or any other destination, from the components stored in the VCS repository. You can do this at any time using the Build options in the VCS. You can also build projects at a specific time and date using the 'Scheduled Build' option, which is described in the next section in this chapter.

Building any type of project from the VCS guarantees that all the components in your local copy are up-to-date. In the context of Omnis application design, building a project means creating a library containing up-to-date classes for you to work on or test. If your Omnis application contains multiple libraries, doing a build from the VCS guarantees that your whole application is up-to-date. You can also build a project containing non-Omnis components and reproduce the original folder hierarchy required in your application. Using the revision labeling features, you can build previous versions of a library or project for comparison, troubleshooting, or debugging.

When you build a project that contains classes from more than one library, the VCS copies all the components to a single library by default. However by setting the Build options you can build classes to separate libraries, thus maintaining your original library structure.

You can update a local copy of a library in the Studio Browser using the 'Update from VCS' option. This is appropriate for updating single libraries, but is not appropriate for building a complete project. The Update from VCS option is described later in this chapter.

To build a project with multiple libraries

- Select the VCS in the Studio Browser and click the Options option
- Click on the Build tab and check the 'Maintain Project Structure' option; this will ensure your project builds to multiple libraries where applicable

To build a project

Whether or not you are building to one or more libraries:

- Select your project in the VCS Browser
- Click on the Build Project option, or right-click on the project and select the Build Project option from the context menu

The Build Manager lets you configure the project build in detail. It contains one line only if you have chosen to build a project containing Omnis classes into one library, with the project name as the library name. If you have set the preference to maintain your library structure, the Build Manager lists each separate library in the project. If your project contains non-Omnis components you can enter the name and path of the target folder.

- Select the library or project and specify the Build options as follows

You can name the output library using the Label and Build As options:

- **Label**
  lets you select the version of the library that you want to build; see the Labels section below
- **Build As**
  lets you select the name of the output library

You can set the following options for the output library:

- **Use Locked and Unlocked folders**
  if set, you can use locked folders, otherwise if the option is unchecked, all folders and libraries are unlocked

- **Locked and Unlocked**
  if set, creates a locked and unlocked version of your library

- **Locked**
  if set, the build process creates a locked version of your library preventing other users from seeing or changing the contents of the library; checking the Locked check box option locks the whole contents of a library but you can lock individual classes using the Lock Classes window.To lock individual classes, uncheck the Locked option, right-click the project in the Build window, select the Lock Classes option, and select the classes you want to lock in the popup window. Note you can also lock folders, which will lock the whole contents of the folder

- **Disable Class Data Notation**
  if set, you will no longer be able to read or write $classdata in the built library from any Omnis class using Omnis code. In addition, JSON export of the library is disabled as the class data is disabled. IMPORTANT: future access to this library in the Omnis VCS will no longer be possible. **Setting this property is an irreversible operation.**

- **Disable Method Text Notation**
  if set, you will no longer be able to read or write $methodtext or $methodlines in the built library from Omnis code or via the Property Manager. In addition, method text will not be exported during a JSON export of the library. **Setting this property is an irreversible operation.**

- **Multiple Build Paths**
  when checked, you can select separate build paths for each project using the context menu on the project list

- **Strip Comments**
  removes comments from your Omnis code, as well as variable & file descriptions; also enables the class selection context menu; see the Remove Comments section below

- **Overwrite**
  if true, overwrites all the components in the target library or folder, that is, the VCS removes all the classes in the target library and completely rebuilds the library. Otherwise if the option is unchecked, the VCS updates only the components that are different from the current library or project. This is useful for quickly bringing an existing library up-to-date with the current checked-in classes in the VCS

- **Lower case**
  ensures the library name is in lower case

- Type the path for your library or project in the 'Build into' field or use the Browse button to select a folder or create a new one

- When you have set the options for your project, click on the **Build** button

The VCS opens the Build Results dialog showing the progress and status of the build. The VCS creates a log file for the build and displays it at the bottom of the build window.

Excluding Classes

You can exclude specific classes from a build. The context menu on the project class list includes the option "Exclude Classes" which allows you to select classes you do not wish to include in a build.

Removing Comments

When you create a build from the VCS, you can remove the comments from the code in your libraries by checking the 'Strip Comments' check box. If the option is set, all comments within methods are removed, as well as all variable definitions, and descriptions for file, query, and schema classes. This will make library files smaller, which may be better for deployment.

You can select classes that you want to retain their comments when you build a project. To do this, right-click on the list of projects in the Build Window, select the 'Do Not Strip Comments' option, and select the classes in which you want the keep comments.

Under the Build tab in the VCS Options window, there are four checkboxes that, if checked, will ensure that comments in variables, file classes, query classes and schema classes are retained.

Checked out Classes

When you create a build from your Omnis VCS repository, the project may contain classes or objects that are checked out of the current repository. If you proceed with the build, your project may not contain the most up-to-date classes. In this case, you may want to look in your project before building the project to ensure that all the classes in the project you want to build are all checked in.

If the 'Warn if there are classes checked out' option is set (in the VCS Options), the VCS will warn you that the build contains classes that are checked out. In this case, a window will open listing all the checked out classes with an option to proceed with the build or cancel it. To build a completely up-to-date copy of a library, you should ensure that all classes are checked into the VCS before proceeding with a build.

System Tables

When you build a library, the VCS includes the system tables by default, assuming you have copied them to your project. The system tables contain library-specific settings such as fonts, input masks, field styles, and in the case of #ICONS the icons in your library, so they are important in maintaining the correct look-and-feel and behavior in your application. Remember that when you check in a version of the library, the VCS does not automatically put the system tables into the library; you must specifically show and select them in the Studio Browser just as you do the other classes in your library.

Scheduled Builds

The Omnis VCS allows you to build projects at a specific time and date, which for large development teams may be useful if you need to build a library out of office hours. You can set up a Scheduled Build from the 'Scheduled Build' option in the main Omnis VCS Browser. When you select this link, the Build Manager will open containing a new button called 'Schedule…', which opens the 'Schedule Build Process' window.

The Schedule Build window allows you to control the frequency of the builds and set the time that you want the build to run. The following options are available:

- **Never**
  no scheduled builds occur, but you can still build a project manually
- **Daily**
  the build will occur every day at a specified time
- **Weekly**
  you can specify which day(s) of the week the build should occur during the week; you can also set the time
- **Monthly**
  the build will occur once a month, on the specified day of the month; you can also set the time

The best time to schedule the build is when there is no one else logged on to the VCS. You must ensure that Omnis Studio is running at the time of the build and that you are logged on to the correct VCS repository. If you do not have Studio open at the time of the build schedule, when you next log on during that day, the VCS will prompt you that you have missed the scheduled build and ask if you want to run it then. For example, if you have the build set to run at 7am on a Monday, but do not start Omnis until 8am, Omnis will remind you about the build that you missed.

You should use the 'Build Manager' window to define the projects you want to build and how you want them to be built using the normal options that are available. When you click the Finish button these preferences will be saved and the timer started according to the schedule defined.

Labels

You can choose to label the build either at the time you schedule the build or when the build runs. Note that if you select 'Label at Build' and there is already a label with that name, the VCS will redefine the label with the current state of the classes. This may or may not be what you require, so caution should be exercised, particularly if you are setting the build to run every day and therefore you may lose the ability to rebuild to a specific point in the project development at a later point in time. If you select 'Label Now', the VCS will prompt you if there is already a label defined and give you the chance to confirm that you want the label redefined.

Locked or unlocked

You can specify whether the VCS builds a locked or unlocked version of your library (or both) using the Library Status dropdown list in the Build window.

Build notes

If you have selected a path to save them into, the build notes will be saved to a file prefixed 'AutomaticBuildNotes_' within that path. By default the path is the BuildFolder within the main Studio folder.

Labels

In long-term projects, the components in your project may undergo many revisions. The VCS tracks these revisions using labels. You can reproduce a particular version of a library using the appropriate label. A label is a string of up to fifty characters that you can assign to a project and each of its components. When a project is complete and you want to release it, you can assign a release name by labeling the project. You can see the label for a project or individual component in its Information window.

To label a project

- Select the project in the VCS Browser
- Click on the Label Project option, or right-click on the project and select the Label Project option from the context menu
- Type in the label and click OK

Once a project has been labeled, the last label assigned is shown in the VCS Project Browser if the browser is in Details view.

When you build a project, you can use labels to select either the latest version of all the components, or an older version. You can also check out a specific revision of a component using its label. See the description of the Check Out process in the section on *Managing Components* below.

If you delete a component that has a label, it remains in the VCS. If you build a project using that label, the VCS finds and includes the component in the build even though you have deleted it from later revisions of the project. The VCS does not include the component in any builds or labels that occur *after* you delete the component. For example, if you label a project and its components on Monday, delete a component from it on Tuesday, and do a build on Wednesday using Monday's label your library *will* include the component. However if you do a build on Wednesday using the most recent project label, your library *will not* include the component you deleted on Tuesday.

Server Connections

When the VCS loses its network connection, the list of classes and all hyperlink options are hidden and replaced with a **Refresh** option (previous versions may have issued numerous messages when a connection was lost). Pressing Refresh will poll the database to see if the connection has been restored, but if the connection is still down an error message will be shown. Polling the database may still result in a wait of several seconds before the connection is restored or any error message is shown.

Finding Classes

The 'Find Class' option in the VCS Browser lets you search for a class within the current VCS repository. You can search for a class within one or more projects. A list of matching classes is displayed, and you can double-click a class to display that class in the appropriate project.

Updating local libraries from the VCS

The 'Update from VCS' option, available in the main Studio Browser, allows you to bring a local library up to date with the latest version of the library in the current VCS repository. If you have large libraries and multiple developers, you may find this a quick way of ensuring that a local copy of a library has all the changes made by other developers.

To use this feature, select a library in the Studio Browser and click the 'Update from VCS' option. Providing that you have a VCS session open and that there is a project in the VCS repository that matches the name of the current local library, this option will bring up the checkout window (in Copy Out mode only) listing all the classes that have a newer class modified date in the VCS than your copy of the library. It will also list all classes that have been added to the repository that do not exist in your local library. If there are classes which you do not want to update, you can uncheck them in the checkout window.

Read-only classes

When you try to modify a class in a library built from the VCS, it is only possible to edit the class when the class is checked out. If it is not possible to edit the class, the class is considered to be ”read-only“.

- If a library has a non-empty VCS build date, then all class editors behave in a read-only fashion, when $showascheckedout is kFalse for the class being edited.
- You can disable this behavior using the $alloweditifnotcheckedout Omnis preference ($root.$prefs) which is set to kFalse by default. Set it to kTrue, to disable this behavior.
- When this behavior is enabled, new classes in a library with a non-empty VCS build date (created in any way), are automatically marked as checked out.
- Replace will not work against a read-only class, an error is written to the find and replace log.

Note that this does not affect the notation (although the notation inspector will not allow changes to a read-only class).

Managing Components

The VCS has an extensive array of component management functions, including

- Granting user privileges for components
- Associating a component with more than one project
- Managing revisions of components
- Deleting and renaming components

Granting User Privileges for Components

If you are the first user to put a component under version control, you are the *owner* of that component. To allow other developers access to a component, you must grant them privileges to it. Without the right privileges, other developers can only view and get information about a component as Observers. Supervisors have all privileges at all times.

To grant component privileges

- Display the components in your project by double-clicking on the project in the VCS Browser
- Select the set of components for which you want to grant the privileges
- Click on the Privileges option or right-click on the component(s) and select the Privileges option from the context menu

The Assign Component Privileges window lets you assign privileges for specific components to particular users. You can choose one of the following levels of privilege

Privilege level	Description
Read/Write	the user may check in, check out, copy out, get information on and build libraries with components
Read Only	the user may get information on, build libraries with, and copy out a component into a local library, but cannot check out the component nor check it back in with changes
Info Only	the user may get information only about a component

- Select one or more users and one or more components
- Select one of the levels of privilege and click the **Set** button to grant the privileges

Assigning the Info Only privilege also revokes existing privileges from a user. Info Only is the default privilege that any user has for a component. The owner of the component must grant the user, including Supervisors, the Read Only or Read/Write privileges necessary for building libraries or changing the component.

By default, the owner of a component starts with Read/Write privileges for that component, and you must have Read/Write privileges to grant privileges for it to other users.

If you select components for which you don’t have Read/Write privileges, the VCS disables the radio buttons and Set button. If you select components for some of which you have Read/Write privileges, you can use the radio buttons. The VCS does not however, grant the privileges to the user(s) on those components for which you do not have Read/Write privilege.

Sharing Components between Projects

You may decide to use a single component in many projects, such as a custom logon window, or a generic search component. The VCS can *associate* or *link* a component with libraries in different projects, storing the component only once, but building it into different libraries when required. You need only revise the component in one place, rebuild the projects, and the VCS propagates the changes to all the libraries that contain the component.

You must have Supervisor access or be the owner of a component to associate or link it with another project or library.

To associate a component with a project

- Open the project containing the component, and select the component
- Click on the Link option, or right-click on the component and select the Link option from the context menu

The Link option displays a window showing all your open projects (the link window will not open if there are no other open projects, or if the component is already linked to all other open projects).

- Select the project with which you want to link the component, or you can open a project in the list and select a particular library
- Click OK to link the component

If the Maintain Project Structure build option is not checked, the component is associated with the single library together with all the other components. Otherwise if the build option is enabled, the component is associated with a library of the same name as the one containing it. If such a library does not exist, it is created automatically.

To delete a linked component in a specific project

- Open the project containing the linked component
- Select the component, and click on the Delete option, or right-click on the component and select the Delete option from the context menu

If the component has a label, it becomes disassociated and a small trash can icon is shown. If you build a project using a label, the component is still available for the build, even though the library or project in question may no longer associate with the component.

Managing Revisions

When you first check in a component, the VCS creates the component in its repository and sets the version to 1 and the revision to zero. Each time you check in a new revision of the component, the VCS stores the new component and increments its revision number. The sequence of changes to a component is stored in its *revision history*.

To see the revision history for a component

- Select the component in the VCS
- Click on the Revision History option, or right-click on the component and select the Revision History option from the context menu

The Revision History window tells you

- the version string and revision number
- the user who modified the component
- the component name
- and change notes, if any

The revisions for a component are listed in chronological order with the most recent revision at the top. You can delete old or newer revisions of a component.

If you have Read/Write or Read Only privileges for the component, you can copy out the latest revision of the component from the Revision History window by clicking on the Copy Out Revision button. A list of currently open libraries appears, so you can choose which one to copy the component into. Select a library and click on Select.

If a component already exists in the library with the same name as the copy, the VCS prompts for a new name. Keeping the old name overwrites the existing component; entering a new one renames the copy.

Viewing all Project Revisions

The "Project Revisions" allows you to see all the revisions to a project, rather than having to drill down to a class and see the revisions which are only available for that class.

The Project Revisions option is available at the project level as a *context menu* option in the tree list or via the hyperlinks when clicking in the list of projects. You can filter the revisions from the droplists according to the user who created the revision or a date period. You can also filter by typing in the edit box.

You can drilldown to see which labels have been applied to the revision by right-clicking on it. This window also allows you to compare or copy out in the same way as the class level revision window. You can also select multiple classes to copy out multiple revisions. It is also possible to double-click rather than using the context menu to view the revision data.

VCS Revisions

All class types now have the $vcsrevision property, which allows the Omnis VCS to determine whether or not classes in a local library are up to date with the latest revision in the VCS repository. The $vcsrevision property is the revision number of the class, and is used for classes stored in the Omnis VCS. It is set to zero if the library was not built by the VCS (i.e. it was created in the IDE), or if the library was built prior to Omnis Studio 10.2.

The property is read-only in the Property Manager, as it is intended for use by the Omnis VCS library, or any custom class editing tools you may have created. It can only be assigned by executing code.

Method Inspector

The Method Inspector lets you examine the methods in a class without having to Check Out or Copy Out a class or build the library. This may be useful if you want to quickly check what methods are contained in a class or what code is contained in a particular method within a class.

To open the Method Inspector, you can select the class and choose the 'Method Inspector' hyperlink option in the VCS Browser, or Right-click on the class and select the option from the context menu.

On the first tab of the Method Inspector window, there is a list of all the methods within the class. If the class has a superclass, the superclass methods are also shown. They are displayed in blue to differentiate them as inherited methods.

If the selected class is a window, report, remote form, menu or toolbar, there is another checkbox displayed at the bottom of the list to allow you view field methods. By default this option is not selected except for menu and toolbar classes.

Further properties of the method are shown in this tab: description, execute on client and web service method. There is also a context menu option to allow you to view the path to the method within the class – this is of most use if you are looking at field methods as you can see the exact path to the method.

If you double-click a method, the code for the method is displayed on the second tab. Should the method be inherited, there is a context menu option to allow you to view the superclass code.

Orphan Components

It is possible for a component that has a parent folder to lose its parent folder; in this case, the component is regarded as an 'orphan' component. A component can lose its parent folder either because the parent folder itself no longer exists, or the id of the parent folder has changed and the component is not aware of the change. The id of the parent folder for a component is stored in its $parentfolder property.

The 'Find Orphans' option in the VCS will look for components where their $parentfolder id does not exist and place these components in the 'Orphans' folder. You can move the component(s) back to their respective folders or recreate the appropriate folder. In most cases, the option will return no orphan components.

Component Services

The Component context menu gives you information about a component, and lets you rename, delete, or unlock a component. To open the context menu you must right-click on the component.

To get Info about a component, or rename, delete, or unlock one

- Right-click on the component in the VCS and select the appropriate menu option

or assuming the component is selected, you can

- Click on the Info, Delete, or Unlock option in the list of options in the VCS Browser

Get Information

This window displays the information about the component, including the labels, the revision state, the check out history, and links.

Deleting Components

To delete a component, you must own the component and it must be checked in. The Delete option checks whether the component belongs to more than one library and whether any labels apply to it. If not, the VCS prompts for confirmation that you want to delete the component. If it does belong to more than one library, the VCS displays a list of associated libraries, letting you choose the ones from which to remove the component.

Unlocking Components

You can unlock a locked component by clicking on it and selecting the Component>>Unlock menu option. A warning message advises you that having done this, you will not be able to check in a previously checked out version of the component.

Renaming Components

To rename a component, you must own the component and it must be checked in. You can rename any component in the VCS, but renaming Omnis classes requires some additional care because of the potential dependencies of some classes on the class you want to rename. That is, before you rename a class in the VCS you must change all references to it in any other classes that refer to the class. You do this by checking out all the relevant classes to the IDE Browser and using the Omnis Find and Replace tool.

To rename a class

- Check out the class and any other classes that contain methods which refer to the component
- Change the name of the component in the Studio Browser, and answer Yes when you are prompted for whether or not you want to use Find and Replace; this renames the component and changes all references to it in any other components
- Check in all the components *except* for the renamed one
- Select the component in the VCS, and click on the Unlock menu option to release its lock
- Click on the component again when the VCS has refreshed
- Click on the component to rename it

Comparing Classes

The Compare Classes option in the VCS lets you compare all the classes in two different VCS projects, or it lets you compare individual classes in different projects. It also lets you compare different revisions of the same class. (The Compare Classes tool is also available in the Studio Browser and allows you to compare classes in libraries not checked into the VCS.) The Compare Classes tool compares the properties and methods of the specified classes and highlights any differences. Specifically, the comparison tool will compare each line in a method of one class with the corresponding lines in a method of the other class and will highlight even the smallest change or difference between the two classes or revisions.

To use the Compare Classes tool

- Select a project or component and click on the Compare Classes option in the VCS Browser

The Compare Classes tool is also available in the Revision History window to allow you to compare revisions of the same class.

The Compare Classes window lets you load two different versions of the same VCS project or library: you should load the older project or library into the Library/Project A list (on the left) and the newer version into the Library/Project B list (on the right). The context menu on the Compare Classes window (Right-click to open it) lets you switch libraries from List A to List B.

You can compare all the classes in a VCS project or library that have a modified $classdata, which avoids comparing classes that have not changed. The Compare tool shows which classes have a modified $classdata and by default will compare only these classes. You can uncheck the ‘Only include classes where $classdata differs’ checkbox to compare all the classes in a project or library.

By default, the Compare Classes tool only compares any classes that are selected in the VCS. To compare all classes, regardless of any selections in the VCS, you can uncheck the ‘Only include classes pre-selected in the class browser or VCS’ option in the Compare window.

VCS Options

You can set your individual preferences for the VCS using the Options option in the list of options in the VCS Browser. You can set Display, Check out, Check in, Build, and Method Inspector options in the VCS Options window.

To set your VCS options

- Select the VCS in the Studio Browser, or select a project, and click on the Options option

Display Options

The Display options let you specify the format of dates and times for the VCS windows and dialogs. The default format is: D m Y H:N:S, such as 12 May 09 14:30:35.

- **Show linked icon**
  (disabled by default) enables the Linked icon for classes and components that are linked across different projects using the Link option.

Check Out Options

The Check Out options control how components are checked out of the VCS.

- **Default mode is "Check Out Modifiable" copies**
  determines whether or not a component is checked out or copied out; the default is for components to be checked out in modifiable mode.
- **Default replacements to overwrite existing items**
  determines whether or not an existing component in your target library is overwritten when a component is checked out. The default is that components are checked out and overwritten without warning, otherwise when the option is disabled, you are prompted to overwrite or rename the component.
- **Automatically copy out related components**
  relates to Omnis classes that have superclasses and design tasks. If you have a class that is a subclass of another class, and you check out the subclass, this option ensures the superclass is checked out. When this option is disabled the superclass will not be copied out; in this case, you will not see any fields and methods that are contained in the superclass. If this option is set, read-only copies of the superclass(es) are copied out, as well as the design task for the original classes being checked out.
- **Default to copy out component if already checked out**
  when this option is set, it forces components to be checked out even if they are already checked out. If this option is disabled, the VCS will warn you that the component is already checked out and prompt you to Copy out or cancel the operation.
- **Maintain superclass / design task library name**
  ensures that classes retain the library name as part of the superclass or design task name. The preference is disabled by default to ensure backward compatibility. With the preference disabled, the library name is stripped out at checkout or build time. This means that if, for example, you have 'LibraryA' with a class called 'classA' which has a superclass 'LibraryB.ClassSuper' and there is also a 'ClassSuper' in 'LibraryA', Omnis would switch the superclass to 'LibraryA.ClassSuper' whenever the class is checked out or built. You can check this option to retain the superclass name to avoid this problem.
- **Update the destination library with preferences from the VCS**
  ensures that when enabled the preferences of the destination library are updated from those stored in the VCS.
- **Ignore Checked Out Classes**
  when enabled the VCS will not copy out a component if it is already checked out.
- **Prompt for Options and Notes**
  when enabled the VCS will prompt you for Options and Notes when checking in/out
- **Show all options without tabs**
  when enabled allows you to show all Check in/out options in one screen by hiding the tabs

Check In Options

The Check In options control how components are checked in to the VCS.

- **If Target Component Does Not Exist**
  controls what happens when you check in a new component. By default, the 'Automatically add it to the Project' option is selected and components are added to a project automatically; otherwise if the 'Prompt for further instructions' option is selected the VCS asks if you want to add the component to the current VCS project.
- **When Component Version Number Changes**
  controls revision numbers for new versions of a component. By default, the 'Reset revision numbers to zero' option is selected and revision numbers are set to zero for new versions; otherwise if the 'Continue to increment revision numbers' option is selected the VCS continues to increment revision numbers regardless of the component version. (Version and revision numbers are described earlier in this chapter.)
- **Assign privileges when checking in for the first time**
  is useful when, as supervisor, you first set up your project. When enabled and you check in components into the VCS for the first time, the component privileges window is opened allowing you to set the access privileges for each component.
- **Disable automatic library preference changes**
  If enabled, this option prevents the automatic updating of library preferences from the library containing the classes to be checked in.
- **Enable version number validation**
  allows you to disable version number checking when checking objects into the VCS.
- **Prompt for Options and Notes**
  when enabled the VCS will prompt you for Options and Notes when checking in/out
- **Ignore file system folders for external components**
  when enabled the VCS checks in external components *without the file system folders* being created within the repository
- **Show all options without tabs**
  when enabled allows you to show all Check in/out options in one screen by hiding the tabs
- **Make library read-only when checking in for the first time**
  when enabled the library will become read-only after being checked in for the first time

Build Options

The Build options control builds in the VCS.

- **Maintain Project Structure**
  controls whether or not a build retains the library structure of the components within a project. If this option is enabled, components are built into separate libraries mirroring the original library structure, otherwise all the components in a project are built into a single library.
- **Only create locked and unlocked folders when required**
  Selecting this option prevents the VCS from generating locked and unlocked folders in the build path unless the Build process needs them.
- **Warn if there are classes currently checked out**
  tells you if there are any classes checked out when you attempt to build a project; if you build a project containing checked out classes your library may not contain the most up-to-date version of classes
- **Do not build empty folder classes**
  when enabled the VCS will not build empty folders
- **When Stripping Comments**
  During a build all comments in your Omnis code are stripped out. When enabled, the 'Stripping Comments' options ensure that comments are retained in all Variable definitions (in the Variable pane of the Method Editor), as well as all File, Query, and Schema classes.

Method Inspector Options

The Method Inspector options are as follows:

- **Ignore File Classes**
  When enabled the 'Ignore File Classes' option ensures that #??? references to file classes are not included in the Method Inspector window; the Method Inspector is available in the Studio class browser and allows you to inspect the methods of any classes in the current VCS project.

Branches Options

The Branches options only affect projects that contain branches, so for all new VCS repositories the Branch options will not appear. The Checking in/out options on the Branches tab allow you to set the check in/out preferences for VCS projects that contain branches.

- **Use Default Branches**
  If the 'Use Default Branches' option is enabled (the default) and the current project has branches, the default branch is used to check out classes when using the context menu in the Studio browser. If the option is disabled, a list of available branches is displayed to allow you to choose the branch from which to check out the class.
- **Ignore Branches**
  If the 'Ignore Branches' option is enabled (the default), the VCS automatically uses the branch the component has been checked out from. If the component is present in more than one branch, a window is displayed to allow you to choose the branch from which to check out the class or component.

Reports

The VCS has a number of reports available to those with Supervisor status. You can access them by clicking the Reports option in the VCS Browser. Users without supervisor status do not have access to VCS reports and will find this menu option is grayed out.

To use the VCS reports

- Select the VCS in the Studio Browser tree and click on the Reports option

The VCS Management Reports dialog lets you select a report. The Parameters pane and Description field will change depending on the report you click on.

- Select a report and set up its parameters
- Click on the Print button to generate the report
- Select the required destination, and click on OK

Branches

Where appropriate, the reports in the VCS will reflect the existence of branches in your projects.

VCS API

From Studio 11, some of the functions of the Omnis VCS have been exposed to allow you to interact with the VCS or the contents of a project programmatically. You can make API calls to the Omnis VCS by calling:

$root.$modes.**$dotoolmethod(**kEnvToolVcs,**'vcs_method_name'**[,parameters,..]),

The first parameter is always kEnvToolVcs to specify a VCS method, followed by the VCS method name and the appropriate parameters.

Tokens

Each API call requires the use of a *token,* which is a unique string generated by the VCS when a successful logon occurs. This token is an essential parameter to all the calls (apart from $x_logonVCS) as it is the mechanism that the VCS uses to verify the validity of the API call. By default, a token will last for 60 minutes, but you can extend the token lifespan to up to 8 hours. When the token time has expired, you will be logged off automatically and will need to logon again.

Logon

The $x_logonVCS method allows you to logon to the Omnis VCS using an existing SQL session, returning a token which must be passed by the other methods.

```
Do $root.$modes.$dotoolmethod(kEnvToolVcs,'$x_logonVCS',cHOSTNAME,cUSERNAME,cPASSWORD,nTokenTime,cToken,cEr
```

```
Do $root.$modes.$dotoolmethod(kEnvToolVcs,'$x_logonVCS',cHOSTNAME,cUSERNAME,cPASSWORD,nTokenTime,cToken,cEr
```

cHOSTNAME, cUSERNAME, cPASSWORD are character strings used to identify the session to log onto. The session must have been previously set up via the SQL Browser Session Manager. cHOSTNAME is the name of the VCS session previously set up and cUSERNAME and cPASSWORD are the credentials that are setup inside the VCS (not the database credentials).

nTokenTime is a value to determine (in minutes) how long the token will remain valid for. If 0 is passed, a default value of 60 minutes is applied; the token time can be up to a maximum value of 480 (8 hours).

rSessionOrSessionPoolRef is an optional reference to a SQL session (e.g. $sessions.MY_VCS_SESSION) or a session pool (e.g. $sessionpools.MY_VCS_SESSIONPOOL). This allows you to logon to a session via code rather than using any defined VCS sessions in the SQL Browser Session Manager. If supplied, the API will attempt to logon using this regardless of cHOSTNAME. If no valid reference is passed, the logon will attempt to use cHOSTNAME.

If the logon is successful, **bStatus** will return kTrue and **cToken** will contain a token generated by the VCS which must be used to authenticate subsequent API requests.

As with all the methods (except Logoff), **cErrors** will contain any errors.

Logoff

The $x_logoffVCS method logs out of the current VCS session.

```
Do $root.$modes.$dotoolmethod(kEnvToolVcs,'$x_logoffVCS',cToken)
```

You need to pass cToken to logoff.

Get Token Info

The $x_getTokenInfo method returns information about the token and session.

```
Do $root.$modes.$dotoolmethod(kEnvToolVcs,'$x_getTokenInfo',cToken,rRow,cErrors) Returns bStatus
```

If successful, **rRow** lists the Token, Token Expiry Time, the session name you are logged on to, VCS username you are logged on with, the token timeout in minutes and the Logon time.

List Projects

The $x_listProjects method returns a list of projects.

```
Do $root.$modes.$dotoolmethod(kEnvToolVcs,'$x_listProjects',lLibList,cToken,cErrors) Returns bStatus
```

If successful, lLibList will contain the list of projects that are available in the VCS repository.

List Classes

The $x_listProjectClasses method returns a list of classes in the specified project, and has the following syntax:

```
Do $root.$modes.$dotoolmethod(kEnvToolVcs,'$x_listProjectClasses',cLIBNAME,lClassList,cToken,cErrors) Retur
```

cLibName is the name of the project in the VCS from which you want to return the class list.

If the call is successful, **lClassList** will return a list containing the following information: className, classType, classVersion, classRevision, status (0 - checked in, 1 - checked out), checkedOutDate, checkedOutBy, checkOutNotes, checkedInDate, checkedInBy, checkedInNotes.

List Class Revisions

The $x_listClassRevisions returns a list of revisions for a class, and has the following syntax:

```
Do $root.$modes.$dotoolmethod(kEnvToolVcs,'$x_listClassRevisions',rClassRef,lClassRevList,cToken,cErrors) R
```

refClassRef is a reference to a class in your local library. If the call is successful, a list of revisions will be returned in **lClassRevList**. The first column of this list will contain the revision number.

If you wish to copy out a revision, use a revision number from lClassRevList in the parameter revID in $x_checkOut to check out the revision. For example:

```
Do $root.$modes.$dotoolmethod(kEnvToolVcs,'$x_checkOut',refClassRef,refLibRef,cToken,bCheckOrCopy,revID,cEr
```

Class Status

The $x_classStatus method returns the status of a class, its checked out status, who checked it out, and so on.

```
Do $root.$modes.$dotoolmethod(kEnvToolVcs,'$x_classStatus',refClassRef,rRow,cToken,cErrors) Returns bStatus
```

refClassRef is a reference to a class in your local library. If the call is successful, the row variable **rRow** will be populated with the name of the class, its checked out status, who checked it out and when, the date of the last revision and who checked it in, as well as the current revision number.

Checked Out Classes

The $x_checkedOutClasses method returns a list of checked out classes.

```
Do $root.$modes.$dotoolmethod(kEnvToolVcs,'$x_checkedOutClasses',cUserName,[cLibName],lList,cToken,cErrors)
```

cUserName is a character string for the VCS user name. The optional parameter **cLibName** filters the list of checked out classes to the supplied library name. If the call is successful, **lList** will be a list of classes containing project name, user name, class type, class name, checked out date, check out notes and the original library name.

Is Class Current

The $x_isClassCurrent method tells you if a local class is up to date or not.

```
Do $root.$modes.$dotoolmethod(kEnvToolVcs,'$x_isClassCurrent',refClassRef,cClassStatus,cToken,cErrors) Retu
```

refClassRef is a reference to a class in your local library. If the call is successful, **cClassStatus** will contain either 0 or 1: if 0, the class is up to date with the VCS, or if 1, the VCS version is newer than the local copy.

Check Out

The $x_checkOut method allows you to check out or copy out a class.

```
Do $root.$modes.$dotoolmethod(kEnvToolVcs,'$x_checkOut',refClassRef,refLibRef,cToken,bCheckOrCopy,revID,cEr
```

```
Do $root.$modes.$dotoolmethod(kEnvToolVcs,'$x_checkOut',refClassRef,refLibRef,cToken,bCheckOrCopy,revID,cEr
```

refClassRef is a reference to a class in your local library and **refLibRef** is a reference to the library you are checking the class out to. **bCheckOrCopy** is a boolean allowing you to either check out (kTrue) or copy out the class (kFalse). If you wish to copy out a specific revision of a class, you can use a revision number from lClassRevList returned by $x_listClassRevisions in the **revID** parameter to check out the revision.

refClassRef is a reference to a class in your local library and **refLibRef** is a reference to the library you are checking the class out to. **bCheckOrCopy** is a boolean allowing you to either check out (kTrue) or copy out the class (kFalse). If you wish to copy out a specific revision of a class, you can use a revision number from lClassRevList returned by $x_listClassRevisions in the **revID** parameter to check out the revision, else pass 0. **cCheckOutNotes** is a string containing notes to be associated with the checkout.

Check In

The $x_checkIn method allows you to check in a class.

```
Do $root.$modes.$dotoolmethod(kEnvToolVcs,'$x_checkIn',refClassRef,refLibRef,cToken,cErrors) Returns bStatu
```

refClassRef is a reference to a class in your local library and **refLibRef** is a reference to the library you are checking in from. The project must already exist in the VCS as it is not currently possible to create a new project using the API.

```
Do $root.$modes.$dotoolmethod(kEnvToolVcs,'$x_checkIn',refClassRef,refLibRef,cToken,cErrors,cCheckInNotes)
```

refClassRef is a reference to a class in your local library and **refLibRef** is a reference to the library you are checking in from. The project must already exist in the VCS as it is not currently possible to create a new project using the API. **cCheckInNotes** is a string containing notes to be associated with the checkin.

Label Project

The $x_labelProject method allows you to label a project.

```
Do $root.$modes.$dotoolmethod(kEnvToolVcs,'$x_labelProject',cProject,cLabel,cToken,bOverwrite,cErrors) Retu
```

cProject is the name of the project, **cLabel** is the label and **bOverwrite** indicates whether to overwrite an existing label of the same name.

Build Project

The $x_buildProject method allows you to build a project to a specified folder.

```
Do $root.$modes.$dotoolmethod(kEnvToolVcs,'$x_buildProject',cProject,cBuildPath,cLabel,bLocked,bOverwrite,b
```

cProject is the name of the project, **cBuildPath** is the directory to build into, **cLabel** is the label to use, **bLocked** indicates whether to build a locked library, **bOverwrite** identifies whether to overwrite an existing library, **bLowercase** builds the file name in lowercase.

Chapter 16—Omnis Data File Migration

The Data File Migration tool is not available in some editions of Omnis Studio including the Community Edition. Omnis datafiles should not be used for new applications.

The **Omnis Data File Migration tool** lets you migrate the data in your Omnis data files to PostgreSQL or SQLite, ensuring the future stability and longevity of your OmnisSQL applications, and addressing several long-term issues with the old-style data file architecture. For example, SQL statements are no longer limited to 64KB; compound Indexes may now contain columns of different types; and various SQL parsing issues including issues with GROUP BY and ORDER BY should also be resolved.

Following a one-time conversion of your Omnis data file(s) to a PostgreSQL or SQLite database using the migration tool, the Omnis *DML commands** in your old library will execute against the selected database with no modifications to the original library code**.

The migration tool allows you to switch from an Omnis data file to a SQL database reasonably quickly and easily, without having to rewrite a lot of data handling code, which will make data storage more robust while giving you a route to convert your application to all SQL code.

*The commands that operate against Omnis data files and file classes have in the past been collectively referred to as the Omnis *Data Manipulation Language* or Omnis DML.

**In Studio 10.2 or above, emulation is enabled via two properties; $root.$prefs.$mapdmltodam and $libs.*your-lib*.$prefs.$dmlemulation. Omnis will retain these settings.

We would like to thank Nick Renders and Thad Bogert for their help in developing the DML emulation technology.

Converting Omnis Data Files

IMPORTANT: Backup Your Data Files

IN ALL CASES, YOU SHOULD MAKE A SECURE BACKUP OF ALL OMNIS DATA FILES BEFORE OPENING/CONVERTING THEM WITH THE DATA FILE CONVERSION TOOL.

The conversion option, called **Convert Data File to RDBMS,** is available under the **Tools>>Add-Ons** option on the main Omnis menubar or Toolbar, allows you to convert an existing single- or multi-segment Omnis data file into a PostgreSQL database or a SQLite data file.

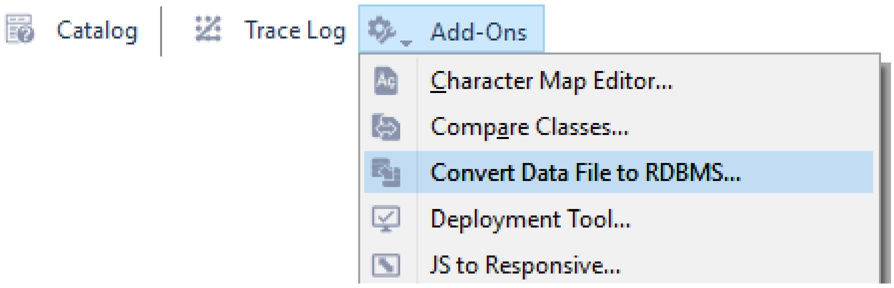
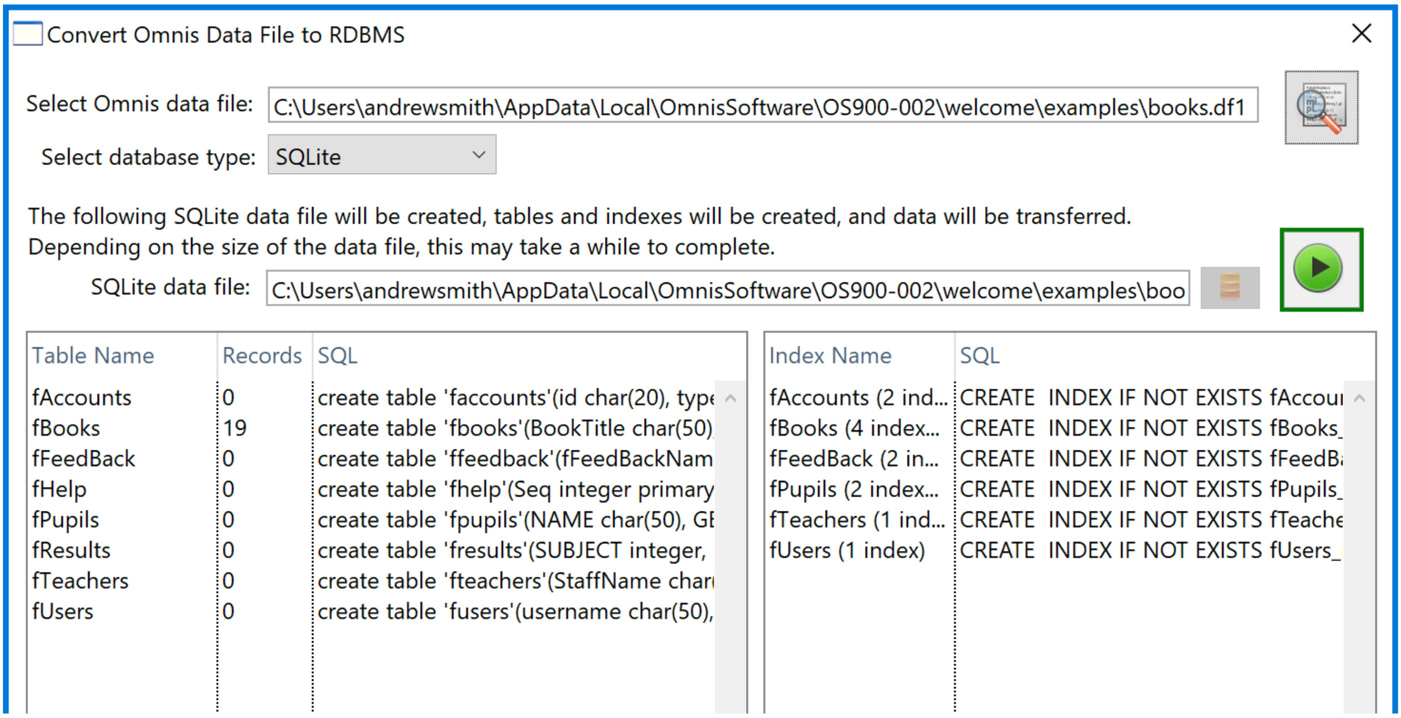


Figure 170:

To convert a data file, browse for or enter the path of the Omnis .DF1 file, select the database type (either PostgreSQL or SQLite), then press *Go*.



Convert Omnis Data File to RDBMS

Select Omnis data file: C:\Users\andrewsmith\AppData\Local\OmnisSoftware\OS900-002\welcome\examples\books.df1

Select database type: SQLite

The following SQLite data file will be created, tables and indexes will be created, and data will be transferred.
Depending on the size of the data file, this may take a while to complete.

SQLite data file: C:\Users\andrewsmith\AppData\Local\OmnisSoftware\OS900-002\welcome\examples\boo

Table Name	Records	SQL
fAccounts	0	create table 'faccounts'(id char(20), type
fBooks	19	create table 'fbooks'(BookTitle char(50)
fFeedBack	0	create table 'ffeedback'(fFeedBackNam
fHelp	0	create table 'fhelp'(Seq integer primary
fPupils	0	create table 'fpupils'(NAME char(50), GE
fResults	0	create table 'fresults'(SUBJECT integer,
fTeachers	0	create table 'fteachers'(StaffName char(
fUsers	0	create table 'fusers'(username char(50),

Index Name	SQL
fAccounts (2 ind...	CREATE INDEX IF NOT EXISTS fAccou
fBooks (4 index...	CREATE INDEX IF NOT EXISTS fBooks_
fFeedBack (2 in...	CREATE INDEX IF NOT EXISTS fFeedBa
fPupils (2 index...	CREATE INDEX IF NOT EXISTS fPupils_
fTeachers (1 ind...	CREATE INDEX IF NOT EXISTS fTeache
fUsers (1 index)	CREATE INDEX IF NOT EXISTS fUsers_

Figure 171:

The conversion process copies all tables and indexes, and populates the tables with data copied from the Omnis data file.

Once converted, you may remove the Omnis data file (and retain it as a backup).

SQL Identifiers

When converted into SQL tables, all SQL identifiers (table and column names) are converted to lower-case. Any non ISO-SQL characters, including spaces and symbols are converted to underscores. The emulator uses SQL aliases when loading values back into the corresponding File class columns.

Connected Files

During conversion any File connections are preserved using foreign key fields added to the table definition. For example, where File class A has connections to File classes B and C, table A will contain two additional integer columns named “fk_b” and “fk_c”.

During conversion, these columns will be populated with the Sequence column values for Files B and C.

For any Files that do not expose a Sequence column, the converter adds one. Using the above example, table B would be given a column named "b_seq", and table C would be given a column named "c_seq". File A's foreign key fields will then link to these.

Enabling DML Emulation

The Conversion library (omsqlconv.lbs) serves two functions. As well as converting old-style Omnis data files to SQLite or to PostgreSQL, the conversion library also works in the background providing various methods and dialogs to assist the DML emulator. Most DML emulation now occurs in the Omnis core and in the PostgreSQL & SQLite DAMs. Enabling DML emulation in Studio 10.2 is a two-step process.

Step 1 – Set $mapdmltodam

To enable DML command emulation, it is necessary to set the *$mapdmltodam* Omnis preference*, for example, for SQLite:

```
Do $root.$prefs.$mapdmltodam.$assign('SQLITEDAM')
```

or for PostgreSQL:

```
Do $root.$prefs.$mapdmltodam.$assign('PGSQLDAM')
```

or you can set it in the Property Manager under Omnis >> Prefs.
This setting then applies to any libraries subsequently enabled for DML emulation.

Step 2 – Set $dmlemulation

You also need to enable your library for emulation by setting its $dmlemulation preference to kTrue, e.g.

```
Do $clib.$prefs.$dmlemulation.$assign(kTrue)
```

or you can set it in the Property Manager under Library >> Prefs.

*Note: this behavior has changed since Studio 10.1 and now allows multiple libraries to work using the same DML emulator. Where your application uses multiple libraries, only those libraries with $dmlemulation set to kTrue will execute against the nominated database.

Note that changing the emulation mode ($mapdmltodam) while the emulator is in use will shut down the emulator, causing any open data file connections to be lost.
Likewise, $dmlemulation should not be changed whilst connections are in use. Traditional DML and emulated connections are not inter-operable.

Both of these properties are saved together with any other Omnis / Library preferences.

Emulated Commands

Once enabled, the following DML commands will be executed using the emulator, and against the specified SQL database:

- **Data files:**
  Close data file, Close lookup file, Create data file, Floating default data file, Open data file, Open lookup file, Prompt for data file, Set current data file, Set default data file, lookup().
- **Data management:**
  Build indexes, Delete data, Drop indexes, Open runtime data file browser, Rename data.
- **Changing data:**
  Cancel prepare for update, Delete, Delete with confirmation, Do not flush data, Do not wait for semaphores, Flush data, Flush data now, Prepare for edit, Prepare for insert, Prepare for insert with current values, Test for only one user, Update files, Update files if flag set, Wait for semaphores.
- **Files:**
  Clear all files, Clear main & connected, Clear main file, Clear range of fields, Clear selected files, Set main file.

- **Finding data:**
  Clear find table, Disable relational finds, Enable relational finds, Find, Find first, Find last, Load connected records, Next, Previous, Prompted find, Single file find, Test for a current record, Test for a unique index value.
- **Searches:**
  Clear search class, Reinitialize search class, Set search as calculation, Set search name, Test data with search class.
- **Sort fields:**
  Clear sort fields, Set sort field.
- **Lists:**
  Build list from file, Load from list, Replace line in list.
- **Others:**
  Begin reversible block, End reversible block, Quit all methods, Enter data, Queue OK, Queue cancel, $root.$getodbfilelist(), $cdata().$name, plus various sys() calls including sys(11), sys(82), sys(83) and sys(89).

Changes to Library Code

Aside from setting the $mapdmltodam root preference and the $dmlemulation library preference, it should not be necessary to make any changes to your library code since emulated DML commands will execute against the selected database automatically.

When enabled, the *Open data file* and *Prompt for data file* commands will automatically look for a corresponding SQLite '.db' file or a logon config file ('.dfq' or '.dfp' for SQLite or PostgreSQL respectively) in place of the exisitng data file name ('df1').

The *Create data file* command will create a new SQLite data file.

Multiple Users and Concurrency

Semaphores

When executing against a PostgreSQL database, the emulator also emulates Omnis semaphores used when locking tables prior to an insert, update or delete. A separate *semaphore thread* connects to the database and polls a special *_semaphore* table intermittently. The _semaphore table stores the lock state and update time for any replicated File classes. The _semaphore table is created automatically if it does not exist so it not detrimental to delete/update this table (for instance in the event of a deadlock).

The _semaphore table is defined with the following columns:

_tablename (varchar 255)	_timestamp (bigint)	_locked (bool)

For a given table, the _timestamp(epoch) value is only updated when an *Update files* command is executed. The semaphore thread uses this value to detect changes to the database table that may have occurred since the last *Find* command was executed. The _locked column is set to true when a *Prepare for…* command is executed. If one or more read/write Files are already locked by another client then the Prepare for… command either waits or fails depending on whether *Wait for semaphores*/*Do not wait for semaphores* is in effect. _locked is cleared either when an *Update files* or *Cancel prepare for update* is executed.
The *Test for only one user* command will also remove any stray table locks if a single database user is detected.

Note that SQLite data files can only be accessed by a single (Omnis) process. For this reason, multiple access to the data file is only possible using the SQLite Data Bridge (analogous to the Omnis Data Bridge).

When accessed directly from Omnis Studio, SQLite data files are considered to be single-user access only, i.e. semaphores are not applicable.

Using the SQL Browser

Database connections opened by emulator are displayed in the Omnis SQL Browser and normally use the *internal name* specified with the corresponding Open data file / Prompt for data file / Create data file command. You can use these sessions to execute your own SQL statements e.g. via the *Interactive SQL* utility. Avoid blocking these sessions however since this may cause emulated DML commands to hang.

Logon Config Files

Logging on to SQLite using a Config File

You can specify a logon configuration file when connecting to a SQLite database file. The logon configuration file for SQLite should have the .dfq file extension and may contain one or more session property assignments, e.g.

```
hostname=c:\Users\\myUser\mydata.db
```

If 'hostname' is not present the library uses the pathname of the .dfq file and substitutes .db.

Logging on to PostgreSQL

You need to specify a logon configuration file when connecting to PostgreSQL. The logon configuration text file (".dfp") may contain any relevant PGSQLDAM session property, for example:

```
hostname=192.168.0.10
port=5432
username=postgres
password=postgres
database=postgres
```

As mentioned, the emulator automatically substitutes the ',dfp' file extension when it encounters the Open data file command, but you can change your library code if preferred, e.g.

```
Open data file { C:\Users\MyUser\Desktop\pgconfig.dfp,internalName }
```

The *Prompt for data file* command allows you to browse for a logon configuration file. The *Create data file* command reads the logon configuration file and attempts to create the specified database.

The emulator will open a prompt if either hostname, username, or password is missing from the logon config file.

Adding Comments

Should you wish to add comments to the logon config file, the emulator ignores any line that is either empty of commences with a hash(#) character, for example:

```
hostname=c:\Users\myUser\mydata.db
## This is a comment ##
```

DML Command Logging

For basic logging and performance monitoring, you can add the following line to the logon config file:

```
logging=1
```

For additional logging of begin/end method calls and reversible blocks; add 2. For additional timing information and SQL statement logging; add 4. Setting:

```
logging=5
```

will suffice for most purposes. When enabled, logging information sent to the Omnis Trace Log so you can see output in real-time.

For low level DAM debugging, you can add the following lines to your logon config file:

```
debugfile=c:\path-to-text-file.txt   (no quotes)
debuglevel=4
debugsize=0
```

You can also access the emulator's session object via $sessions and enable/disable DAM debugging dynamically if required. In the event of technical support issues, the resulting text file can be used to diagnose faults.

Caveats

Single file find

In traditional DML, *Single file find* commands effectively move the cursor to a different point in the find table. A subsequent *Next* command would carry on from the new cursor position. Conversely, the emulator's find tables are usually filtered using a SQL WHERE clause, so there are likely to be gaps in the data. For this reason, the emulator restores the cached File contents & cursor after each Single file find command. This means that subsequent Next commands operate on the find table as it was before the Single file find command was executed, and Single file find commands now operate in isolation.

Load from list

Please note that the DML emulator is only able to load variables that are **currently in scope**. This behavior departs from traditional DML which is able to access CRB references from other methods including super and sub-class methods. This does not affect File class fields which are always in scope.

Limitations with Search Calculations

Please note that certain kinds of legacy notation may cause problems for the DML emulator. For example: *Set search as calculation F_Address.Insert_Date<dat(lDate)+1*
In the above example; *dat(lDate)+1* is an implicit (if unintuitive) instruction to add 1 day to the result of the dat() function! In this case, it is necessary to rewrite the search calculation as: *Set search as calculation F_Address.Insert_Date<dadd(kDay,1,dat(lDate))*
If you spot unexpected SQL errors (i.e. when Logging to the Trace Log is enabled), these may also be a result of other search calculations that the emulator cannot handle.

Check Data and Repair

Commands that check and alter the data file structure are largely redundant once a data file is ported to PostgreSQL or SQLite. Some commands allow existing tables to be altered for instance if you modify a File class after converting the data file.

- **Delete data** – this command invokes a DROP TABLE statement against he named table.
- **Rename data** – this command invokes an ALTER TABLE… RENAME TO… statement.
- **Drop indexes** & **Build indexes** – these commands potentially invoke multiple SQL statements to DROP or CREATE INDEXes based the underlying File class' indexed columns.
- **Update data dictionary, Reorganize data** & **Quick check** – these commands do nothing.
- **Check data** – this command compares the structure of each specified File class against its corresponding SQL table and can build a list of ALTER TABLE statements that are executed (if Perform repairs is specified). Changes detected include column name, data type, sub-type or provision for larger sub-length, changed column index state, not-null state and column description.
- **Open check data log** & **Close check data log** these commands open/close a window displaying the list of SQL operations generated by a previous call to Check data. Press Execute to run the SQL commands.
- **Clear check data log** – this command clears the list of SQL statements generated by a previous call to Check data.
- **Print check data log** – this command writes the above list of SQL statements to the Omnis trace log.

New OmnisSQL DAM

The OmnisSQL DAM has been enhanced and now contains an internal SQLite object, giving it the ability to connect to old-style Omnis data files *and* to SQLite data files. The new DAM is designed to behave identically to the old-style DAM, i.e. it supports the same properties and methods. The SQL syntax and functionality supported by the new DAM is also exactly the same, i.e. there is no support for encryption, procedures, triggers or extended ISO SQL supported by SQLite. This is done to ensure backward compatibility with Omnis SQL.

Should you wish to adopt enhanced SQLite features, you will need to modify the library further so that it uses the SQLite DAM in place of the OmnisSQL DAM.

For details of the OmnisSQL Language Definition, please refer to the OmnisSQL chapter. Please note that the new OmnisSQL DAM supports legacy data files and SQLite only. To support PostgreSQL, your application will need to be modified to use the PostgreSQL DAM in place of the OmnisSQL DAM.

Logging on to SQLite

Once converted to SQLite, you can modify your library code to connect to the SQLite data file (.db file) in place of the old-style Omnis datafile (.df1 file), e.g.

```
Do omsqlSess.$logon('/Users/myUser/mydatafile.db','','') Returns #F
```

Note that no other code changes are necessary. When the DAM encounters the '.db' file-extension, it automatically connects to the SQLite data file (and you may remove the old Omnis data file).

Chapter 17—Deployment

This chapter describes the deployment of desktop apps on Windows and macOS using the Deployment Tool. For information about deploying your web and mobile applications, created using Remote forms and the JavaScript Client, see Deploying your Web & Mobile Apps.

The **Deployment Tool** allows you to build and customize the Omnis product tree (Runtime installer) suitable for deploying your applications on Windows and macOS desktop computers. On **macOS,** the tool lets you create a bundle containing your application files, while on **Windows** it allows you to create a full or split trees.

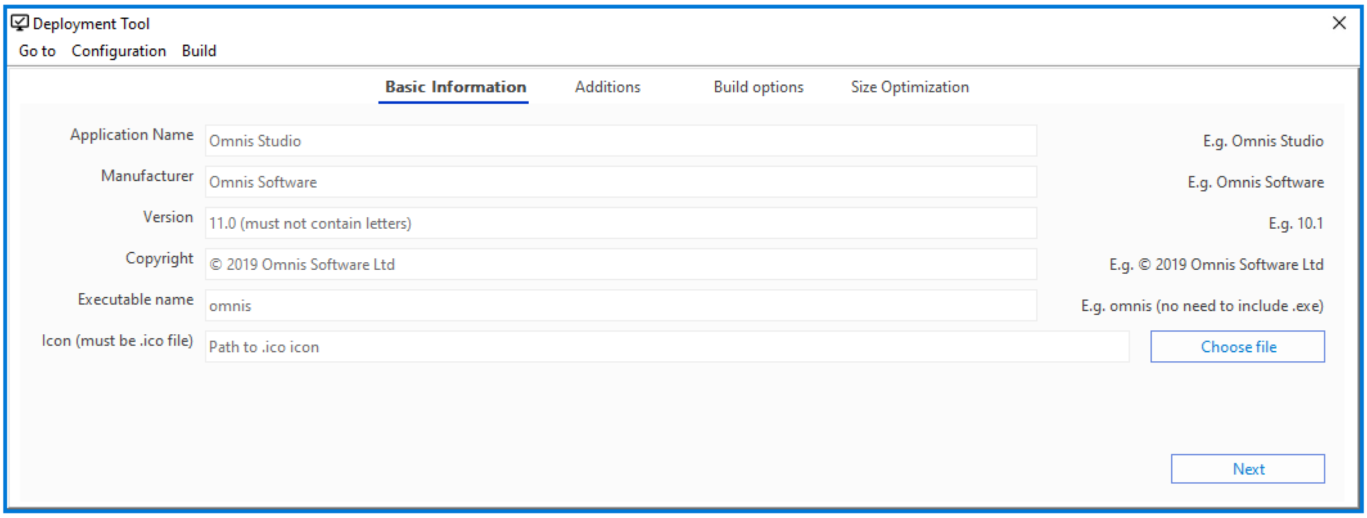
In addition, for Windows only, there is an external component, called **RCEdit,** that allows you to edit various Omnis program resources, to fully customize or 'brand' your product installation on Windows: see Windows Resource Editor.

Deployment Tool

You can open the Deployment Tool using the **Tools >> Add-ons >> Deployment tool** option on the main Omnis menu bar. The tool is system dependent, so there are minor differences between the capabilities and interface across Windows and macOS which are described below.

Windows

On Windows, on the **Basic Information** screen you can enter your Application Name, the Manufacturer, Version, Copyright notice, Executable name and the path to an .ico file to replace the Omnis icon.



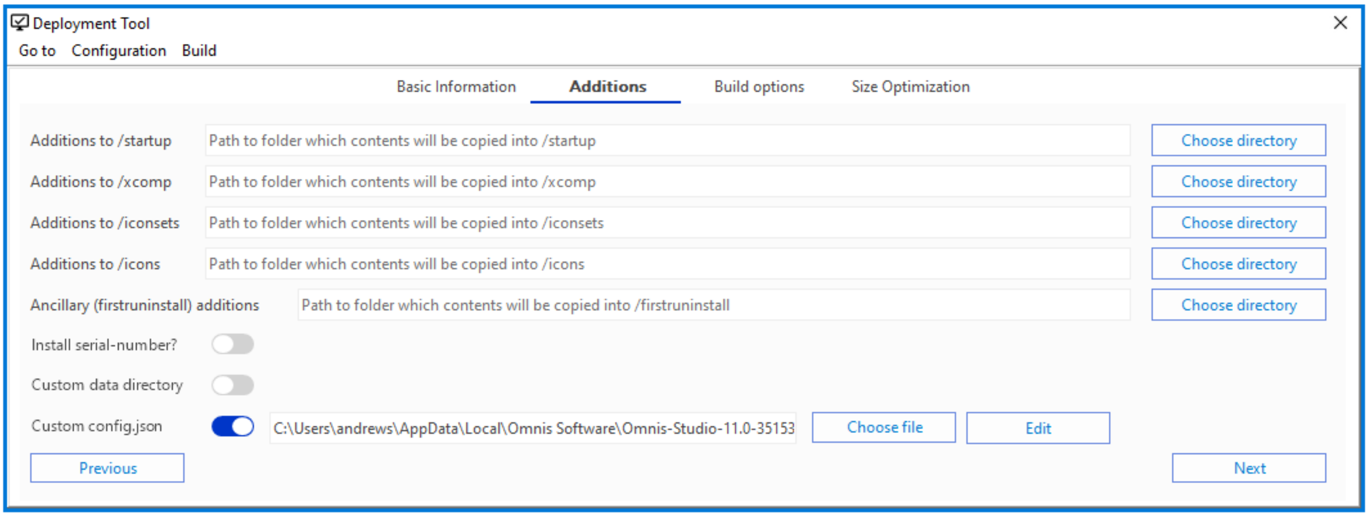
Application Name	Omnis Studio	E.g. Omnis Studio
Manufacturer	Omnis Software	E.g. Omnis Software
Version	11.0 (must not contain letters)	E.g. 10.1
Copyright	© 2019 Omnis Software Ltd	E.g. © 2019 Omnis Software Ltd
Executable name	omnis	E.g. omnis (no need to include .exe)
Icon (must be .ico file)	Path to .ico icon	

Figure 172:

The **Additions** screen lets you specify the bundle's startup folder, the xcomp folder (containing any external components), iconsets and icons folder (any icons for the library), plus the location of any files you want to add to the **firstruninstall** folder (see Build options).

The **Install serial number** option allows you to add an Omnis Runtime (client license) serial number to pre-serialize the bundle. If this option is used, a file called serial.txt will be added and deployed to the end user's read-write folder (e.g. the AppData folder) containing the Omnis serial number: the file has a single line in the format SN=xxxxxx where xxxxxx is the Omnis Runtime serial (client license) number. (Note if you are installing the Omnis Server version and wish to pre-serialize it, you can use a serial.txt file in the Omnis root folder. If you wish to set the port number you can set it in the config.json in the server section.)

The **Custom data dictionary** option allows you to add a custom read/write directory. The **Custom config.json** option allows you to specify an Omnis configuration file containing your own settings. A customtool folder is created within the installed writable directory of Omnis containing a copy of the config.json file from the current instance of Omnis. From there you can edit the configuration file for your application to build. Alternatively, you can select an existing config.json to use for your application.



Additions to /startup: Path to folder which contents will be copied into /startup

Additions to /xcomp: Path to folder which contents will be copied into /xcomp

Additions to /iconsets: Path to folder which contents will be copied into /iconsets

Additions to /icons: Path to folder which contents will be copied into /icons

Ancillary (firstruninstall) additions: Path to folder which contents will be copied into /firstruninstall

Install serial-number? ○

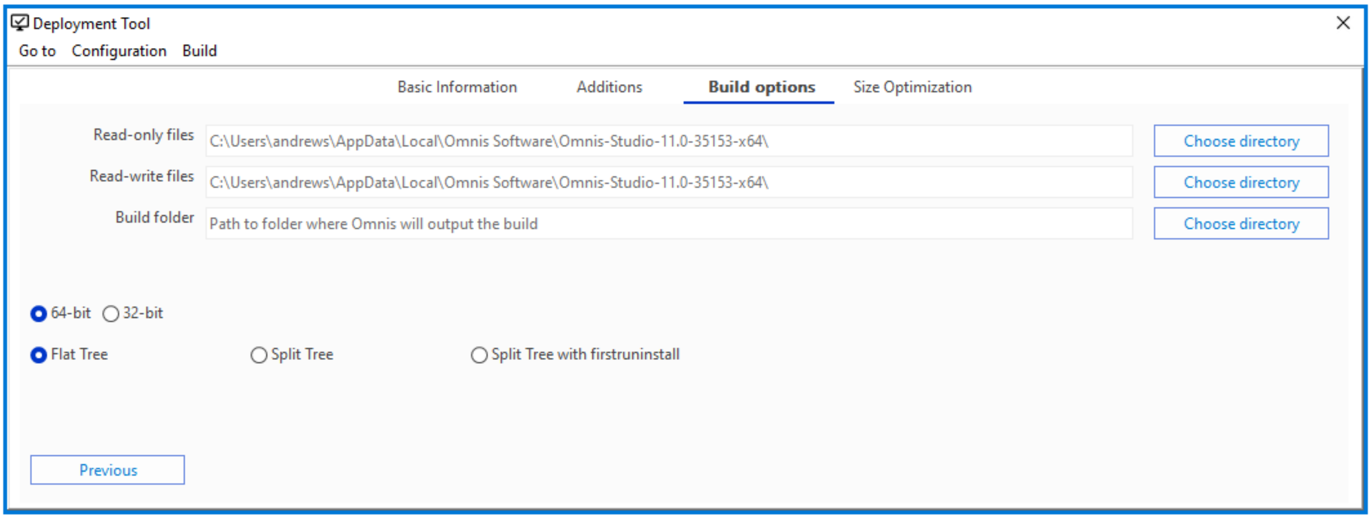
Custom data directory ○

Custom config.json ● C:\Users\andrews\AppData\Local\Omnis Software\Omnis-Studio-11.0-35153

Figure 173:

On the **Build options** screen you need to specify the location of the Omnis **Read-only files** (these are usually your Omnis Studio xx.x RT folder in your ProgramFiles), the **Read-write files** (usually the files in AppData). The path specified in the **Build folder** field is the path for the output.

You can also select the option to clear the build folder if there is any error in the build process, plus the option for 32 or 64 bit.



Read-only files: C:\Users\andrews\AppData\Local\Omnis Software\Omnis-Studio-11.0-35153-x64\

Read-write files: C:\Users\andrews\AppData\Local\Omnis Software\Omnis-Studio-11.0-35153-x64\

Build folder: Path to folder where Omnis will output the build

● 64-bit ○ 32-bit

● Flat Tree ○ Split Tree ○ Split Tree with firstruninstall

Figure 174:

On Windows, you can also select the option to build a folder or separate folders providing a Flat tree, Split tree or a Split tree with firstruninstall, as follows:

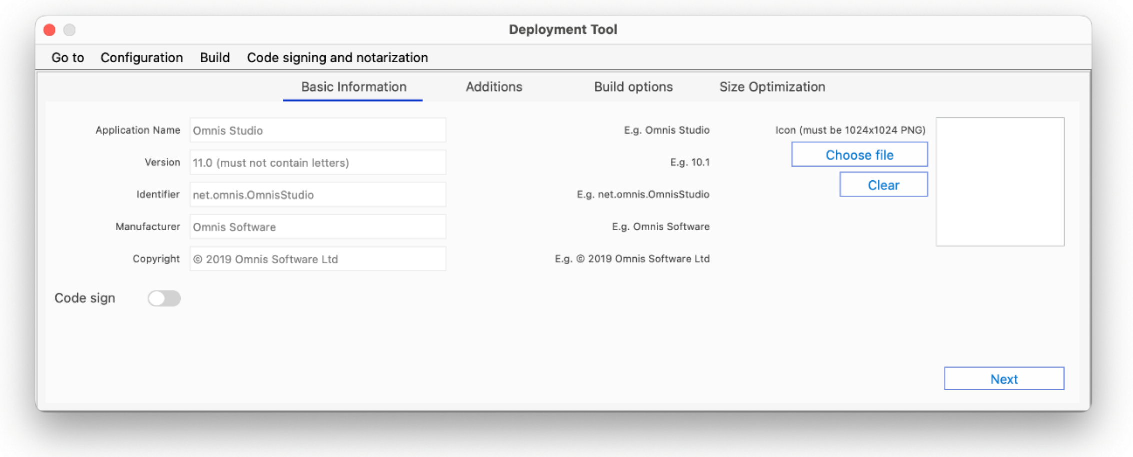
- The **Flat tree** option will output a folder containing both read and read/write files.
- The **Split tree** option will output a folder containing read-only files, to go in the Program Files folder, and a read-write folder containing files for the AppData folder.
- The **Split Tree with firstruninstall** option will output a folder containing read-only files and inside that an additional “firstruninstall” folder, containing the read-write files that Omnis will copy the first time the application will be run.

The **firstruninstall** option allows you install and setup your application without the need to build an installer, which may be quicker or more convenient for your deploy process or product cycle. For example, you could use 7Zip's SFX archive feature to create an executable which simply unarchives itself in such a way to make installers and the complexities they come with unnecessary.

The **Size Optimization** tab will give some information regarding the estimated size before the build and the estimated size saved. You can remove files or folders during a build by specifying them on this tab. When adding files or folders to be removed, you only need to specify the relative path to the file or folder inside the readonly/readwrite directory.

macOS

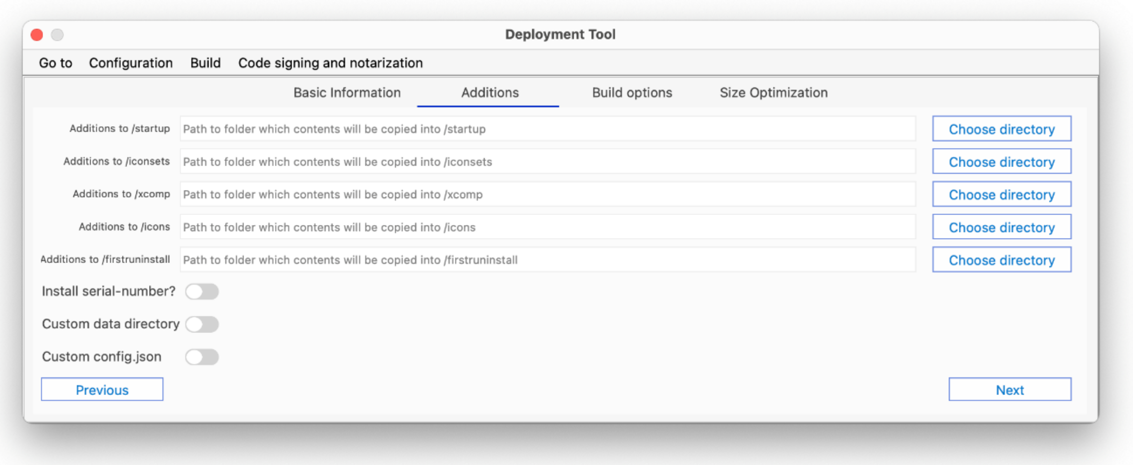
For macOS, you can change the Application Name, Version, Identifier, Manufacturer, Copyright notice and Icon in the first screen:



Application Name: Omnis Studio — E.g. Omnis Studio
Version: 11.0 (must not contain letters) — E.g. 10.1
Identifier: net.omnis.OmnisStudio — E.g. net.omnis.OmnisStudio
Manufacturer: Omnis Software — E.g. Omnis Software
Copyright: © 2019 Omnis Software Ltd — E.g. © 2019 Omnis Software Ltd
Icon (must be 1024x1024 PNG)
Code sign

Figure 175:

The second screen allows you to specify the bundle's startup folder, iconsets (for the library), xcomp and icons folders, as well as the option to pre-serialise the bundle or add a custom read/write directory.



Additions to /startup: Path to folder which contents will be copied into /startup
Additions to /iconsets: Path to folder which contents will be copied into /iconsets
Additions to /xcomp: Path to folder which contents will be copied into /xcomp
Additions to /icons: Path to folder which contents will be copied into /icons
Additions to /firstruninstall: Path to folder which contents will be copied into /firstruninstall
Install serial-number?
Custom data directory
Custom config.json

Figure 176:

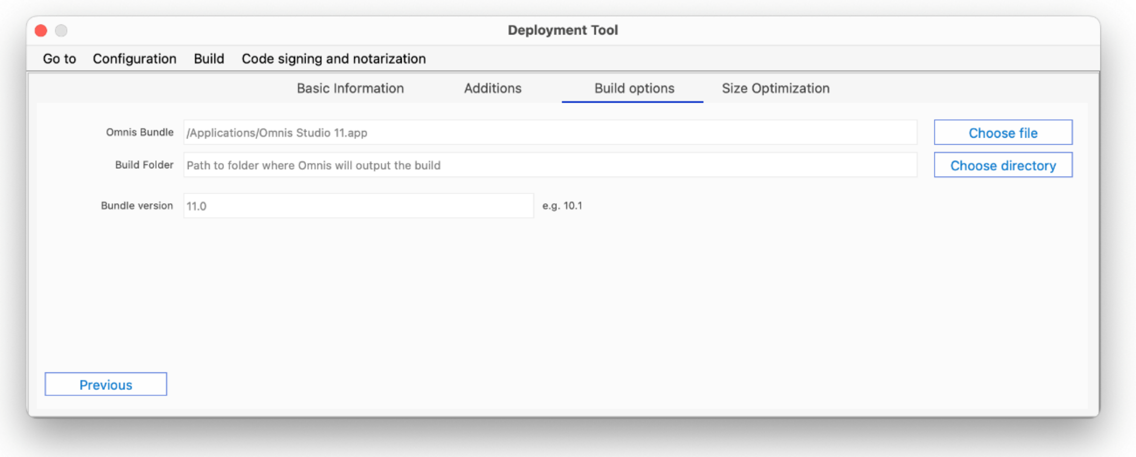
On the Build options screen you need to select the **Omnis Bundle** (runtime), a **Build folder** (the path to the folder where you want your output to go), and the version of your Bundle. The **Go to** menu allows you to see the build folder in the system file explorer.

You can also select the option to clear the build folder if there is any error in the build process.

The **Size Optimization** tab will give some information regarding the estimated size before the build and the estimated size saved. You can remove files or folders during a build by specifying them on this tab. When adding files or folders to be removed, you only need to specify the relative path to the file or folder inside the bundle.

Code Signing and Notarizing (macOS)

On macOS, you can **Code sign** and **Notarize** your application using the Deployment tool, which will make deployment easier and faster. *See the next section in this chapter for more information about Code Signing Omnis manually.*



Omnis Bundle: /Applications/Omnis Studio 11.app

Build Folder: Path to folder where Omnis will output the build

Bundle version: 11.0 e.g. 10.1

Figure 177:

To Code sign and Notarize you need an app-specific password, and this can be obtained by logging into your Apple ID account, via https://appleid.apple.com/account/manage and selecting "Generate password…" under APP-SPECIFIC PASSWORDS. Once generated, copy your app-specific password.

Next, create a new Keychain Access password with the name AC_PASSWORD and paste the password you just copied as the password and your Apple developer email address for the account, as follows:

You will also need a developer certificate in Keychain Access that can be used to code sign and notarize. You can obtain one from your Apple developer account and you can add it to the Keychain Access by double-clicking on the certificate.

Now using the Deployment tool, you need to enable the **Code sign** option on the Basic Information tab to display the Notarize options.

This will open options which allow you to enter the Code signing identity. Code signing is required before notarizing and the code signing identity should be the name of your certificate as it comes from Keychain Access, such as:

Developer ID Application: Omnis Software Ltd (XYZ123XYZ123)

Once you enable **Code sign,** you can also enable the **Notarize** option which displays a new entry field for your email and a new tab that handles entitlements.

Your email is used when submitting your application to Apple for notarization and you will receive an update from Apple on the status of notarization on the same email.

The "Notarization options" tab can be used to toggle application entitlements, which tell Apple more about what your application will need access to. You need to supply some defaults which reflect what you distribute, and if you need more access, you can just cheque the appropriate option.

Once you are ready to build, just can use the **Build** menu or the shortcut Cmd+B on macOS or Ctrl+B on Windows.

If you Code sign and Notarize as part of the build, Omnis may be unresponsive for quite a time, since a number of things need to happen: existing code signatures are removed, new code signatures are applied, a DMG is built and submitted to Apple, and so on. During this process, you need to leave Omnis open, and once the process is finished you will be prompted if it was successful or not.

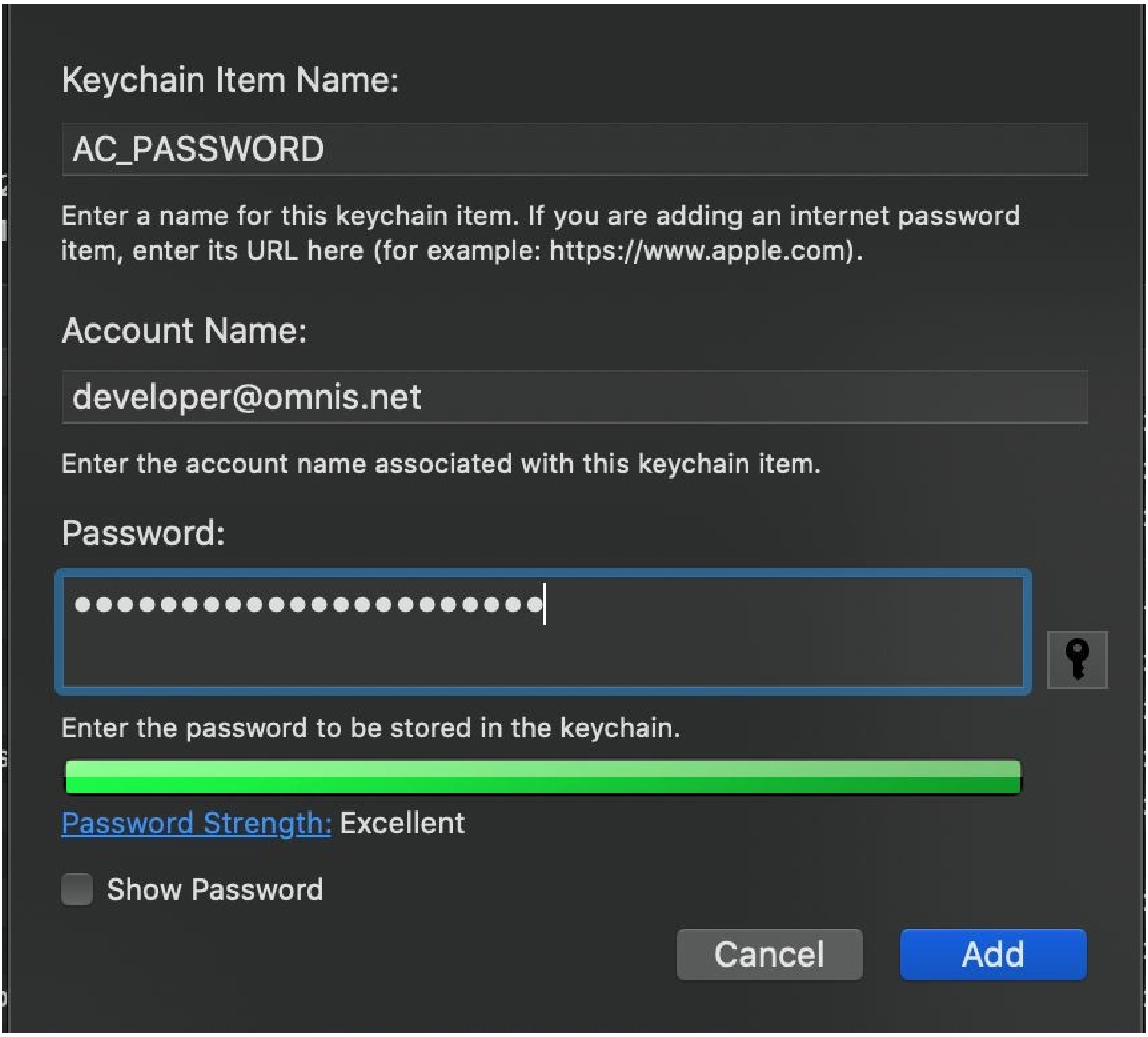
Once you have successfully notarized your application, you need to stapler either the DMG or the application bundle before distributing. You can use the "Staple an existing Bundle or DMG" from the Code signing and notarization menu.

You can also check if a DMG or application bundle is successfully stapled via the terminal by executing "stapler validate [path to dmg or app]".

Deployment Tool API

In Studio 11, a number of methods have been exposed in the Deployment Tool API to allow you to manage builds in your own code, rather than via the Deployment Tool UI. Using the method $root.$modes.$getapiobject("customtool") Returns iObRef a number of API calls for the Deployment Tool have been exposed.

**$setcallbackinst**($cinst) takes a reference to an instance that implements $completed and $error. If the callback instance is set via this method, the deployment tool API will call either $completed or $error instead of returning the outcome to the caller. Note $error also receives a character variable as a parameter containing the error message and $completed can receive a 36-character long string on macOS if the built bundle is submitted for notarization.



Keychain Item Name:

AC_PASSWORD

Enter a name for this keychain item. If you are adding an internet password item, enter its URL here (for example: https://www.apple.com).

Account Name:

developer@omnis.net

Enter the account name associated with this keychain item.

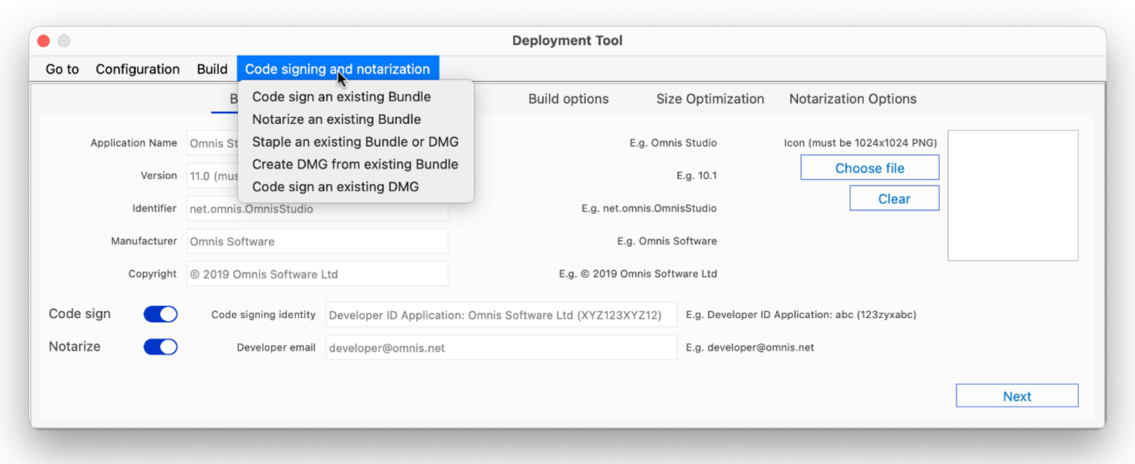
Password:

●●●●●●●●●●●●●●●●●●●●●●

Enter the password to be stored in the keychain.

Password Strength: Excellent

Figure 178:



Deployment Tool
Go to
Configuration
Build
Code signing and notarization
Code sign an existing Bundle
Notarize an existing Bundle
Staple an existing Bundle or DMG
Create DMG from existing Bundle
Code sign an existing DMG
B
Build options
Size Optimization
Notarization Options
Icon (must be 1024x1024 PNG)
Choose file
Clear
Next

Application Name: Omnis St — E.g. Omnis Studio

Version: 11.0 (mus — E.g. 10.1

Identifier: net.omnis.OmnisStudio — E.g. net.omnis.OmnisStudio

Manufacturer: Omnis Software — E.g. Omnis Software

Copyright: © 2019 Omnis Software Ltd — E.g. © 2019 Omnis Software Ltd

Code sign ● — Code signing identity: Developer ID Application: Omnis Software Ltd (XYZ123XYZ12) — E.g. Developer ID Application: abc (123zyxabc)

Notarize ● — Developer email: developer@omnis.net — E.g. developer@omnis.net

Figure 179:

$run(cConfigFilePath,cError[,cUUID]) requires the path to the deployment configuration file, a character variable to return errors to, or if on macOS, the UUID when the build is submitted for notarization. If successful, the method returns kTrue, otherwise kFalse is returned. The configuration file can be built using the GUI version of the Deployment tool.

Managing Builds via the API

The Deployment Tool API supports builds with in-memory data structures rather than file-based only. You can get the data structures, load and save to a config.json file programmatically.

$getBuildDataStructure() returns a row containing the main data structure for a cross-platform build.

$getEntitlementsDataStructure returns a row containing two row: the standard and extended entitlements data structures (in that order).

$loadConfig(cPathToConfig.json, rBuildDataStructure, rEntitlements, cErrorText) takes in the path to a config.json containing data structures, a row that receives the build data structures, a row that receives and entitlements rows and a character variable that receives and error text. Returns true if successful, otherwise false.

$setBuildData(rBuildData) sets the build data structure in-memory to rBuildData row, or you can use $run without passing in the path to a build config.json in order to use the in-memory data values.

$setEntitlementsData(rEntitlements) sets the entitlements data structure in-memory to rEntitlements row (note the rEntitlements row must contain two rows where the first is standard entitlements and the second is extended entitlements. Works only on macOS.

$saveConfig(cPathToFile, bOverwrite, cErrorText) saves the build data and entitlements data currently stored in the API object to a .json file in cPathToFile. If cPathToFile already exists, bOverwrite (defaults to false) will be used to determine if the file should be overwritten. cErrorText receives any errors if unsuccessful; function returns true if successful, otherwise false.

The $run function can be run without passing the path to a build config.json file, e.g. Do api.$run("",cError, cUUID) Returns bOutcome as long as the build data has been set via $setBuildData, the build can start. You can use $run by passing the path a build config.json and the build/entitlements stored in the file will be used.

Code Signing Omnis

The Omnis Studio application package on macOS is code signed, which provides increased security for you and your end users. A signed application can be trusted to originate from the developer who signed it, and to not have been altered in any way by any third-party, therefore guaranteeing the authenticity of an application. Signed applications within macOS can automatically be granted permissions to perform actions, such as accessing services from the network and running built-in software such as AppleScript commands.

An application can only be signed if its code portion remains unchanged. For the Omnis application, the code portion is located in the Omnis package, e.g.:

```
Omnis\ Studio\ 11\ x64.app/Contents/MacOS/
```

Firstruninstall and Application Support folders

Any files that are updated by Omnis must be stored as user application data located in the user's home directory, that is, in the Application Support folder:

```
~/Library/Application Support/Omnis/
```

To do this, when Omnis starts up it will check for the existence of a folder called 'firstruninstall' in the macOS folder in the Omnis package. Any items which are contained in this folder are copied by default to a folder in Application Support with the same name as the Omnis package, e.g.:

```
~/Library/Application Support/Omnis/Omnis Studio 11 x64
```

The copy will not occur if the destination folder already exists, therefore avoiding any files being overwritten.

This provides a mechanism to place all data folders and their contents into the 'firstruninstall' folder, e.g. icons, studio, startup. Once copied into Application Support they are only updated in that location and leave the original macOS folder unchanged and its signature valid.

Updating Components

With the signed version of Omnis Studio, an external or JavaScript component can be added or updated in the user data folder. This allows the signed code part of Omnis to remain unaltered, so it maintains a valid code signature. For example, a standard component can be placed in the following folder:

```
~/Library/Application Support/Omnis/\Omnis\ Studio\ 11\ x64/xcomp
```

and a JavaScript component here:

```
~/Library/Application Support/ Omnis/\Omnis\ Studio\ 11\ x64/jscomp/
```

If the required folder does not exist it can be created by the user.

The user data folder is always searched first, so if a component with the same name exists in the code section of the Omnis tree the user version will be loaded in preference.

Deployment

When deploying your own application, you can update the distributed files in the Omnis package to include your own libraries and components and to edit the name of the application. Those files placed in the firstruninstall folder will be treated as user data and will be copied to the Application Support folder.

By default, user data for each installation of Omnis goes into a subfolder of Application Support called “Omnis” and the name of the Omnis package is used to provide the folder for the individual installation.

So for example an installation here:

```
/Applications/Omnis Studio 11 x64.app
```

Will have a default user data location of:

```
~/Library/Application Support/Omnis/Omnis Studio 11 x64
```

To customize the subfolder, edit resource 25599, and to customise the installation folder, edit resource 25600. These resources are located in the Localizable.strings file for the language used, e.g.

```
/Omnis Studio 11 x64.app/Contents/Resources/English.lproj/Localizable.strings
```

Both entries are empty for default behavior.

```
"CORE_RES_25599" = "";
"CORE_RES_25600" = "";
```

After you update the Omnis package files, the package will need to be re-signed with your own signing identity. You cannot sign a file that has extended Finder information attributes, so these need to be removed before signing. This can be done recursively over the entire package by using the following command:

```
xattr -r -d com.apple.FinderInfo <package_path>
```

For example:

```
xattr -r -d com.apple.FinderInfo /Applications/My\ Application.app
```

Signing you own application requires a code signing identity which can be generated by adding a development or production certificate via the Certificate section of the Apple developer member center. The machine where signing is to occur must have the certificate and private key installed. To list all valid code signing identities available on a machine, use the following command from the terminal:

```
security find-identity -p codesigning -v
```

Which will, for example, produce the following output with key and identity listed:

```
1) 44FFBA8B7DFFB1AFFF36FD0613D6E5FC61FF8DFF "Certificate" (CSSMERR_TP_NOT_TRUSTED)
2) B3EF62FF18E0FFB83D3A8FF3672CF80EFF367FFF "Mac Developer: John Doe (24FFEXFF39)"
2 valid identities found
```

To sign the package use:

```
codesign -f --deep --verbose -s <identity> <package_path>
```

For example:

```
codesign -f --deep --verbose -s "Mac Developer: John Doe (24FFEXFF39)" /Applications/My\ Application.app
```

If the command completes with no errors, a similar line to the following should appear in the output:

```
:signed app bundle with Mach-O thin (x86_64) [com.myCompany.MyApplication]
```

The application is now signed and ready for deployment.

Do not subsequently alter the contents of the package as this will invalidate the signature.

You can verify the signature using the following:

```
codesign --display --verbose=4 <package_path>
```

Which will list items such as the signing authority, signing time, etc.

Patching a signed tree

If you wish to distribute an updated Omnis application (the program file), and replace the application in an existing signed Omnis tree, then this can be achieved by doing the following:

- Replace the binary in the original signed tree with the new version.
- Re-sign the Omnis tree with the same signing identity which you used to sign the original tree.
- Take the patched binary out of the tree for distribution.

Components can be patched without re-signing into the xcomp and jscomp folders of the user data location, e.g.:

```
~/Library/Application Support/Omnis Studio 11 x64
```

Always ensure the tree has a valid signature by running:

```
codesign --display --verbose=4
```

Omnis data folder

Resources 25599 and 25600 can now be used to specify the Omnis data folder on the Windows platform. You can edit the omnisdat.dll string table with a resource editor and modify 25599 and 25600 to be used to specify the sub-folders of the appdata directory. The Omnis data folder becomes:

```
<appdata folder>\<resource 25599>\<resource 25600>\
```

If resource 25599 is not empty, resource 25600 must also not be empty.

Update Manifest Files

Omnis employs an update mechanism to update files in the user data contents of an Omnis installation, that is inside Application Support on macOS and AppData on Windows.

When Omnis starts, it reads the contents of the 'version' file in the root of its installation files, that is '/Application/Omnis Studio 11.app/Contents/MacOS/version' on macOS and 'C:\Program Files\Omnis Software\Omnis Studio 11\version' on Windows. If that file is not present, the Omnis internal build version is used, as returned by sys(123). The version file in the read-only location is referred to as 'deployment version'.

Similarly, Omnis retrieves the revision of another hidden .version file in the root of the user-data location, e.g. [path to application support]/Omnis/Omnis Studio 11/.version on macOS or [path to appdata]\Omnis Software\Omnis Studio 11\.version on Windows. If that file is not present, 0 is assumed to be the version of the user data. The .version file in the read-write location is referred to as 'data version' or 'user data version'.

If the data version is lower than the deployment version, Omnis will check for updates that need to be applied in a special folder named "manifest" in the root of the read-only location: this manifest folder will contain files named after the new deployment version, e.g. 23071 or 23072. Inside these manifest files, there are paths of files or folders to remove from the user data location.

Omnis will remove those from the user data and then the firstruninstall mechanism will copy whatever is missing into the user data location, and the .version file in the user data will be updated accordingly.

Update on macOS

On macOS if an Omnis deployment is replacing an existing installation using a simple drag and drop approach, that is, from a disk image to the Applications folder, files which already exist in the current user's Application Support folder will not be updated from the files in the firstruninstall folder of the new disk image.

If there are files which need to be patched, Omnis provides the use of *Update Manifest Files* to allow a deployment to specify the files in an existing set of user data which need to be removed so they can be replaced by newer files from the firstruninstall folder.

When Omnis starts it will read an integer deployment version number from a file called "version" in the Omnis application's macOS folder:

```
/Applications/Omnis\ Studio\ 11.0.1.app/Contents/MacOS/version
```

If this file does not exist then the deployment version number will be set to the Omnis internal build number (as returned by sys(123)).

Omnis will read the version of the user data from a hidden file (.version) in the root of the user's application data folder for the Omnis application.

```
/Users/<username>/Library/Application Support/Omnis/Omnis Studio 11.0.1/.version
```

If this file does not exist, the user data version is set to zero. If the user data version is lower than the deployment version, Omnis will check for updates that need to be applied.

The updates are specified in a set of files which should be placed in a folder called "manifest" within the Omnis application's macOS folder. Each file should be named for the version which specifies the changes. For example, if the new deployment is version 23071 there should be a file named 23071.

```
/Applications/Omnis\ Studio\ 11.0.1.app/Contents/MacOS/manifest/23071
```

The manifest file should contain the paths of each file or folder which needs to be updated for that version. Each path should be separated by a new line.

Therefore, if file 23071 contains:

```
studio/v40.lbs
startup/vcs.lbs
```

This will indicate that the Studio Browser and VCS are to be updated (removed from the user data) for version 23071.

There can be a manifest file for interim versions if updating an older version of Omnis. Therefore, if the user version is 23069, the deployment version is 23071 and the manifest folder contains 23070 and 23071 then both files will be used for updating.

If updates are applied, the hidden user data version is updated to the deployment version.

Note that the Omnis application deployment tree (code-side) still needs to be re-signed (and notarised) for each version of a deployment.

It is not recommended that individual files are updated in an existing signed tree (as this will invalidate the signature).

Windows Resource Editor

The **Windows Resource Editor,** or RCEdit, allows you to edit various Windows resources, including the product version, version string, and the icon for the Omnis executable. The external component implements a number of functions (methods) which you can call from your Omnis code using Do rcedit.$<function-name>(*parameters*).

Setting the application manifest

$setapplicationmanifest(cFile, cManifest) sets the manifest in cManifest to file path cFile. For example:

```
Do rcedit.$setapplicationmanifest("C:\omnis.exe", "C:\folder\newManifest.xml") Returns #F
```

Setting a resource string

$setresourcestring(cFile, cResource, cValue) sets resource in cResource to value in cValue for file path cFile. For example:

```
Do rcedit.$setresourcestring("C:\omnis.exe", "1", "This is the new value") Returns #F
```

Setting the product version

$setproductversion(cFile, cProductVersion) sets the Windows executable resource "Product Version" to version in cProductVersion for file path cFile. For example:

```
Do rcedit.$setproductversion("C:\omnis.exe", "11.1")
```

Setting the Omnis icon

$seticon(cFile, cIcon) sets the Windows executable icon in file path cFile to .ico file path in cIcon. For example:

```
Do rcedit.$seticon("C:\omnis.exe", "C:\newIcon.ico") Returns #F
```

Setting the program version string

$setversionstring(cFile, cVersion, cValue) sets the version string in cVersion to value in cValue for file path in cFile. For example:

```
Do rcedit.$setversionstring("C:\omnis.exe", "Comments", "Comment version string") Returns #F
```

Setting the program file version

$setfileversion(cFile, cFileVersion) - sets the Windows executable resource "File Version" to value in cFileVersion. For example:

```
Do rcedit.$setfileversion("C:\omnis.exe", "11.1.0.0") Returns #F
```