Contents

Extending Omnis **4**

About This Manual 4

Chapter 1—Web Services **5**

What is REST? 6

Creating a Web Services Client 7

Creating your own Web Services 12

Cross Origin Resource Sharing 23

Logging 24

Authentication 24

Chapter 2—OJSON **26**

Data Structure and Addressing 26

Static Methods 27

JSON External Component Object 28

Chapter 3—Java Objects **31**

Setting Up 31

Creating Java Objects 35

Subclassing Java Objects 38

Using Java Objects 38

Method Overloading and Pattern Matching 45

Calling Overloaded Methods Directly 48

Nested Object Arrays 48

Modifying The System Package List 49

Overloaded Types 49

Frequently Asked Questions 51

Chapter 4—Omnis .NET Objects **51**

Introduction 51

Software Requirements 51

Setting up 52

Creating .NET Objects 53

Subclassing .NET Objects 56

Using .NET Objects 56

.NET Objects example library 63

Method Method Overloading and Pattern Matching 65

Nested Object Arrays 69

Overloaded Types 70

Chapter 5—oXML 71
About oXML . . . 71
What is XML? . . . 71
Creating a Document Object . . . 76
Document Objects in oXML . . . 79
Manipulating XML Documents . . . 79
Creating XML documents . . . 85

Chapter 6—oProcess 87
About oProcess . . . 87
Properties . . . 88
Methods . . . 88
Using oProcess . . . 90

Chapter 7—OW3 Worker Objects 90
Example Apps . . . 91
Using the OW3 Workers . . . 91
HTTP/2 support . . . 91
Base Worker Properties . . . 91
Base Worker Constants . . . 92
Base Worker Methods . . . 92
OAUTH2 Worker Object . . . 95
HTTP Worker Object . . . 101
SMTP Worker Object . . . 107
FTP Worker Object . . . 111
IMAP Worker Object . . . 116
JavaScript Worker Object . . . 121
POP3 Worker Object . . . 124
CRYPTO Worker Object . . . 126
HASH Worker Object . . . 127
LDAP Worker Object . . . 129
Python Worker Object . . . 130
Java Worker Object . . . 130
Web Worker Objects . . . 134
External Commands . . . 142

Chapter 8—Omnis Graphs 144
About Graph2 . . . 144
Chart Types . . . 145
Common Graph Properties . . . 149
Common Graph Methods . . . 150
XY Charts . . . 151

Pie Charts . . . 158
Polar Charts . . . 160
Meter Charts . . . 163
Graph Layers and the Prelayout Event . . . 167
Graph Clicks and Drilldown . . . 168
Changing the Color of Graph elements . . . 168
Adding Colored Zones . . . 169
Parameter Substitution and Formatting . . . 169
Labels . . . 174
Using Graphs in Reports . . . 175
Using Graphs in the Web client . . . 175

Chapter 9—Remote Studio Applet **178**
How does it work? . . . 178
Object Interfaces . . . 178
Studio Remote Tasks . . . 179
Remote Studio Examples . . . 180

Chapter 10—Automation **182**
Instantiating an Automation Server . . . 182
Automation Server Functionality . . . 183
Built-in Methods . . . 184
Lifetime of an Automation Server Instance . . . 184
Automation Event Handling . . . 185
Automation to Omnis Variable Conversion . . . 185
Automation Errors and Limitations . . . 186
Automation Examples . . . 186

Chapter 11—Apple Events **188**
Apple Events Object . . . 188
Apple Event Methods . . . 188

Chapter 12—Omnis ODBC Driver **189**
Enable ODBC Access . . . 189
Download and Install the Driver . . . 193
Configure ODBC DSNs . . . 193
Testing the DSN . . . 195
Using SQL . . . 195

Chapter 13—Blowfish Encryption **197**
About Blowfish . . . 197
Padding . . . 197
Binary Encryption . . . 198
String Encoding . . . 198

Extending Omnis

Omnis Software Ltd

Released May 2023
Updated Jun 2023 Revision 35439
Updated Oct 2023 Revision 35659

About This Manual

This manual describes all the features in Omnis Studio that allow you to extend the capabilities of Omnis Studio for creating full-featured enterprise and multi-tier applications. For example, it covers topics as diverse as Web Services, using the OJSON component, and using the OW3 Worker Objects.

You should read the Omnis Programming manual before this one, to learn about the general tasks and techniques required for creating Omnis applications. In addition to this manual, there are the *Omnis Reference* manuals, and a comprehensive Help system describing the Omnis Studio commands, functions, and the notation, available from within the Omnis Studio development environment using the F1 key.

Copyright info

The software this document describes is furnished under a license agreement. The software may be used or copied only in accordance with the terms of the agreement. Names of persons, corporations, or products used in the tutorials and examples of this manual are fictitious. No part of this publication may be reproduced, transmitted, stored in a retrieval system or translated into any language in any form by any means without the written permission of Omnis Software.

© Omnis Software, and its licensors 2023. All rights reserved.

Portions © Copyright Microsoft Corporation.

Regular expressions Copyright (c) 1986,1993,1995 University of Toronto.

© 1999-2023 The Apache Software Foundation. All rights reserved.

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

Specifically, this product uses Json-smart published under Apache License 2.0 (http://www.apache.org/licenses/LICENSE-2.0)

© 2001-2023 Python Software Foundation; All Rights Reserved.

The iOS application wrapper uses UICKeyChainStore created by http://kishikawakatsumi.com and governed by the MIT license.

Omnis® and Omnis Studio® are registered trademarks of Omnis Software.

Microsoft, MS, MS-DOS, Visual Basic, Windows, Windows Vista, Windows Mobile, Win32, Win32s are registered trademarks, and Windows NT, Visual C++ are trademarks of Microsoft Corporation in the US and other countries.

Apple, the Apple logo, Mac OS, Macintosh, iPhone, and iPod touch are registered trademarks and iPad is a trademark of Apple, Inc.

IBM, DB2, and INFORMIX are registered trademarks of International Business Machines Corporation.

ICU is Copyright © 1995-2023 International Business Machines Corporation and others.

UNIX is a registered trademark in the US and other countries exclusively licensed by X/Open Company Ltd.

Portions Copyright (c) 1996-2023, The PostgreSQL Global Development Group

Portions Copyright (c) 1994, The Regents of the University of California

Oracle, Java, and MySQL are registered trademarks of Oracle Corporation and/or its affiliates

SYBASE, Net-Library, Open Client, DB-Library and CT-Library are registered trademarks of Sybase Inc.

Acrobat is a registered trademark of Adobe Systems, Inc.

CodeWarrior is a trademark of Metrowerks, Inc.

This software is based in part on ChartDirector, copyright Advanced Software Engineering (www.advsofteng.com).

This software is based in part on the work of the Independent JPEG Group.

This software is based in part of the work of the FreeType Team.

Other products mentioned are trademarks or registered trademarks of their corporations.

Chapter 1—Web Services

The Web Services component provides support for RESTful web services for client and server. The component instantiates a Web Worker Object with properties and methods based on the type of web service you are using. You must create the client interface (remote forms or window classes etc) to the Web Service Object.

In addition, the Web Server plug-in allows the Omnis App Server to expose your Omnis code as a RESTful Web Service. This is described under the Creating your own Web Services section.

In addition, there is a JSON external component, called OJSON, that allows JSON based objects returned from RESTful resources to be manipulated: this is described in the OJSON chapter.

From Studio 8.1 onwards, you no longer need to serialize Omnis with a separate Web Services plug-in serial number in order to develop a Web Services client or to consume a Web Service. If you are creating your own Web Services, from your Omnis code, your server will still require an Omnis App Server deployment license when you are ready to deploy your app.

Note that the old WSDL-based Web Services component available in versions prior to Studio 8.x is no longer supported in Omnis and has been removed.

What is REST?

REST (Representational State Transfer) is the predominant architectural style that is used to consume and publish Web Services, and is seen as an alternative to other distributed-computing specifications such as SOAP. A RESTful Web Service is identified by a URI, and a client interacts with the resource via HTTP requests and responses using a fixed set of HTTP methods.

A RESTful Web Service follows the following rules to provide access to the resources represented by the server:

1. The resource must be identified by a URI, which is a string of characters similar to a web address, that points to the resource.
2. A client interacts with the resource via HTTP requests and responses using a fixed set of HTTP methods.
3. One or more representations of a resource can be returned and are identified by media types.
4. The content of a resource can link to further resources.

There are two sides to consider for RESTful Web Services:

1. a **Client,** which would be an Omnis library containing methods to consume a RESTful Web Service,
2. and the **Server,** where you can implement a RESTful Web Service by exposing the business logic (remote task methods) in your Omnis library to be consumed by clients.

Example Library

There is an example library that demonstrates the capability of Omnis to consume a RESTful web service – the library is available with a Tech Note: TNWS0002 which is available on the Omnis website.

The example library presents basic weather forecast information by consuming a web service provided by openweathermap.org. The example is based on the free API service which you can use for the example.

API Key

To use the web service consumed in the example library, you must obtain a trial API key from openweathermap.org (which must not be shared with other people): note that the API key we used to create and run the online demo has been removed from the example library and *you will need to obtain your own API key*.

When you open the example Omnis library, you will be prompted to enter an API key. This value is stored in the remote task in the tAPIKey task variable, and you should not be asked to enter it again.

If you reuse any portion of the example app for your own development and deployment, or create your own application using the weather data from openweathermap.org, please remember to obtain a paid-for API key for your own or your clients use.

Testing the Example Library

To test the web service and display the weather forecast, open the example library, right click on the **jsWeather** remote form and select ‘Test Form’ from the context menu. The form should open in your desktop web browser and show the current weather for Saxmundham (the home town for the Omnis development team in the UK) – the same information can be accessed in a table format by selecting the Table hyperlink. In the main remote form there is a pictorial summary of today’s weather with the maximum & minimum expected temperatures, along with the forecast for the next four days. You can find the forecasts for other locations by entering either the city name or zip/post code.

The World tab gives a summary forecast for 20 selected cities in the world. Since the example library uses the free version of the API, all data is cached within the example library to prevent unnecessary calls back to the server. You can view the example app online on our hosted server.

If you publish the form to a webserver, when the form opens, it will try to identify your location using the IP address returned from the remote task. If this fails, it will revert to ‘Saxmundham’ as the default location.

Queueing RESTful requests & Licensing

RESTful requests to the Omnis Server consume a web user license for the duration of the request. In versions prior to Studio 8.1, if all licensed connections were in use when a new RESTful request came into the server, the client received an error. In Studio 8.1 onwards, RESTful requests are now queued internally until they succeed. Note that requests will never be re-queued in a single threaded server (a server where Start server has not been called) since everything executes sequentially.

In addition, there is a new sys function, sys(234), which returns a row of information containing statistics about RESTful requests to the Omnis server. The row has three columns: column 1 is the count of successful calls; column 2 is count of calls resulting in an error; and column 3 is the count of calls internally re-queued because there was not a free user.

Creating a Web Services Client

There are a few key requirements for creating a Web Services Client which are:

1. An HTTP client that allows resources to be submitted and received using various HTTP methods.
2. An HTTP client that allows HTTP headers to be specified for requests, and analysed for responses.
3. A means to manipulate the important media types for RESTful resources: XML or JSON.
4. Support for HTTPS, if required, which in a business environment is usually essential.
5. Support for HTTP basic and digest authentication.

The Web Services Client is implemented as an External Component Object. The External Objects group in Omnis Studio includes a group called **OW3 Worker Objects:** this contains a **HTTPClientWorker** object which is an HTTP Worker Object (along with FTP, SMTP and IMAP objects). *NOTE to existing users:* in versions prior to Studio 8.1 you needed to install and configure Java in order to use the HTTPClientWorker object in the **Web Worker Objects** group, but the new HTTP worker object in the OW3 Worker Objects group does not require Java.

The HTTP worker object functions in a similar manner to the DAM worker objects, although there is a simplification in the way they handle re-use of the object when a request is currently in progress: see the notes about $init.

To use the HTTP worker object, you need to create an **Object Class** which is a subclass of HTTPClientWorker. Having created a new Object class, set its $superclass property to the name of the HTTPClientWorker object by clicking on the dropdown list and selecting the HTTPClientWorker object in the OW3 Worker Objects group in the Select Object dialog.

In the object class, the methods **$completed** and **$cancelled** are inherited from the superclass (the HTTP worker object) which the client worker calls with either the results of a request, or to say the request was cancelled.

You then need to create an Object instance variable in your JavaScript remote form (or window class) based on the new Object class to instantiate the object and allow you to interact with the web service by running its methods.

Properties

The HTTPClientWorker object has the following properties:

$state

A kWorkerState... constant that indicates the current state of the worker object.

$errorcode

Error code associated with the last action (zero means no error).

$errortext

Error text associated with the last action (empty means no error).

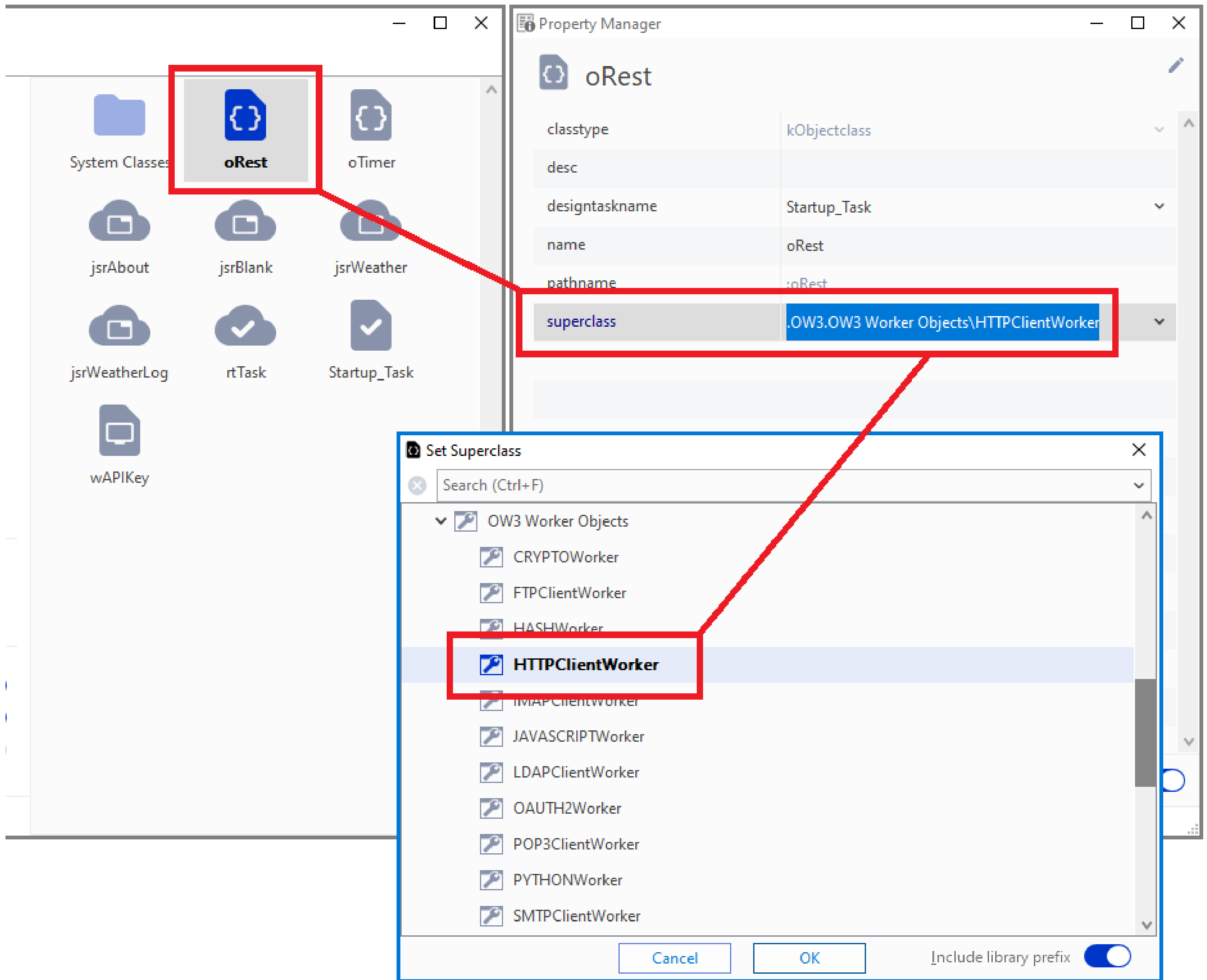


Figure 1:

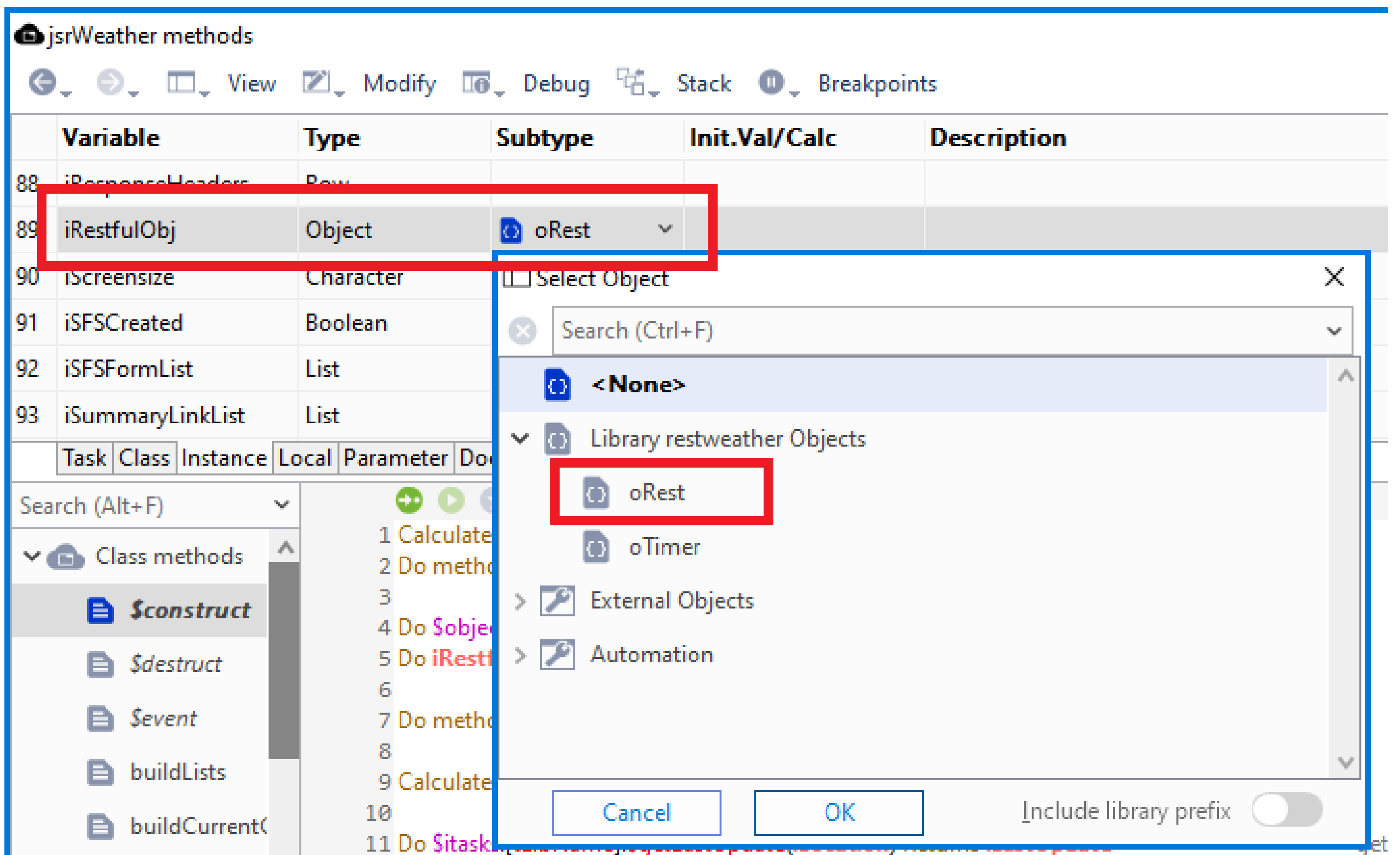


Figure 2:

$threadcount

The number of active background threads for all instances of this type of worker object.

$proxyserver

The URI of the proxy server to use for all requests from this object, e.g. http://www.myproxy.com:8080. Must be set before executing $init for this object.

$timeout

The timeout in seconds for requests from this object. Zero means default to the standard timeout for the HTTP client. Must be set before executing $init for this object.

$shareconnections

(Only applies to the old HTTPClientWorker object in the **Web Worker Objects** group) If true (default) the object shares connections to HTTP servers with other HTTPClientWorker objects rather than managing its own set of connections (consider authentication when setting this - different objects may need different authentication credentials, in which case you should not share connections). Must be set before executing $init for this object.

Methods

The HTTPClientWorker object has the following methods:

$init()

$init(cURI,iMethod,lHeaders,vContent[,iAuthType,cUserName,cPassword,cRealm]) Initializes the worker object so that it is ready to perform the specified HTTP request. Returns true if the worker was successfully initialized. The parameters are:

Parameter	Description
cURI	The URI of the resource, optionally including the URI scheme (http or https), e.g. http://www.myserver.com/myresource. If you omit the URI scheme, e.g. www.myserver.com/myresource, the URI scheme defaults to http
iMethod	A kOW3httpMethod... constant that identifies the HTTP method to perform: kOW3httpMethodDelete, kOW3httpMethodGet, kOW3httpMethodHead, kOW3httpMethodOptions, kOW3httpMethodPatch, kOW3httpMethodPost, kOW3httpMethodPut, kOW3httpMethodTrace
lHeaders	A two column list where each row is an HTTP header to add to the HTTP request. Column 1 is the HTTP header name, e.g. ‘content-type’ and column 2 is the HTTP header value, e.g. ‘application/json’
vContent	A binary or character variable containing the content to send with the HTTP request. If you supply a character variable, the worker converts it to UTF-8 to send in the request. Content can only be sent with Patch, Post and Put methods
iAuthType	A kOW3httpAuthType... constant that specifies the type of authentication required for this request. If you omit this and the remaining parameters, there is no authentication (and this parameter defaults to kOW3httpAuthTypeNone). Supported values are kOW3httpAuthTypeNone, kOW3httpAuthTypeBasic and kOW3httpAuthTypeDigest
cUserName	The user name to use with authentication types kOW3httpAuthTypeBasic and kOW3httpAuthTypeDigest
cPassword	The password to use with authentication types kOW3httpAuthTypeBasic and kOW3httpAuthTypeDigest
cRealm	The realm to use with authentication type kOW3httpAuthTypeDigest

NOTE: If you call $init when a request is already running on a background thread, the object will cancel the running request, and wait for the request to abort before continuing with $init.

$run()

Runs the worker on the main thread. Returns true if the worker executed successfully. The callback $completed will be called with the results of the request.

The following method is from the example library – the method initializes the Web Services object, having already setup the main parameters for the $init() method, and calls the web service.

```
# iURI (Char) initialized as "http://api.openweathermap.org/data/2.5/weather"
# The full URI sent has the API key, location appended as well as an optional parameter to return the data in
# iHTTPMethod (Int) set to kOW3httpMethodGet
# iHeadersList (List)
# iContentChar (Char)
Do iHeadersList.$define(iHeaderName, iHeaderValue)
Do iHeadersList.$add("content-type", "application/json")
# call the web service
Do iRestfulObj.$init(iURI, iHTTPMethod, iHeadersList, iContentChar)
Do iRestfulObj.$run() Returns lStatus
```

See $completed for handling the response from the web service.

Note: $run should not be used if you are hosting the RESTful service you are calling in the same instance of Omnis as your client. This would prevent the server-side execution from running and hang Omnis.

$start()

Runs the worker on the background thread. Returns true if the worker was successfully started. The callback $completed will be called with the results of the request, or alternatively $cancelled will be called if the request is cancelled.

$cancel()

If required, cancels execution of the worker on the background thread. Will not return until the request has been cancelled.

Method names

If you add further methods to your web services object you should avoid using names that may conflict with any possible reserved words on your system.

Callbacks

The HTTPClientWorker object calls the following callbacks, which must be defined in your object class:

$cancelled

Called to report that the request has been cancelled.

$completed

Called to report completion of the request. It has a single row variable parameter with columns as follows, including the content returned in the final column:

Column	Description
errorCode	An integer error code indicating if the request was successful. Zero means success, i.e. the HTTP request was issued and response received - you also need to check the httpStatusCode to know if the HTTP request itself worked
errorInfo	A text string providing information about the error if any
httpStatusCode	A standard HTTP status code that indicates the result received from the HTTP server

Column	Description
httpStatusText	The HTTP status text received from the HTTP server
responseHeaders	A row containing the headers received in the response from the HTTP server. The header values are stored in columns of the row. The column name is the header name converted to lower case with any - (hyphen) characters removed, so for example the Content-Length header would have the column name contentlength
responseContent	A binary column containing the content received from the server

The following is the $completed method in the oRest object in the example library:

```
# called by the client worker with the results
Calculate iResponse as pRow
Calculate iResponseHeaders as pRow.responseHeaders
Do OJSON.$formatjson(pRow.responseContent) Returns iReturnStr
Do OJSON.$jsontolistorrow(pRow.responseContent) Returns iJSONRow
```

The JSON external component can be used to process the JSON output.

CA Certificates

To make your RESTful based connection secure, you need to add a certificate file (.crt) to the appropriate place on your operating system. On Linux, the default installation for handling certificates is Open SSL; the CA certs are typically installed in /etc/ssl/certs/ca-certificates.crt.

The client certificates are specified by the last three arguments of the $setsecureoptions() method (which must be called before calling $run or $start).

- **$setsecureoptions()**
  Sets the options that affect how secure connections are established

  $setsecureoptions([bVerifyPeer=kTrue,bVerifyHost=kTrue,cCertFile='',cPrivKeyFile='',cPrivKeyPassword=''])

The last three arguments are:

- **cCertFile**
  Empty if client cert not required
  For macOS, pathname of .p12 file containing client cert and private key, or its keychain name.
  For Windows, certificate store path expression (CurrentUser\\MY\\<thumbprint>).
  For Linux, pathname of client cert .pem file
- **cPrivKeyFile** (Only required for Linux)
  The pathname of the private key .pem file. Empty if client certificate not required
- **cPrivKeyPassword** (Only required for Linux)
  The private key file password. Empty if client certificate not required

You can find more information about valid certificate store path expressions on the CURL website (https://curl.haxx.se).

Creating your own Web Services

There is a Web Server plug-in which allows you to expose the code in your Omnis applications, in an Omnis RESTful remote task, and provide them as Web Services for any clients to consume. The interface for the web services you can create and provide to clients is exposed as an API or set of APIs. The key requirements for Omnis to act as a server or provider of RESTful based Web Services are:

- Allow an HTTP client to submit and retrieve resources using various HTTPmethods.

- Expose the HTTP headers that arrive with a request and allow headers to be specified for the response.
- A means to manipulate the important media types for RESTful resources: XML and JSON.
- Support for HTTPS, and for HTTP basic and digest authentication.

These requirements can be met with a combination of the Omnis App Server and a standard HTTP Web Server.

There is a tech note and example library on the Omnis website to show how you can implement a Web Service from your Omnis code: TNWS0003: "Getting Started with RESTful Web Services, Tomcat and Swagger UI".

To deploy your Omnis-based Web Service, you will need to setup the Omnis App Server, using the Web Services Server plug-in (mod_OmnisREST.so) rather than the standard plug-in, which is detailed in the tech note: TNJS0003: "Setting Up The Omnis App Server".

A RESTful remote task can have a superclass in another library, provided that the superclass in the other library does not contain URIs. This allows you to use framework libraries.

Omnis RESTful APIs

Omnis RESTful APIs (or ORAs) can be fully defined using a "Swagger" definition, which is the most widely used standard for defining RESTful APIs. This section assumes you are using Swagger, but from Studio 10.2 you can use OpenAPI to define an Omnis RESTful API; see OpenAPI definitions.

Omnis Studio supports Swagger 2.0 for RESTful web services. This only affects the Swagger files Omnis generates, and there is just one definition per service. The Web Service Server library also has a link to save the Swagger file for a service to disk. The reasons for choosing swagger include:

- It makes it easier to document and test them
- It has tools to generate clients for various languages using the swagger definition
- It simplifies the generation of RESTful APIs in Omnis, so you can concentrate on application logic rather than lower level protocol related issues

Note however that there is nothing in the current implementation that requires a developer to use the Swagger definition in REST based web services.

Omnis uses the first non-empty description it can find for a remote task in the service as the description of the service in the Swagger file.

Web Services in the Omnis IDE

Omnis RESTful APIs are visible in the Studio Browser as children of the library node beneath the Web Service Server node (including Swagger and OpenAPI definitions). Each ORA is shown a separate node icon in the tree, and various options or actions are shown as hyperlinks when an ORA is selected. Omnis RESTful APIs have a method list and an error log that you can use to manage the service.

Omnis RESTful APIs have Swagger definitions that can be viewed using the IDE browser hyperlinks for the ORA. There is a hyperlink for the top-level resource listing, and a separate hyperlink for each top-level URI component. Clicking on a link displays the relevant Swagger data for the link (building it if necessary first). In the top of the panels is a read-only URL; you can select the text for the URL, and paste it into a browser or into Swagger UI (in the latter case, the resource listing URL is the only URL you would use). Note that you need to be aware of potential CORS issues when using these links in Swagger UI (see the later section on CORS).

Creating an Omnis RESTful API

To create a Web Service or Omnis RESTful API you need to set some properties of a remote task and add some RESTful methods. The remote task class has two properties to allow you to setup the Web Service:

- **$restful**
  If true, the remote task is RESTful, it can have URI objects, and can be part of a RESTful API by setting $restfulapiname. This property can only be set to kFalse when the remote task and superclasses have no URI objects

- **$restfulapiname**
  If not empty, this is the name of the RESTful API in the library containing the remote task (cannot equal $webservice for remote tasks in lib). The RESTful API name in this property must start with an alphanumeric (a-z) and can only contain a-z, 0-9 and _

To create an Omnis RESTful API (ORA), set the $restful property of a remote task to kTrue, and provide a name in $restfulapiname (this name will appear in the Studio Browser when you have added some methods). Note: the $restful property is an inherited property, so if you create a subclass of a remote task with $restful set to kTrue, the subclass will also be $restful. Further note a remote task with $restful set to kTrue is not yet a member of an ORA. For each remote task that is to belong to an ORA (meaning that it provides URIs and methods for clients to call) set $restfulapiname. Note that all remote tasks in an ORA must be in the same library. The $restful and $restfulapiname properties are available at runtime in remote task instances.

After setting $restful and $restfulapiname for a remote task class, the new ORA will not appear in the browser, because it has not implemented any RESTful methods. Therefore, the next step is to open the method editor for the remote task, in order to add objects and methods.

When a remote task is RESTful, the remote task has a group of objects (named $objs). These objects are the URIs exposed by the remote task to clients. Inheritance works with these objects and their methods in the same way that it works with other Omnis classes that support inheritance. However, $cfield and $cobj are not resolved for URI objects.

URIs

A URI must have one or more components starting with /. Parameter place-holders can be included as component two or later as {paramName} where paramName is unique (case-insensitive) in the URI. The URI cannot have a trailing / and cannot be duplicated. For example:

- /users
- /user/{userId}

In the second case, userId is a parameter place-holder, meaning that the RESTful methods implemented for the URI must all have a parameter named userId which Omnis populates with the userId from the addressed URI.

A URI is considered to be a duplicate (and therefore not allowed in the remote task) if it has the same number of components of another URI in the remote task or one of its superclasses, and all components match; components match if neither is a parameter place-holder and they have the same case-insensitive value, or if either of the components is a parameter place-holder.

HTTP methods

URIs are like other class objects in classes with instances, in that they can have their own methods. There are some special methods supported for URIs, called HTTP methods. These correspond directly to the HTTP protocol methods used by a RESTful API, and they are:

- $delete, $get, $head, $options, $patch, $post, $put

The HTTP methods are named with a leading $ (unlike the HTTP protocol methods) so that they work with the usual Omnis inheritance mechanism. URIs can also have other methods, but these are not HTTP methods and are not part of the public ORA. The name is the only property that determines if a URI method is an HTTP method, so renaming a method can make it become HTTP or non-HTTP accordingly.

Query Parameters

HTTP methods of a URI have some special features and properties. The first parameter for all HTTP methods must be named **pHeaders,** and defined as a Field reference. This references a row which contains the HTTP headers received in the RESTful request from the server. The row has a column for each HTTP header. The column names are created by converting the HTTP header name to lower case and removing any - characters e.g. Content-type becomes contenttype as a column name. If more than one header exists with the same name, the headers are combined into a single comma-separated value.

For methods which accept content with the request ($patch, $post, $put) the second parameter must be named **pContent,** which is a Field reference to the content received in the request.

When you create a new HTTP method, Omnis creates the parameters pHeaders and pContent automatically, and it also adds a character parameter for each parameter place-holder in the URI. In addition, you can add further parameters to the method (which must be of type character, Boolean, integer or number). Each further parameter is then expected to be part of the query string in the full URL used to make the RESTful call to the method; if you provide an initial value for the parameter, the parameter is optional in the query string.

When the RESTful call reaches the remote task method, pHeaders, pContent, the place-holder parameters and the query string parameters are all automatically populated by Omnis.

Note that once you have created the method, you can delete parameters which are required at runtime, e.g. pHeaders. However, Omnis will detect this and generate an error, either at the ORA level (see the error log in the browser) or when the client attempts to call the method.

An HTTP method has some additional properties:

- **Nickname**
  A name which must be unique in the set of all HTTP methods for the ORA. The nickname is used to uniquely identify the method in the Swagger definition for the RESTful API. Clients generated from the Swagger definition typically use the nickname as the method name to call in the client interface. When you create a new HTTP method, Omnis automatically assigns a default nickname

- **Input type**
  Methods which accept content with the request ($patch, $post, $put) have a property called input type, where the value is one of:

- empty if no content is to be supplied with the request

- a MIME type e.g. application/xml

- the name of an Omnis schema class in either the same library as the remote task, or another library. A schema input type is identified by the absence of a / (forward slash), and the supplied content must be application/json.
  The JSON input is automatically converted to an Omnis Row or List (based on the provided JSON, not the Schema).
  The request is automatically denied if the JSON does not contain members named the same as columns in the Schema which have No Nulls set to kTrue. The No Nulls check only happens for Columns at the Schema's top level, Omnis doesn't currently do this validation on nested Schemas (e.g. if you have a List/Row column type).
  You can use an empty Schema and Omnis will automatically convert the JSON to a List or Row. This may be preferable in many cases over using 'application/json' as the Input Type.
  Note that the columns in the schema class must be character, Boolean, integer, number, list or row, and when using list or row, the list or row must have a schema class subtype which also conforms to these type rules
  "application/x-www-form-urlencoded" content type, such as that generated by a form on a web page. Therefore, in addition to URL place-holder parameters, you can populate parameters using either the query string or application/x-www-form-urlencoded content. You cannot use both the query string and application/x-www-form-urlencoded content. To use application/x-www-form-urlencoded, set the RESTful input type to application/x-www-form-urlencoded. Omnis then expects application/x-www-form-urlencoded content containing each of the non-optional non-place-holder parameters. The raw application/x-www-form-urlencoded content is also supplied in the pContent parameter of the RESTful method: application/x-www-form-urlencoded content can only be used with HTTP methods that can send content to the server.

- **Output type**
  This specifies the type of content returned by the method when it returns the HTTP status of 200 (OK). One of:

- empty if no content is to be returned

- a MIME type e.g. application/xml

- the name of an Omnis *schema class* in either the same library as the remote task, or another library. The notes regarding schema classes and the input type also apply to the output type. To return an array of JSON objects you can use schema[] as the RESTful output type, or the return type for one of the HTTP status codes, e.g. mySchema[]. The RESTful method must then return a list defined from the schema rather than a row (see note below about $sendlistsasobjectarray)

- **HTTP response codes**
  A list of codes which can be returned by the method. These are the application codes that can be significant to clients; in addition, the Omnis server will return other codes such as internal server error, which should not be specified here. With each code you can specify optional status text and an optional schema class used to specify some JSON that you will return when the method returns this status code

The HTTP method properties affect how Omnis interacts with the HTTP method:

- When calling a method which accepts content with the request, then there are two possibilities:
  The input type is either empty or a MIME type. pContent is a field reference to a binary variable containing the content if any.
  The input type is a schema class. Omnis parses the JSON content and generates a row. In addition, Omnis checks that every column marked as "No nulls" in the input type schema is present in the row. If parsing fails, or the column check fails, Omnis returns an error to the client. Otherwise, pContent is a field reference to the row generated by parsing the JSON.

- When an HTTP method returns, the HTTP status code is set using $sethttpstatus; if it is not called, the status defaults to 200 (OK); the output type property determines the type of the output.
  If the HTTP status code set using $sethttpstatus is another value, then Omnis looks up the status code in the HTTP response codes, and uses the return type for the status code.

- Omnis uses the output type determined from the HTTP status code as follows:
  If the output type is not empty, then there must be some returned content. Omnis automatically sets the content-type header for the response to either the output type, or application/json if the output type is a schema. In addition, if the output type is a schema, then the return value from the method can either be:
  Binary (not recommended). Omnis looks at the first character of the content, and checks that it is {, as a sanity check to see if it is probably JSON (if the check fails, the client receives an error).
  A row (recommended). Omnis checks that the row is defined from a class with the same name (excluding the library) as the output type (if the check fails, the client receives an error). Omnis then automatically converts the row to JSON.
  If the output type is empty, then there can only be returned content if the method has already added a content-type header using $addhttpresponseheader; otherwise Omnis returns an error to the client.

Object array output type

When $sendlistsasobjectarray is set to true, the JSON generated by Omnis for a returned row or list that contains lists, contains arrays of objects rather than arrays of arrays (in this case the lists must only contain columns with simple types). There is one exception to this rule. If the list to be converted to JSON has a single column named "<array>", Omnis outputs the list as an array.

There is a checkbox on the RESTful panel for the HTTP method in the method editor, that allows you to select this option.

Note that this option applies to both rows returned by the method, and lists returned by the method when the return type is schema[]. In the latter case, the top-level array returned is always an array of objects, therefore you should note that the new option applies to lists contained in the returned list.

Unknown Query String Parameters

RESTful methods can allow unknown query string parameters. The RESTful panel for a RESTful method in the Method Editor has a checkbox option "Allow unknown query string parameters" (the default is unchecked). When checked, it means the RESTful server will accept requests that contain query string parameters that are not specified in the method parameters. The remote task instance can access these unknown parameters using the notation $cinst.$unknownquerystringparams. The properties are:

- **$allowunknownquerystringparams**
  If true, the RESTful method allows query string parameters that are not present in the method parameters. You access these unknown parameters using the property $unknownquerystringparams of the remote task instance.

- **$unknownquerystringparams**
  If unknown query string parameters are allowed, then this property is a row with a character column for each unknown parameter (the column name is the parameter name in lower case and the column value is the parameter value).

Escaping String Parameters

The "Escape query string parameters" option allows you to control whether or not string parameters are URI escaped; it defaults to true (replicating the behavior in versions prior to Studio 11), meaning that query string parameters are URI escaped. When turned off, the query parameters are not URI escaped, allowing you to perform any character encoding conversion yourself. For example, if you receive UTF-8 data instead of ASCII, you could turn this option off and escape the text using the ow3.escapeuritext() function.

Simple Types

In a schema class, a list column can have a so-called simple type as its sub-type. Valid values are <character>, <integer>, <boolean> and <number>. These allow ORAs to define JSON that contains arrays of simple types.

Method Editor

The method editor has additional features for a RESTful remote task. There are menu items that allow you to:

- Insert a new URI
- Delete a URI
- Insert a new HTTP method
- Rename a URI

These menu items are on the context menu for the method tree, and also in the modify menu in the toolbar, provided that the method tree has the focus.

In addition, when the currently selected method is an HTTP method, the variables panel has two additional tabs: RESTful and RESTful notes:

- RESTful allows you to set the input type, output type and HTTP response codes. The status code grid has a context menu you can use to manage its entries.
- RESTful notes allows you to add documentation notes about the method which Omnis writes to the Swagger definition.

Find and replace works with the RESTful properties; double clicking on a RESTful entry in the find and replace log will open the method editor with the property selected.

The Code Assistant lists column names after you enter the name of a list or row variable with a schema or table class as its subtype.

Server Properties and URLs

The Server Configuration dialog allows two properties to be configured:

- **RESTful URL**
  The base URL used to call Omnis RESTful Web Services, e.g. http://www.test.com/scripts/omnisrestisapi.dll Omnis uses this in the Swagger definitions it generates. If empty, Omnis uses http://127.0.0.1:$serverport
- **RESTful connection**
  [POOL,][IPADDR:][PORT]. Controls how the Omnis RESTful Web Server plugin connects to Omnis. POOL is a load sharing process pool name;IPADDR and PORT identify Omnis or load sharing process; if empty, defaults to $serverport

These properties are stored in the config.json file in the Studio folder of the Omnis tree. These properties affect the URLs stored in the Swagger definitions for ORAs implemented in the server.

If you do not set these properties, then the API will be defined to connect directly to the built-in HTTP server in Omnis.

In order to make a call to an ORA, you need a URI. If you look at an ORA in the IDE browser, you can see how the URIs are constructed by looking at the Swagger definition for a top-level URI path (using one of the hyperlinks immediately below the Resource listing hyperlink). The base path will be something like:

http://localhost:8080/omnisrestservlet/ws/5988/api/phase2/myapi

The initial part of the URL (http://localhost:8080/omnisrestservlet) gets the request as far as the Web Server plugin. The next two components of the URL (/ws/5988) tell the Web Server plugin how to connect to Omnis or the load sharing process. These two components are optional, and can be replaced with the Omnis-server header property described in the Phase 1 documentation; however, if you are likely to be doing cross-domain requests, then it is better to use the /ws/5988 form, since it is guaranteed to be sent with an OPTIONS method request. (Note that the “ws” is a fixed value). The second component (5988) has the general syntax definition:

- nnnn (a port number)
- or ipaddress:nnnn (IP address and port number)
- or serverpool,ipaddress:nnnn

The remaining components are forwarded to the Omnis server: /api/phase2/myapi. The first of these remaining components is a fixed value, which tells the Omnis server that this is a call to an ORA (this first component can also have the fixed value swagger as part of a URL to request a Swagger definition, or it can be of the form LIB.RT, as used in Phase 1 of the RESTful server implementation). The next two components are the library name and the ORA name.

When connecting directly to the Omnis server, the base URL is something like:

http://localhost:5988/api/phase2/myapi

Finally, when combined with the URI in a remote task in the server, the URL used to call an HTTP method for URI /users/{id} (with no query string parameters) is something like:

http://localhost:8080/omnisrestservlet/ws/5988/api/phase2/myapi/users/1234

ORA Properties and Methods

There are various properties and objects to support ORAs. As described earlier, the remote task has the properties $restful and $restfulapiname. RESTful remote tasks have a $objs group. Specific methods in this group are:

- **$add()**
  $add([cUri]) inserts a URI into a RESTful remote task and returns an item reference to it. cUri must be a valid remote task URI starting with a /
- **$remove()**
  $remove(rItem) delete the URI; rItem is an item reference to the URI to delete

HTTP methods in a RESTful remote task have the following properties:

- **$httpnickname** (note this was $httpoperationid in previous versions)
  A simple name for the RESTful remote task method exposed via a URI and HTTP method; it must be unique in the RESTful API; it cannot be empty, must start with an alpha character (a-z or A-Z) and can only contain a-z, A-Z, 0-9 and _
- **$httpinputtype**
  Only applies to RESTful remote task HTTP methods. Empty if no input content is required, or the name of a schema class describing the JSON input object if application/json input is required, or a MIME type if other input content is required
- **$httpoutputtype**
  Only applies to RESTful remote task HTTP methods when they return HTTP OK (200). Either empty if no content is output, or the name of a schema class that describes the output JSON object,or a MIME type for other output content
- **$httpnotes**
  Applies to RESTful remote task HTTP methods only. Notes about the method functionality

In addition, HTTP methods have a group:

- **$httpresponses**
  Applies to RESTful remote task HTTP methods only. The group of HTTP response objects that define the possible response codes (not including 200 OK) for the HTTP method

To add a new response code to the group, use:

- **$add(iCode[,cText,cType])**
  Adds a new HTTP response code definition for the method and returns an item reference to it

The members of the HTTP response codes group have properties as follows:

- **$httpresponsecode**
  An HTTP response code, in the range 201-599 (informational status codes 100-199, plus 200, are not reported)
- **$httpresponsetext**
  Text describing the HTTP response code
- **$httpresponsetype**
  This is the name of a schema class that describes the JSON object to be returned as the result of a RESTful call to the HTTP method which returns the associated response code. Empty means no content is returned

There are two properties in $root.$prefs (which are also in config.json):

- **$restfulurl**
  The base URL used to call Omnis RESTful Web Services, e.g. http://www.test.com/scripts/omnisrestisapi.dll. Omnis uses this in the Swagger definitions it generates. If empty, Omnis uses http://127.0.0.1:$serverport
- **$restfulconnection**
  [POOL,][IPADDR:][PORT]. Controls how the Omnis RESTful Web Server plugin connects to Omnis.POOL is a load sharing process pool name; IPADDR and PORT identify Omnis or load sharing process; if empty, defaults to $serverport

A RESTful remote task $construct method receives a row variable parameter with the following columns:

1. **url** or **fullurl**
   url is the partial url starting with the Omnis library component, or
   fullurl contains the full URL, starting with the path to the script, e.g. /omnisrest/ws/5988/api/...
2. **method**
   the name of the HTTP method.

The host name used can be obtained from the host header. There is no way to determine if the request was made using http or https.

Swagger Definitions

Omnis populates the Swagger definitions using the properties of the remote task. The Swagger method summary is the Omnis method description. A schema column with no nulls set to kTrue is marked as a required JSON member in the Swagger model object.

The Swagger resource listing contains various fields that need to be populated e.g. API version number, contact email etc. In order to do this, the Omnis tree contains a default template, and you can also create specific templates for specific ORAs. The default template is the file 'default.json' in the folder clientserver/server/restful/swaggertemplates in the Studio tree. You can edit this, or alternatively copy it and create an ORA specific template, which must have the name <restfulapiname>.json, and be stored in a sub-folder of swaggertemplates named with the library name e.g.

clientserver/server/restful/swaggertemplates/lib/myapi.json

Omnis reads the template each time it generates a new resource listing. Omnis keeps the Swagger definitions in step with changes in the environment e.g. when you save a remote task or relevant schema class, or change the RESTful URL or connection property.

You can use swagger-ui (https://github.com/wordnik/swagger-ui) with the built-in web server to test your ORAs. Take the dist folder for swagger-ui, drop it into the webapps folder of your web server tree, and rename it swagger-ui. Restart the web server. You can then use the URL http://localhost:8080/swagger-ui/index.html#!/path in a browser to open swagger-ui.

If you also place omnisrestservlet in web server 'webapps' folder (and restart the web server), and set Omnis server properties restful-connection to your server port, and restful URL to http://localhost:8080/omnisrestservlet, you can use swagger-ui without any cross-domain issues.

If you select your ORA in the Web Service Server node of the IDE browser, you can click on the Resource listing hyperlink, and copy the URL from the top of the panel showing the Swagger definition. Paste the URL into swagger-ui and press Explore - you should see your ORA.

OpenAPI Definitions

OpenAPI is a more up to date version of the RESTful API description format, and Studio 10.2 now generates OpenAPI 3.0.0 definitions, as well as Swagger 2.0 definitions.

When you select a RESTful service beneath the Web Service Server node in the browser, there are now two pairs of links:

- OpenAPI Definition, Save OpenAPI to File
- Swagger Definition, Save Swagger To File

The OpenAPI definition can be retrieved using a similar URL to that used to retrieve a Swagger definition by replacing 'swagger' in the URL with 'openapi'.

There is a new folder in clientserver/server/restful, named openapitemplates. The files in here have the same use as those in the swaggertemplates folder, except that they apply to OpenAPI definitions.

In addition, cors.json has new OpenAPI members that have a similar purpose to the Swagger members.

In versions prior to Studio 10.2, you could provide a format by prefixing a description of a schema field or HTTP method parameter with "<swagger-...>". In Studio 10.2, you can now provide a format using the prefix of either "<format-...>" or"<swagger-...>".

Media types

HTTP responses for a RESTful method can now be defined to return media types other than application/json via a schema.

Note that if you do this, the Swagger 2 definition is incomplete, since Swagger 2 does not allow mixed response content types. However, the new OpenAPI 3 definition for the RESTful service does handle this correctly.

Managing Return Values

There may be occasions where RESTful API remote tasks are not able to generate their content as the return value of the HTTP method. For these cases, content generation can be deferred until later, for example, until a threaded worker object completes, or to allow push support, possibly using server sent events and text/event-stream content. In order to do this, there are additional steps. Before returning from the HTTP method (where you would usually return content):

```
Calculate $cinst.$restfulapiwillclose as kFalse
```

This prevents the remote task from closing when you return, and it means that you are responsible for closing the remote task by calling $close() at a later point, or by using the remote task timeout mechanism. Note that it is essential to close the remote task, so that the data connection to the client is closed.

Note that setting $restfulapiwillclose to kFalse will be ignored if an error is detected by the Omnis server as part of request processing.

$restfulapiwillclose has the following definition: If true,the RESTful API remote task will close when the Omnis RESTful HTTP method returns. Defaults to kTrue in a new RESTful API remote task. kFalse only applies when the method executes successfully; you must eventually call $close().

After setting $cinst.$restfulapiwillclose to kFalse, you do not need to return any content, headers or status from the method. If you do return content though, then you also need to set the HTTP status and add any response headers before returning the content. Note that the Omnis server no longer automatically adds the content-length header - this becomes your responsibility if this header is required (in many cases like this it is not).

Sending HTTP Content

The $sendhttpcontent() remote task method lets you send HTTP content back to the client:

- **$sendhttpcontent()**
  $sendhttpcontent(xData) sends the next block of HTTP content (xData) to the client for a RESTful API remote task that did not close

When you are ready to generate the response e.g. in a worker callback, call $sendhttpcontent. The xData parameter differs from content returned from a RESTful HTTP method, in that it is always binary (meaning that you are responsible for generating JSON or encoding characters for example).

You can call $sendhttpcontent more than once, to incrementally send content. However, before the first call, you must set the HTTP status and supply the HTTP response headers (including content-length or transfer-encoding chunked if required).

When using $cinst.$restfulapiwillclose set to kFalse, the Omnis server does not attempt to validate the content returned as it does for JSON content when using $cinst.$restfulapiwillclose set to kTrue.

$sendhttpcontent cannot be used in the initial RESTful API HTTP method call.

The $sendhttpcontent() method can be used to send character data (converted to UTF-8 before sending) and list or row data (converted to JSON before sending). In addition, the method has an optional second argument, bChunk, which defaults to kFalse (the current behaviour). When true, bChunk formats the data as a chunk (removing the need to call the formatchunk() function). This improves performance a little, and also allows you to handle web servers which automatically chunk the response. A call to $sendhttpcontent with empty data and bChunk passed as kTrue must be used to terminate the content.

Transfer-encoding chunked

You can return content in multiple blocks using transfer-encoding chunked by using $sendhttpcontent. To facilitate this, there is a built-in function:

- **formatchunk()**
  formatchunk(data) formats the data as a chunk suitable for sending to the client using chunked transfer encoding.Data can be character (which Omnis converts to UTF-8) or binary

Each data block to be sent can be sent with code such as:

```
$ctask.$sendhttpcontent(formatchunk(data))
```

These calls need to be followed by a call to send a zero-length chunk (which terminates) the content:

```
$ctask.$sendhttpcontent(formatchunk())
```

Server Sent Events

You can use $sendhttpcontent to handle a push connection from a client using Server Sent Events. To do this, set the output type for a get method to text/event-stream. Note: you do not need a content-length header for this. You can then send events to the client using $sendhttpcontent. To facilitate this, there is a built-in function:

- **formatserversentevent()**

  formatserversentevent(fieldname,fielddata[,fieldname,fielddata]...): formats data suitable for sending as an event when generating text/event-stream content.Parameters can be character (which Omnis converts to UTF-8) or binary (UTF-8)

For example:

```
Do $ctask.$sendhttpcontent(formatserversentevent("id",1,"data","my event data"))
```

The protocol field names in the example are data and id, with values 1 and “my event data” respectively.

Date and Date-time values

Support for date and date-time values has been added to REST-based Web Services support. RESTful services typically use a subset of ISO8601 to exchange date and date-time values, which are supported in Swagger which is used to define web services in Omnis. ISO8601 represents the date or date-time as a character string. See http://swagger.io/specification/ and search for RFC3339 in the page - a link from there takes you to http://xml2rfc.ietf.org/public/rfc/html/rfc3339.html#anchor14.

Swagger has two format specifiers for character string values: date and date-time. The Swagger generator has been enhanced so that for fields (either in a schema or in the HTTP method parameters) of type character, the description can contain a swagger tag that specifies the format, e.g. the description for query string parameter startDate could be "<swagger-date> This parameter is the start date for the requested work". When Omnis generates the swagger definition for the web service, it looks for these swagger tags, and uses them to set the format for Swagger string types. This has the additional benefit that you can use other supported Swagger string formats, e.g. password and byte.

Dates and date-time values are still exchanged as character values. The application code therefore needs to parse and generate the ISO8601 date and date-time values. To support this, there are two new functions to manipulate ISO8601 dates, or at least the subset of ISO8601 needed to work with Swagger and the Omnis RESTful server:

- **iso8601toomnis()**
  iso8601toomnis(cISO8601,bNeedTime,bNeedTimeZone,cErrorText]) converts ISO8601 date/date-time string to Omnis date-time and returns result (in UTC time if cISO8601 contains time and time zone).
  Returns #NULL and sets cErrorText if an error occurs

- **omnistoiso8601()**
  omnistoiso8601(dOmnisDateTime,bNeedTime[,cErrorText]) converts dOmnisDateTime (assumed to be in UTC) to an ISO8601 date or date-time string (depending on bNeedTime) and returns the result.Returns #NULL and sets cErrorText if an error occurs

Note that for a RESTful service, you should always use time zones for input date time values, so you would always pass bNeedTimeZone as kTrue to iso8601toomnis if you are passing bNeedTime as kTrue.

omnistoiso8601() always outputs the timezone using the "Z" UTC time indicator.

Web Services Functions

HTTP Headers

The following functions facilitate using date HTTP header values.

- **parsehttpdate()**
  parsehttpdate(httpDate) parses a date value in HTTP header format (e.g. Sun, 06 Nov 1994 08:49:37 GMT) and returns an Omnis date-time value (in UTC) or NULL if the value cannot be parsed successfully.

- **formathttpdate()**
  formathttpdate(omnisDate) formats the Omnis date-time value (assumed to be in UTC) as an HTTP date header value and returns the resulting string.

- **parsehttpauth()**
  *parsehttpauth(*auth*)* parses the HTTP Authorization header value *auth* and returns a row variable containing the extracted information. See *Authorization* section for more details.

BASE64 encoding

The following functions are for handling BASE64 encoded data. You are recommended to use these with RESTful requests that require them, rather than the functions in OXML.

- **bintobase64()**
  bintobase64(vData) encodes *vData* as BASE64 and returns the result. *vData* can be either binary or character. If *vData* is character, Omnis converts it to UTF-8 before encoding it as BASE64.

- **binfrombase64()**
  binfrombase64(vData) decodes the binary or character *vData* from BASE64 and returns the resulting binary data. Returns NULL if *vData* is not valid BASE64.

Cross Origin Resource Sharing

Cross Origin Resource Sharing (CORS) “is a mechanism that allows many resources (e.g., fonts, JavaScript, etc.) on a web page to be requested from another domain outside the domain the resource originated from” (wikipedia). An Omnis RESTful API can handle CORS by implementing the $options HTTP method, and by handling the Origin and other headers when processing other HTTP methods (see above for details). In addition, you can configure the Omnis Server (in both the development and server runtime versions) to automatically handle CORS. This means that the Omnis Server can be configured to automatically send the response to OPTIONS, and to add the correct CORS headers to the response buffer before passing a simple or actual request to the application.

The configuration of CORS for RESTful-based web services is stored in a separate configuration file, ‘cors.json’ which should be added to the ‘Studio’ folder: there is no CORS configuration if the cors.json file is not present in the Studio folder. (In previous versions the CORS configuration was stored in a “CORS” section in config.json which is now redundant: since it is a separate file you can edit it while the Omnis Server is running.)

There is a template cors.json file containing the required settings located in the ‘templates’ folder in the Studio folder: you can make a copy of this file and place the copy in the Studio folder, making any necessary changes.

The cors.json file can be changed while Omnis is open, but you need to inform Omnis of the change. This can be done using a button on the server configuration dialog, “Reload CORS Config”.

The CORS object can have the following members (note that everything here is optional, and the most likely result of omitting data is that a request will be passed to the application to handle, or a method not supported error will be returned to the client if the application does not implement the method):

- originLists: Each member of originLists is a named list of origins, i.e. possible values for the HTTP Origin header. (Each list is an array)
- headerLists: Each member of headerLists is a named list of HTTP headers (Each list is an array)
- exposedHeaderLists: Each member of exposedHeaderLists is a named list of HTTP headers (Each list is an array)
- APIS: This object has members as follows:
  *: Server wildcard **CORS entry**. See below for the definition of CORS entry
  Swagger: Server Swagger CORS entry
  libraryname.apiname. Each library.api object has members as follows:
  *: API wildcard CORS entry
  Swagger: API Swagger CORS entry
  CORS entries named using a URI string. These URI strings need to
  match URI object names in the API

A CORS entry has members as follows:

- origins: This has either the value * (meaning that when this CORS entry is used, all origins are allowed), or the name of a member of originLists (meaning that when this CORS entry is used, only the origins in the list are allowed).
- headers: This has either the value * (meaning that any header requested by the client using the Access-Control-Request-Headers header is acceptable), or the name of a member of headerLists (meaning that only headers in this list can be requested by the client using the Access-Control-Request-Headers header).
- exposedHeaders: The name of a member of exposedHeaderLists. Headers in this list will be returned using Access-Control-Expose-Headers when handling a simple or actual request.
- supportsCredentials: If true, and the origin is allowed, the server adds Access-Control-Allow-Credentials with value true.
- maxAge: The number of seconds that a client is allowed to cache the result of an OPTIONS method.

CORS processing in the Omnis server occurs when a request with an Origin header arrives. The server tries to locate a CORS entry for the request. There are two cases:

- When the client is requesting Swagger data, the server looks for the API Swagger CORS entry. If the API has no configuration, or no API Swagger CORS entry, the server looks for the Server Swagger CORS entry.

- When the client is executing an API method (resulting in either the method call or an OPTIONS method call), the server looks for the CORS entry exactly matching the URI that will be used to make the request; if that is missing, the server looks for the API wildcard CORS entry; and if the latter is missing, the server looks for the Server wildcard CORS entry.

If the above processing does not locate a CORS entry, then the server does not carry out any CORS processing, and the request continues as it would without CORS. If however the above processing locates a CORS entry:

- The server will attempt to generate the response to OPTIONS, provided that the logic in section 6.2 of the W3C Recommendation referenced earlier applies.
- The server will add CORS headers to the response buffer for other requests, provided that the logic in section 6.1 of the W3C Recommendation referenced earlier applies.

In order to understand what is going on, there is a new log type that you can specify in the datatolog member of the main Omnis log configuration: "cors". Using this will cause the server to log CORS issues that mean the CORS processing in the server has not handled the request, and is passing it on to the application if possible.

Logging

You can monitor or debug REST requests and responses by adding the relevant items to the "log" member of the Omnis configuration file (config.json) which is located in the 'studio' folder in the development and server versions of the Omnis tree. The REST request and response items can be added to the "datatolog" array in the "log" member using the following format:

```
"log": {
       "logcomp": "logToFile",
       "datatolog": [
           "restrequestheaders",
           "restrequestcontent",
           "restresponseheaders",
           "restresponsecontent",
           "tracelog",
           "seqnlog",
           "cors",
           "headlessmessage",
           "headlesserror",
           "systemevent"
       ],
       "overrideWebServicesLog": true,
       "logToFile": {
           "folder": "logs",
           "rollingcount": 10
       },
       "windowssystemdragdrop": true
   },
```

Note: when copying or editing sections in the config.json, you must be careful to include a single trailing comma when required to separate items, i.e. include a comma if another item follows the item you are editing.

Including the "cors" item will cause the server to log CORS issues that mean the CORS processing in the server has not handled the request, and is passing it on to the application if possible.

Further information about Logging in the Omnis Server is available in the Deployment chapter in the 'Creating Web & Mobile Apps' manual.

Authentication

You must be responsible for setting up authentication in your Omnis library. When using a real Web Server, rather than the built-in web server, you can configure the URL for the web service to support basic or digest authentication. There is also the option of using https, and also client certificates to further secure connections.

The *parsehttpauth(*auth*)* function parses the HTTP Authorization header value *auth* and returns a row variable containing the extracted information. Column 1 of the returned row (named scheme) is the scheme (e.g. basic). Other columns are scheme dependent. Examples for various auth header values:

- **Basic**
  Returned row has three columns:
  scheme: basic
  username: Aladdin
  password: open sesame

```
Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

- **Digest**
  Returned row has 10 columns:
  scheme: digest
  username: Mufasa
  realm: testrealm@host.com
  nonce: dcd98b7102dd2f0e8b11d0f600bfb0c093
  uri: /dir/index.html
  qop: auth
  nc: 00000001
  cnonce: 0a4f113b
  response: 6629fae49393a05397450978507c4ef1
  opaque: 5ccc069c403ebaf9f0171e9517f40e41

```
Digest username="Mufasa",realm="testrealm@host.com",nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",uri="/dir/index
```

- **OAuth**
  Returned row has 9 columns:
  scheme: oauth
  realm: Example
  oauth_consumer_key: 0685bd9184jfhq22
  oauth_token: ad180jjd733klru7
  oauth_signature_method: HMAC-SHA1
  oauth_signature: wOJIO9A2W5mFwDgiDvZbTSMK%2FPY%3D
  oauth_timestamp: 137131200
  oauth_nonce: 4572616e48616d6d65724c61686176
  oauth_version: 1.0

```
OAuth realm="Example",oauth\_consumer\_key="0685bd9184jfhq22",oauth\_token="ad180jjd733klru7",oauth\_signature
```

- **Bearer**
  Returned row has 2 columns:
  scheme: bearer
  token: 0b79bab50daca910b000d4f1a2b675d604257e42

```
Bearer 0b79bab50daca910b000d4f1a2b675d604257e42
```

- **Any other scheme**
  Returned row has 2 columns:
  scheme: scheme name in lower case
  data: the rest of the header data

Web Server Configuration for Authentication

If you want to setup Basic and/or Digest authentication for your web services running on Apache Web Server or IIS, please refer to the tech notes section on the Omnis website at: www.omnis.net/technotes

Manipulating Resources

The server can use the same mechanism for manipulating resources as the client, so refer to the sections above. Configuring and logging the Omnis RESTful API server is achieved in the Omnis configuration file (config.json).

Chapter 2—OJSON

The OJSON external component allows you to manipulate JSON based content in your Omnis applications – specifically it can be used to handle content returned by REST based web services. JSON is a text form that allows you to transmit data objects consisting of attribute–value pairs, and is an alternative to XML.

It is used in the Export Library to JSON option in Omnis, which provides a convenient way to store libraries and classes in a version control environment such as GIT. In addition, JSON is used to store various Omnis configuration files, including the main config.json, as well as the theme files.

The JSON component provides two ways to generate and parse JSON objects and documents:

1. A basic mechanism that simply maps directly between an Omnis list or row and JSON. This uses static methods in the OJSON component.

2. A more structured mechanism that uses an external component object called JSON in the OJSON external component.

Data Structure and Addressing

JSON maps on to a hierarchical Omnis list or row, with one exception, namely that it allows for mixed types in arrays – Omnis can cater for mixed types in list columns internally, but there is no support in the Omnis language for such a list column. Therefore:

1. you can map a JSON object to an Omnis row variable, where the column names are the JSON object member names.

2. you can map a JSON array where all of the members have the same primitive type (or null) to a single column Omnis list.

3. you can map a JSON array where the members are arrays or objects, or where there is more than one primitive type, to an Omnis row variable with columns named___1, ___2, etc. Note that this restricts such arrays to a dimension of 32000 or less.

Note: Primitive types are string, integer, float and Boolean.

The external component JSON object implemented in OJSON uses a character string called the member id to address an entity in a JSON document (an entity is essentially a node in the JSON document tree, so it can represent a primitive type, null, an array or an object):

1. The member id of the root of the document is the empty character string.

2. The member id for other objects is the dot-separated path through the document to the entity e.g.

a. If the root is an object with members a, b and c, the member ids for the object members are: a b and c

b. If b is an array with two elements, the member ids of the array elements are b.[0] and b.[1]

c. If b.[1] is an object with string member c, then c has the member id b.[1].c

Using member ids you can directly address any entity in the JSON document. Note that the empty member id can only be used to address the array or object which the root of the JSON document tree.

JSON Arrays

Omnis allows you to manipulate JSON arrays of objects, mapping them to and from Omnis list variables. There are two static functions in the OJSON external, $objectarraytolist() and $listtoobjectarray(), that work with a single level JSON array, where each array element is an object, and each object has members which are only simple types (integer, number, boolean, string).

When writing a list to an object array, OJSON validates the list, and returns an error if the Omnis data type of a column value is not suitable.

When parsing an array of objects, OJSON validates the data as it parses it, to make sure it is a single array of objects containing only simple types. In addition, OJSON adds columns on the fly, and if a column already exists makes sure the data in the JSON is of the same type as the already added column. This works best when all entries in the array are objects with identical members.

Static Methods

The OJSON object provides the following static methods:

OJSON.$jsontolistorrow()

OJSON.**$jsontolistorrow**(vData,[&cErrorText])
Parses the JSON array or object in vData (either binary (UTF8/16/32) or character) and returns a row or a list representing the JSON. Returns NULL and cErrorText if an error occurs.

OJSON.$listorrowtojson()

OJSON.**$listorrowtojson**(vListOrRow [,iEncoding=kUniTypeUTF8, &cErrorText, iOptions=kOJSONoptionNone]) converts the list or row to JSON. Returns JSON with specified encoding (UTF8, UTF16BE/LE, UTF32BE/LE or Char). Returns NULL and cErrorText for an error.

The iOptions parameter can be used to make $listorrowtojson process all members of a top-level row, and discard empty or null values appropriately, recursively descending into child lists and rows. The iOptions parameter can be one of the following constants, which can be summed together to get the desired result:

1. **kOJSONoptionNone** (the default)
   0 - No option specified - results in the old behavior
2. **kOJSONoptionOmitEmpty**
   1 - Omit empty values, objects and arrays from the output JSON
3. **kOJSONoptionOmitNull**
   2 - Omit NULL values from the output JSON

OJSON.$couldbearray()

OJSON.**$couldbearray**(vData)
Returns true if vData (either binary (UTF8/16/32) or character) could possibly be a JSON array because its first character is [.

OJSON.$couldbeobject()

OJSON.**$couldbeobject**(vData)
Returns true if vData (either binary (UTF8/16/32) or character) could possibly be a JSON object because its first character is {.

OJSON.$formatjson()

OJSON.**$formatjson**(vData)
Parses the JSON in vData (either binary (UTF8/16/32) or character) and returns a formatted representation (or error message if parsing fails) suitable for use in a multi-line entry control.

The following static methods allow you to manipulate object arrays:

OJSON.$listtoobjectarray()

OJSON.**$listtoobjectarray**(*lList*[,*iEncoding*=kUniTypeUTF8,*&cErrorText*])
Writes a list with simple columns to an array of objects; returns JSON with specified encoding (UTF8,UTF16BE/LE,UTF32BE/LE or Character). Returns NULL and cErrorText for an error.

OJSON.$objectarraytolist()

OJSON.**$objectarraytolist**(*vData*[,*&cErrorText*])
Parses vData (binary (UTF8/16/32) or character). vData must be a JSON array of objects containing members with simple types. Returns a list representing JSON. Returns NULL and cErrorText for an error.

The following static methods allow you to manipulate an array of arrays:

OJSON.$listtoarrayarray()

OJSON.**$listtoarrayarray**(*lList*[,*iEncoding*=kUniTypeUTF8,*&cErrorText*]) writes a list with simple columns to an array of arrays; returns the JSON with specified encoding (UTF8, UTF16BE/LE, UTF32BE/LE or Character). Returns NULL and cErrorText for an error.

OJSON.$arrayarraytolist()

OJSON.**$objectarraytolist**(*vData*[,*&cErrorText*]) parses vData (binary (UTF8/16/32) or character). vData must be a JSON array of arrays containing members with simple types. Returns a list representing the JSON. Returns NULL and cErrorText for an error.

These methods only work with simple types as array members. $arrayarraytolist requires that each JSON array must have the same number of elements, and each JSON array member at a particular index must be of the same data type.

JSON External Component Object

After constructing the OJSON object, it represents an empty JSON object. The methods supported by the external component object (with the exception of $getlasterror()) all set an error code and error text if an error occurs during their execution. In addition, you can use the method $runtimerrors to set a flag that causes the component to generate a runtime error (entering the debugger if applicable) when an error occurs - this can be useful when developing code that uses OJSON. The object provides the following methods.

Note: in all of these descriptions, cMember is the member id of an entity in the JSON document tree:

$getjson([cMember,iEncoding=kUniTypeUTF8])

Returns the JSON for the OJSON object (when cMember is an empty string or omitted) or the specified array or object member (cMember) using the specified encoding.

$setjson(cMember,vData)

Sets the OJSON object (when cMember is an empty string) or the specified array or object member (cMember) to the JSON supplied in vData (either binary (UTF8/16/32) or character). Returns a Boolean, true for success.

$getlasterror([&cErrorText])

Returns the error code from the last OJSON object method executed; also optionally populates cErrorText with a description of the error.If no error occurred, returns zero and the error text is empty.

$runtimeerrors(bGenerate)

Call with true to cause a runtime error when a method returns an error (so the debugger is entered if applicable), or false to stop runtime errors. Returns previous value of bGenerate. Default is no runtime errors.

$listmemberids()

Returns a single column list of the member ids for all of the members.

$isobject(cMember)

Tests specified member. Returns true if the member is a JSON object.

$getobject(cMember)

Gets the specified object. Returns a row containing the object members or NULL if the member is not an object.

$setobject(cMember,wRow)

Sets the specified member to the object specified in wRow. Returns a Boolean, true for success.

$addmember(cMember,cNewMemberName,vValue)

Adds member cNewMemberName with value vValue to object cMember. Returns a Boolean, true for success.

$removemember(cMember,cMemberName)

Removes member cMemberName from object cMember. Returns a Boolean, true for success.

$hasmember(cMember,cMemberName)

Returns true if cMemberName is a member of object cMember.

$listmembers(cMember)

Returns a single column list which contains the member names of the object cMember. Returns NULL if cMember is not an object.

$isarray(cMember)

Tests specified member. Returns true if the member is a JSON array.

$getarray(cMember)

Gets the specified array. Returns NULL if the member is not an array, a single column list if the array elements all have the same primitive type or are NULL, or for mixed arrays a row with one column per array element.

$getarraylength(cMember)

Returns the number of elements in the array cMember. Returns zero if cMember is not an array.

$setarray(cMember,vListOrRow)

Sets specified member to an array.Accepts either a single column list or a row where the columns are the array elements (the latter allows for arrays of mixed types). Returns a Boolean, true for success.

$push(cMember,vValue)

Adds an element with value vValue to the end of the array cMember. Returns a Boolean, true for success.

$pop(cMember)

Removes the last element from the end of the array cMember and returns its value as a row variable. Returns NULL if cMember is not an array or if cMember is empty.

$isstring(cMember)

Tests specified member. Returns true if the member is a JSON string.

$getstring(cMember)

Gets specified string member. Returns JSON string value (empty if member is not a string). Unescapes JSON syntax.

$setstring(cMember,cString)

Sets specified member to JSON string with value cString. Returns a Boolean, true for success.

$isbool(cMember)

Tests specified member. Returns true if the member is a JSON Boolean.

$getbool(cMember)

Gets specified Boolean member. Returns Boolean corresponding to JSON Boolean (false if member is not a Boolean).

$setbool(cMember,bBool)

Sets specified member to JSON Boolean with value bBool. Returns a Boolean, true for success.

$isinteger(cMember)

Tests specified member. Returns true if the member is a JSON integer.

$getinteger(cMember)

Gets specified integer member. Returns integer 64 bit (zero if member is not integer).

$setinteger(cMember,iInt)

Sets specified member to JSON integer with value iInt. Returns a Boolean, true for success.

$isfloat(cMember)

Tests specified member. Returns true if the member is a JSON floating point value.

$getfloat(cMember)

Gets specified floating point member. Returns number floating dp (zero if member is not floating point).

$setfloat(cMember,nNum)

Sets specified member to JSON floating point with value nNum. Returns a Boolean, true for success.

$isnull(cMember)

Tests specified member. Returns true if it is null.

$setnull(cMember)

Sets the specified member to null. Returns a Boolean, true for success.

Chapter 3—Java Objects

The *Java Objects* library allows you to use Java Objects from Omnis code. Once a Java Object has been created in Omnis, its methods can be called in the same manner as any Omnis object.

This chapter assumes that you have both a basic knowledge of Omnis Studio notation and a good knowledge of the Java Language.

IMPORTANT NOTE: Oracle has changed the way it licenses Java. So in order for you to avoid the ongoing use of Java in connection with Omnis Studio, we no longer provide various Java files in Omnis Studio 10 or above and consequently we have removed various Omnis libraries or features that rely on Java: these files are available from support if you need them. Specifically, the javaobjs and javacore libraries have been removed; the Reset Java Class Cache hyperlink in the Studio Browser is not shown, and will only appear if the JavaObjs Library is put back in the Omnis tree and loaded.

Setting Up

To use Java Objects in Omnis, you must install the latest version of **Java 8** which is now available from Oracle who acquired Sun Microsystems and the Java technology.

For Studio 10 or above you need to install the javaobjs and javacore libraries, available from support.

Software Requirements

To use Java in Omnis Studio for development and deployment (such as Java Objects or the Web Services component, which uses Java) you need to install and reference **Java Version 8,** which is available from Oracle: you can download the Java Developer Kit (JDK), for Windows or macOS, or Java Runtime Environment (JRE), for Windows only, from the following location:

- http://www.oracle.com/technetwork/java/javase/downloads

Java Configuration

Having installed the latest JDK or JRE you need to configure the JVM, either using a new entry in the Omnis configuration file (config.json), or by setting an environment variable: OMNISJVM64 or OMNISJVM32 depending on whether you are running the 64-bit or 32-bit version of Omnis Studio. If you specify a value in config.json, it overrides the value in the environment variable.

To setup the JVM in the config.json file, add an entry named “jvm” in the “java” object in the configuration file, for example, on Windows:

```
"java": {
  "jvmPath":"c:\\Program Files\\Java\\jre8\\bin\\server\\jvm.dll",
  "resetClassCacheOnStartup": false
}
```

Or on macOS:

```
"java": {
  "jvmPath":"/Library/Java/JavaVirtualMachines/jdk1.8.XXX.jdk/Contents/Home/jre/lib/server/libjvm.dylib",
  "resetClassCacheOnStartup": false
}
```

You can set the JVM in the config.json file on a Linux server in a similar manner.

Current Restrictions

There are currently a number of restrictions imposed on *Java Objects* as follows:

- Java Events are not supported.
- The Java Objects component is non-visual and the use of awt/swing is not supported.
- Java Interface classes cannot be used directly in Omnis. To use an interface class, create a custom class which implements the interface.
- J2SE/Java SE is currently supported, meaning that you will have access to a Java Virtual Machine and can run any Java Code on this virtual machine with respect to "b". However, J2EE is not supported.
- Java Objects under macOS are supported on version 10.3 (Panther) or above.

Java Example Library

Omnis Studio includes an example library called "example.lbs" that demonstrates the use of Java Objects. This library is referred to throughout this chapter and is designed to be used in conjunction with the Java Package called 'example'. Both the Omnis library and the Java Package can be found in the \Java\JavaCode\example in the main Omnis folder.

Environment Variables

The following assumes that you are the System Administrator of the platform on which you are working, or that you have a basic knowledge of setting environment variables on your respective platform.

To use Java Objects in Omnis, you must define or edit a number of environment variables, including the CLASSPATH and OMNISJVM64 OR OMNISJVM32 depending on whether you are running the 64-bit or 32-bit version of Omnis Studio. These variables are slightly different for each platform and are described in the following sections.

You do not need to add the Omnis Java folder, 'OSXX\java\JavaCode', or any of its sub-folders to the CLASSPATH environment variable since this folder is registered by Omnis automatically. You can place your own Java classes in this folder or any of its sub-folders to register them automatically. However if you want to use any Java classes anywhere on your system you need to include their location in the CLASSPATH.

Supported Characters in CLASSPATH

When setting the CLASSPATH on Linux and macOS, any folder names containing UTF-8 based extended characters should work, but on Windows characters are restricted to extended ASCII only.

Windows

The environment variable OMNISJVM64 OR OMNISJVM32 should be defined to point to the installation of the Java Virtual Machine that you wish to use. The Java Virtual Machine is normally located in the bin\client folder of your J2SE installation. For example:

```
OMNISJVM64 OR OMNISJVM32= C:\jdk8.XXX\jre\bin\client\jvm.dll
```

or

```
OMNISJVM64 OR OMNISJVM32= C:\Program Files\Java\jre8.XXX\bin\client\jvm.dll
```

The CLASSPATH environment variable should also be defined to point to folders that contain any custom classes that you wish to use with Omnis. For example:

```
CLASSPATH=c:\java\myclass;c:\java\myclass1
```

Setting environment variables under Windows

In Windows, you can add or edit environment variables in the System Properties dialog in the Control Panel. On the Advanced tab, click the Environment Variables button, then click New to define a new variable, or select the name of the User or System variable you want to change and click Edit.

macOS

The CLASSPATH environment variable should be defined to point to folders that contain any custom classes that you wish to use with Omnis. For example:

```
CLASSPATH=/Volumes/HD/classes/myclass:/Volumes/HD/classes/myclass1
```

Linux

The environment variable OMNISJVM64 OR OMNISJVM32 should be defined to point to the installation of the Java Virtual Machine that you wish to use. The Java Virtual Machine is normally located in the lib/i386/client folder of your J2SE/Java SE installation. For example:

```
OMNISJVM64 OR OMNISJVM32=/usr/java/jdk8.XXX/jre/lib/i386/client/libjvm.so
export OMNISJVM64 OR OMNISJVM32
```

The CLASSPATH environment variable should also be defined to point to folders that contain any custom classes that you wish to use with Omnis. For example:

```
CLASSPATH=/classes/java/myclass:/classes/java/myclass1
export CLASSPATH
```

The PATH environment variable should be modified to include the path to the bin folder of your J2SE/Java SE installation. For example:

```
PATH=/usr/java/jdk8.XXX/bin:$PATH
export PATH
```

The LD_LIBRARY_PATH environment variable should be modified to point to the lib/i386 and lib/i386/client folders of your J2SE/Java SE installation as follows:

```
LD_LIBRARY_PATH=/usr/java/jdk8.XXX/jre/lib/i386:
/usr/java/jdk8.XXX/jre/lib/i386
/client:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

Object parameters

Java reflection requires that parameter objects exist on the main classpath. To ensure that objects or object references can be passed to a Java method, place the path to the jar file in the CLASSPATH variable.

OMNISCLASSPATH environment variable

You can use the OMNISCLASSPATH environment variable to specify Java classes instead of CLASSPATH. This can increase performance in situations where you have multiple Java applications installed that are not being used with Omnis. Under normal circumstances, the classes for these applications will be added to the Omnis Class Cache, even though they are not being used. By using OMNISCLASSPATH, you will be able to specify only the classes that you wish to use with Omnis.

If OMNISCLASSPATH is not defined, CLASSPATH will be used.

Getting and Setting Environment Variables in Omnis

There are three functions in Omnis that you can use to for manipulate environment variables.

Listing Environment Variables

The listenv() function returns a 2 column list of the Omnis process environment variables. Column 1 contains the variable name and column 2 the variable value. The list is sorted by the variable name column, and is case insensitive. There is no need to define the list first; the returned list has 2 character columns, name and value. For example:

```
Calculate lList as listenv()
# returns a list of environment variables on the current system
```

Setting an Environment Variable

The putenv(*name,value*) function sets the Omnis process environment variable with the specified *name* to the specified *value*. Creates a new environment variable if necessary, and returns true for success, false for failure. For example:

```
Do putenv("OMNISCLASSPATH","C:\Program Files\OmnisSoftware\OSXX\java")
# Creates the environment variable OMNISCLASSPATH and sets its value to C:\Program Files\OmnisSoftware\OSXX\ja
```

Getting the value of an Environment Variable

The getenv(*name*) function returns the value of the Omnis process environment variable with the specified *name*. For example:

```
Calculate lVar1 as getenv("OMNISJVM64 OR OMNISJVM32")
# Returns the location of the JVM, e.g. C:\jdk8.XXX\jre\bin\client\jvm.dll
```

Memory Allocation

There are some Omnis preferences that allow you to set the memory allocation for Java. You can set the properties on the Java tab of the main Omnis preferences, visible in the Property Manager. The properties are:

- $usejavaoptions (boolean)
  when kTrue, the Java Virtual Machine will use the options specified in $javaoptions Omnis preference; if set to kFalse, the JVM will use its own internal memory settings.
- $javaoptions
  You can use this property to specify any switches you wish to pass to the Java Virtual Machine, e.g. to set a system property value, specify -Dproperty=value

In version 8.0.1 and earlier, the $javaoptions preference was limited to 255 chars. This has been extended and the theoretical limit is now the maximum length of an Omnis character variable. You can add the following parameters to the $javaoptions property, assuming $usejavaoptions is set to true.

Original heap size	To specify the original heap size for the JVM add "-Xms <heapsize>" to the $javaoptions property
Maximum heap size	To specify the maximum heap size for the JVM add "-Xmx <maxheapsize>" to the $javaoptions property
Stack size	To specify the stack size for the JVM add "-Xss <stacksize>" to the $javaoptions property—-

Note: The above options are for advanced users only. If you are unsure of what the settings for the above properties should be, you should leave $usejavaoptions set to kFalse.

Note to existing users: the above options replace the Omnis $root preferences $javainitialheap, $javamaxheap and $javathreadstack. The $javaother property was renamed to $javaoptions.

The Java Class Cache

Starting Omnis

Once you have configured your platform to run J2SE/Java SE, start Omnis. There may be a brief pause during startup. This is because Omnis is starting the Java Virtual Machine and collecting information on Java System Classes. This procedure does not occur every time you start Omnis as Java Class information is cached. However, this procedure will occur again if you reset the Omnis Class Cache (see the Advanced Topics section for further details on resetting the Omnis Class Cache).

Quitting Omnis

When Omnis starts up, it scans all of the paths on the CLASSPATH environment variable and creates a cache of all available classes on your system. When Omnis exits, any classes that you have added are appended to the internal Omnis Java class cache. When Omnis is restarted, all classes are read from this class cache.

Reloading Classes

Omnis will detect that you are attempting to add an existing class and will not attempt to add the class a second time. This will not produce an error in your code, which means that you do not have to worry about implementing a "first time" flag when adding classes.

Resetting the Java Class Cache

To reset the Java class cache, you can either delete the cache file manually or use the *Reset class cache* option in the Studio Browser (click on the Studio option in the treelist of the Studio Browser, then click on the Java option to show the *Reset class cache* option). The cache file is called 'jcache1.dat' and can be found in the omnis\java folder.

Creating Java Objects

Creating Java Objects is a two-stage process that is similar to the process of creating Automation Objects in Omnis. A Java Object is first declared and then constructed using the $createobject() method. This process is described in the following sections.

Defining Java Objects

Java Objects can be manipulated in a similar manner to other Omnis Objects. A Java Object can be defined in a variety of ways:

- Declaring an object in the method editor and setting the object subtype to JavaObjs.

or

- Declaring an object reference and initializing it to point to a Java Object using notation.

or

- Declaring an object in the method editor that does not have a subtype and initializing it to a Java Object using notation.

It is *not* possible to use a Java Class that contains either the string 'omnis' or 'javacore' in its name.

Defining Java Objects via the Method Editor

To create a Java Object in the method editor, simply declare an object variable and set its subtype to the desired Java Objects Class. The Java Class can be chosen from the "Select Object" dialog. To select a Class, navigate the Object Selection Tree until the desired class is found, highlight the class and click OK. The Object Selection Tree is split into two sections under the "JavaObjs" heading. These are "ClassPath" and "System". Any classes that have been placed in any of the folders that are specified in the CLASSPATH environment variable will appear in the "ClassPath" section. All other available classes will appear in the "System" section.

Defining Java Objects using Object References

Object References are similar to object variables but you are responsible for the allocation and de-allocation of the object. Object References are declared in the Method Editor but do not have a subtype. The Object that is associated with an Object Reference is determined at runtime when it is allocated via notation:

To declare an Object Reference, simply declare a variable in the Method Editor and set its type to "Object Reference".

To allocate an Object to an Object Reference, you must use notation. As notation strings for Java Objects can be very long (due to the large hierarchies that are present in some Java Packages) the "Object Selection Tree" is displayed as a helper window when entering *Java Objects* notation. If a class is selected from this tree then a large portion of the notation is created for you.

All Java Object notation begins with *$extobjects.JavaObjs Library.$objects*. The remainder of the notation string is dependant on the Java Class that is being used. For example:

```
$extobjects.JavaObjs Library.$objects.//JavaObjs\ClassPath\zipdemo\zipdemo//
```

As you can see from the above, the Object Selection Tree has completed the notation string by adding *JavaObjs\ClassPath\zipdemo\zipdemo*. In addition to this, a call to the $newref() method has to be added manually. This method is responsible for allocating an Object to the Object Reference. The command reads as follows:

```
Calculate lzipreference as $extobjects.JavaObjs Library.$objects.//JavaObjs\ClassPath\zipdemo\zipdemo//.$newre
```

When the above command is executed, a zipdemo Java Object will be created and assigned to the Object Reference *lzipreference*. It is your responsibility to ensure that this Object is deleted using the $deleteref() method once it has served its purpose. For example:

```
Do lzipreference.$deleteref()
```

Defining Java Objects via Notation

Java Objects can also be defined using the object type via notation. To define a Java Object via notation, simply follow the instructions in the previous section but use an *Object* type instead of an *Object Reference* type and call the *$new()* method instead of the *$newerf()* method.

Constructing Java Objects

Once a Java Object has been declared, it needs to be constructed. Declaring a Java Object simply tells Omnis that the Object exists. Constructing a Java Object using $createobject() actually performs the work of constructing the Object inside the Java Virtual Machine.

By default all Java Objects will have at least one $createobject() method which takes no parameters. This method can be used to create a Java Object as follows:

```
Do lzipdemo.$createobject()
```

Other Java Objects may have more than one $createobject() method. In this case, each $createobject() method corresponds directly to a Java Constructor for the class that is being used. For example, the java.lang.String class has the following constructors:

String()
String(String original)
String(byte[] ascii, int hibyte)
String(byte[] bytes)
String(char[] value)
String(byte[] bytes,int offset,int length,String enc)
String(StringBuffer buffer)
String(byte[] bytes,String enc)
String(char[] value,int offset,int count)
String(byte[] ascii,int hibyte,int offset,int count)
String(byte[] bytes,int offset,int length)

These are mirrored in Omnis by the following $createobject() methods:

Calling any of above $createobject() methods will invoke the corresponding Constructor in Java and create the Java Object in the virtual machine. For example, calling:

```
Do mystring1.$createobject(mystring)
```

will invoke the **String**(String original) java.lang.String Constructor if *mystring* is an Omnis Java Object with a java.lang.String subtype that has been previously declared and created.

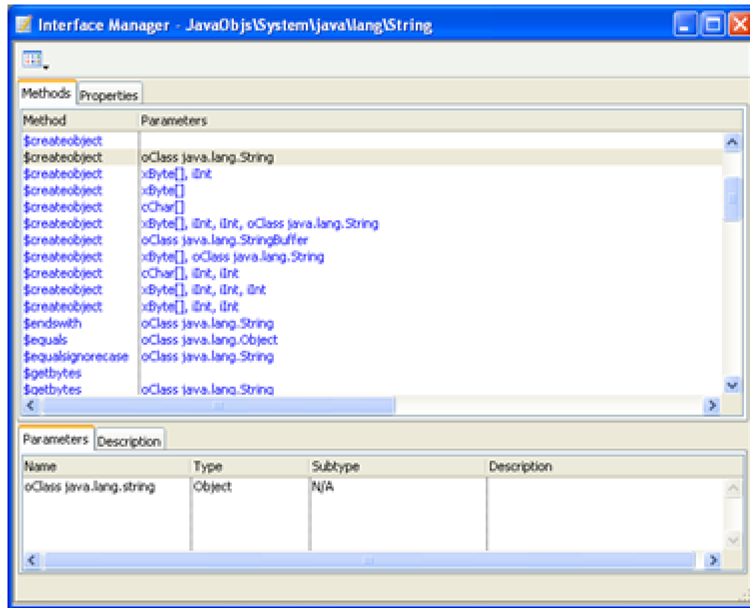


Figure 3:

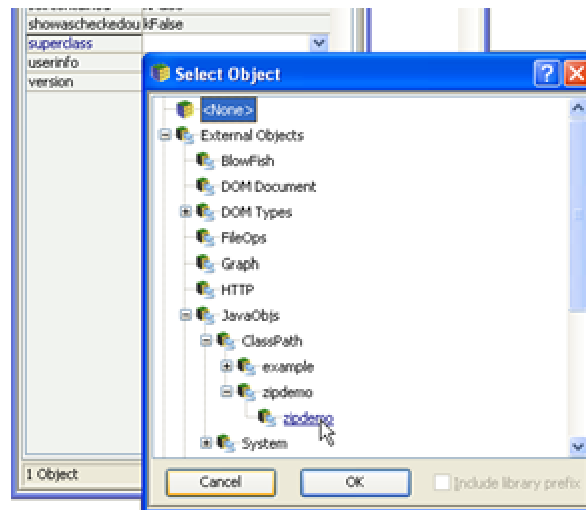


Figure 4:

Figure 5:

Subclassing Java Objects

Java Objects can be subclassed by creating an Omnis Object class (using the New Class>>Object option in the Studio Browser) and setting its $superclass property to the name of a Java Class using the Object Selection Tree.

As a result of the above, an object class called *zipobject* becomes a subclassed object of *zipdemo* and inherits all of its methods as follows:

Subclassed Java Objects can be used to define Object and Object Reference types in the same manner as Java Objects, described in the previous section.

Using Java Objects

The following sections describe how Omnis Data Types are converted to Java Data Types, how to pass parameters from Omnis to Java, how to handle Return Values and how to retrieve data from Java Objects. Other topics such as Error Handling and Cache Control are also discussed.

Parameter Data Types

To call Java Objects effectively, you need to know how Omnis Data Types are mapped to Java Data Types when passing parameters. The following table shows how Omnis Data Types are converted to Java Data Types when used as parameters to Java Object Methods.

Omnis Type	Converts to {Java Type)
Character*	char or char[]
Binary*	byte or byte[]
Number Long integer*	Int, long or byte
Number Short integer	short
Number floating dp* or Number dp	float or double
Boolean	Boolean
List	Java Array
Object	Java Object
Object Reference	Java Object

Those marked '*' are known as *Overloaded Types* as they are compatible with more than one Java Data Type.

As you can see from the above, most of the Data Types in Omnis can be coerced into multiple Java Data Types. Although this coercion happens automatically, it is useful to be aware of *Overloaded Types* as they provide you with greater flexibility when programming with Java Objects. For example, if you wish to construct a java.lang.Byte object, you could define a binary variable, load the byte into the variable and call $createobject() with this variable. However, it is much quicker to simply call $createobject() using the integer value of the byte. For example:

```
Do mybyte.$createobject(65)
```

will create a Java Byte object whose value is 65.

Passing Parameters

Java Object Parameters can be categorized into two basic types. Simple Parameters and Complex Parameters. Simple parameters are parameter types that are compatible with java base types. All other parameter types are considered to be Complex.

Java Object Simple Parameter Types

Omnis data type	converts to (Java base type)
Character	char/char[]
Binary	byte/byte[]
Number Long integer	int/long/byte

Omnis data type	converts to (Java base type)
Number Short integer	short
Number floating dp or Number dp	float/double
Boolean	Boolean

Java Object Complex Parameter Types

Omnis data type	Can be converted to (Java type)
List	an array of any Java type depending on list content
Object	an Object of any Java type depending on content
Object Reference	an Object of any Java Type depending on the content of the referenced Object

Passing Simple Parameter Types

Passing Simple Parameter types to a Java Object is relatively simple. You call the Java Object method with the parameter(s), as you would call any object method in Omnis.

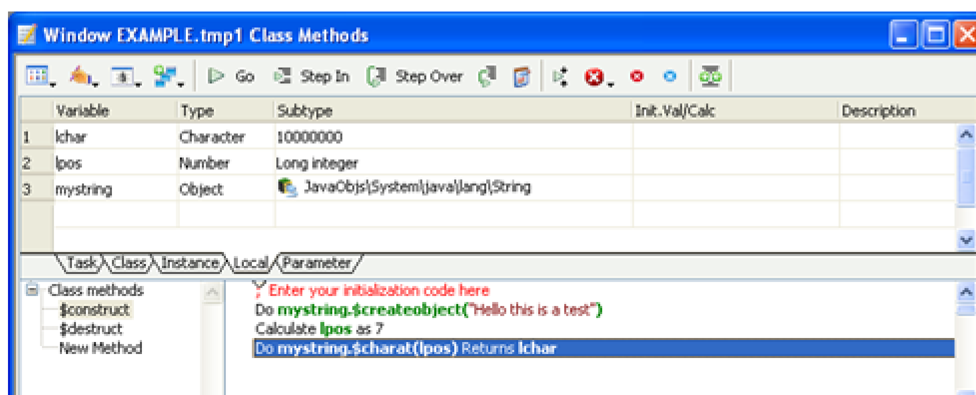


Figure 6:

In the above example, the Simple Parameter Type lpos is passed to the java.lang.String charat() function. This function returns the character value 'o' (the letter o) as this character resides at index 4 of the Java String.

Passing Complex Parameter Types

Complex Parameter types such as lists and Objects can be passed to Java Objects as well. However, these Parameter types must be initialized before being used.

List Types

Omnis lists must contain valid data before being passed to a Java Object. Note that if you use a list that has more than one column, only the first column will be used. Omnis lists which do not contain valid data will cause *Java Objects* to return an error. See the *Error Handling* section for more information.

Omnis converts Single Column Lists to Java Arrays based on the type of the data that is stored in the list. This conversion can be described by the following table:

Omnis List	Converts to (Java Array Type)
Omnis list containing Number Long integer c	int[] or long[]
Omnis list containing Number floating dp types	float[] or double[]
Omnis list containing Number Short integer types	short[]
Omnis list containing Boolean types	boolean[]
Omnis list containing Object Reference types	Array of Java Objects

As you can see from the above, *Overloaded Types* are also supported by Omnis Lists. For an explanation of *Overloaded Types*, see the *Parameter Data Types* section. It should also be noted that char[] and byte[] are absent from the above. This is because Java char[] and byte[] types are created directly from Omnis Character and Binary Variables.

Omnis will always attempt to determine the type of Array that you are trying to pass as a parameter by examining the data in the List. If a list contains Java Objects, Omnis will attempt to determine the type of the objects in the list. If the type of all the objects is found to be the same, it is assumed that you are passing a parameter of that type. For example, if you create an Omnis list of java.lang.String Objects, Omnis will assume that you are passing a String Array Parameter to Java. However, if any of the Objects are found to be of different types, Omnis will assume that you are passing a generic Object Array (java.lang.Object) to Java.

When calling Java methods that take arrays as parameters, it is important to make sure that the Omnis list that you pass to the method in question contains appropriate data. Lists which do not contain appropriate data will cause the function call to fail with an error. (See the *Error Handling* section.) For example, if you create an Omnis List with 12 java.lang.String Objects and add an extra java.lang.Byte object by mistake, your method call will fail if you are attempting to call a Java Function that accepts a java.lang.String Array as a parameter.

An example of how to use Omnis lists as parameters to Java can be found in the example.lbs Omnis library that accompanies Omnis (in the Java\JavaCode folder). The window class *arrayparams,* which is part of this library, demonstrates how to call a Java method that takes an Array of java.lang.String and an Array of int as parameters.

Important Note: When creating a list of Java Objects, Object References should be used. This is because standard objects cannot be used in Omnis Lists. Once you have finished using your list of Object References, remember to delete each reference in the list by calling $deleteref() for each reference. Please refer to OO Programming chapter in the Omnis Programming manual for further information about Object References.

Object Types

Java Objects should be "created" using $createobject() before being passed as parameters to other Java Objects. Passing an uninitialised object to a Java Function will return an error. (See the *Error Handling* section.)

An example of how to use Java Objects as parameters to other Java methods can be found in the example.lbs Omnis library that accompanies Omnis (in the Java\JavaCode folder). The window class *objectparams* in the example library demonstrates how to call a Java method that takes a variety of Java Objects as parameters.

Returning Values From Java Methods

The following table shows how Java Return Types are converted to Omnis Data Types.

Java Type	Converts to (Omnis Type)
char[] or char	Character
byte[] or byte	Binary
int or long	Number Long integer
float or double	Number floating dp or Number dp
short	Number Short integer
Boolean	Boolean
Java Arrays (except char[] and byte[])	List
Objects	Object references

Note: this is a reversal of the table shown in the previous section.

To return a value from a Java Object method, use the Method Editor to specify the variable that will be used to hold the return value. You must ensure that the Omnis variable that you are using is of the correct type to accept the value that will be returned from Java. If this is not the case, an error will occur and the hash variables #ERRCODE and #ERRTEXT will be set accordingly (see *Error Handling*).

The following shows a String object being returned from Java. The object is created via the use of the Java String "substring" function and is returned as an Object Reference.

Note: The $getobjectvalue method is used to retrieve the character value of the lstringref Object Reference. This method is described in the next section.

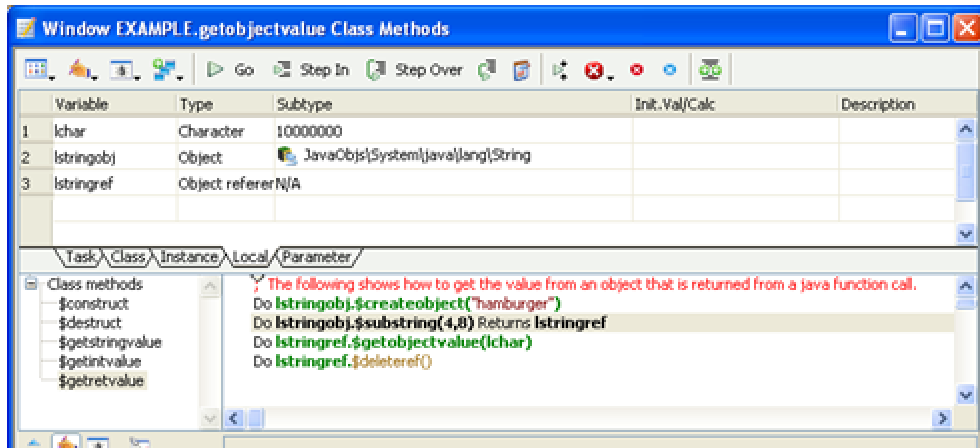


Figure 7:

Data type mapping and performance

There is a Boolean library preference called $javareturnsnative that effects the way in which arrays of Java data types are returned from Java to Omnis. When the property is set to kTrue, Java object methods called from the library return native Omnis types if a suitable conversion exists, rather than an object reference to a Java object. For example, a method returning a Java String object returns an Omnis Character value. Setting the value to kTrue also benefits from the improved performance.

Existing libraries have the $javareturnsnative property set to kFalse for backwards compatibility, but all new libraries will have a default value of kTrue.

Note: If you wish to set the $javareturnsnative property to kTrue in an existing library, you must change the types of the Omnis variables used to return values from Java. For example, when a Java method that returns a java.lang.String is called from an Omnis library with $javareturnsnative set as kFalse, the Omnis return type will be Object reference. Changing the $javareturnsnative property to kTrue means that the return types for these methods need to be changed to the Omnis Character type.

Getting Values From Java Objects

All Java Objects in Omnis have the $getobjectvalue() method. The purpose of this method is to attempt to get the content of the current Java Object and to convert this content into a format that Omnis understands. This is not always possible as many Java Objects do not have a corresponding type in Omnis. However, most of the Objects in Java can usually be coerced into an object type that can, in turn, be converted into an Omnis Data type using $getobjectvalue(). In addition to this, some Objects in Java have their own versions of $getobjectvalue which are designed to convert object content. For example, the java.lang.Float class has functions such as byteValue() and intValue() to coerce the float content to byte and int types.

$getobjectvalue can be used in the following manner:

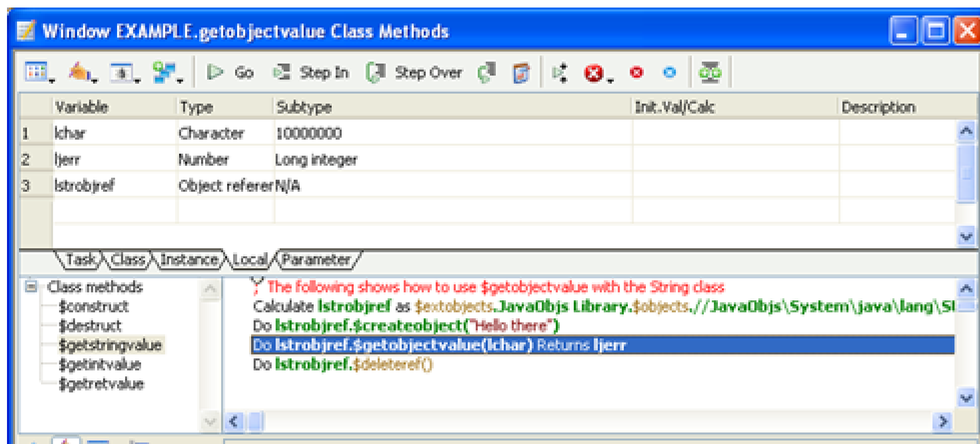


Figure 8:

The above example creates a Java String Object with the value “Hello there”. The call to $getobjectvalue() extracts this value and assigns it to the local variable *lchar*. The local variable *ljerr* is a long integer which is used for error handling. If $getobjectvalue() fails to convert the Object value, *ljerr* will contain an error code, otherwise the value 1 (kJavaCoreOK) is returned.

The $getobjectvalue() method will accept all of the Simple Parameter Types as parameters. Therefore, the format of the $getobject-value() method call is the same regardless of the type of data that is being returned. For example, the following is very similar to the previous example:

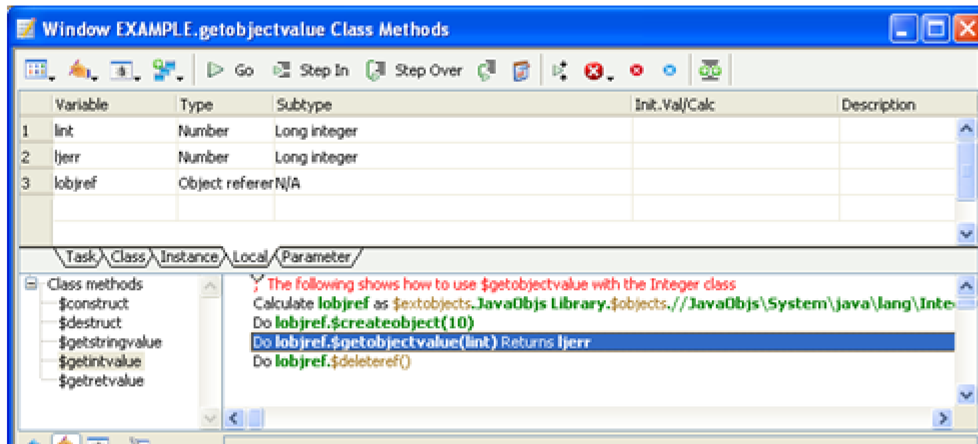


Figure 9:

The $getobjectvalue() method uses the same process of data conversion that is used for Return Values. For a further explanation of this process, see the *Returning Values From Java Methods* section.

Adding Java Classes at Runtime

When Omnis is started for the first time, it will create a cache file for all Java System and Class Path classes automatically. This cache improves performance, as it removes the necessity to interrogate the Java Virtual Machine to find out which classes are available every time Omnis starts up.

Once Omnis has started up, and loaded all the Java classes in your Omnis\Java folder and on your CLASSPATH and System, you can load Java classes while Omnis is running using the $addclass() method. The syntax of this method is as follows:

```
Do JavaObjs Library.$addclass(classgroup, classfilename [,classpath])
```

The classgroup parameter

The *classgroup* parameter specifies the Java Objects category or group in the Java Objects class list that the loaded class will be added to. The group is one of the following constants:

kJavaObjsClassPathGroup: Specifies the classpath category.

kJavaObjsWebServicesGroup: Specifies the webservices category.

kJavaObjsWebServicesClientGroup: Specifies the webservices client category.

The classfilename parameter

The *classfilename* parameter can contain one of the following:

- A fully qualified pathname of the class file to be loaded.
- A fully qualified folder name that contains one or more class files.
- A fully qualified Jar filename.

If a single class is specified, only that class is loaded. If a folder is specified, then all class files within the folder are loaded. If a Jar file is specified, all classes within the Jar file are loaded.

Note: Any classes that are specified *must* already reside on the Java Classpath. To make the best use of $addclass(), your CLASSPATH environment variable should include any paths that you may want to load classes from while Omnis is running. If you are loading a Jar file that itself requires the use of other Jar files, you should ensure that these dependencies are also listed in your CLASSPATH environment variable.

The classpath parameter

The *classpath* parameter specifies the particular classpath that your class resides on. You do not have to specify the entire contents of the CLASSPATH environment variable in this parameter, it is only necessary to specify the classpath that contains the class that you are adding. For example, if your CLASSPATH environment variable contains the path "C:\classes" and you are attempting to add the class "c:\classes\zipdemo\zipdemo.class", you only need to specify "c:\classes" as your classpath parameter. If you do not specify a classpath parameter, the default classpath is used for the group that you have specified. For example, if you have specified a group constant of "kJavaObjsWebServicesGroup", Omnis will use the path of the Omnis Web services directory as the classpath.

The built-in classpath definitions for each group are represented by Omnis constants as follows:

```
kJavaObjsClassPathGroup: <Omnis Folder>\Java\JavaCode
kJavaObjsWebServicesGroup: <Omnis Folder>\Java\WebServices
kJavaObjsWebServicesClientGroup: <Omnis Folder>\Java\WebServices\client
```

Examples

The following code shows some examples of using the $addclass() method:

To load the class "zipdemo.class" that resides on the classpath "c:\classes", use:

```
Do JavaObjs Library.$addclass(kJavaObjsClassPathGroup,"c:\classes\zipdemo\zipdemo.class","c:\classes")
```

The above will add zipdemo.class to the JavaObjs ClassPath group. To add the same class to the Web Services group, use the following:

```
Do JavaObjs Library.$addclass(kJavaObjsWebServicesGroup,"c:\classes\zipdemo\zipdemo.class","c:\classes")
```

To load the class example.class from the Omnis JavaCode folder, use:

```
Do JavaObjs Library.$addclass(kJavaObjsClassPathGroup,"<insert your omnis installation folder pathname here>\J
```

To load all of the classes in the "myclass" folder on the classpath "c:\classes" use:

```
Do JavaObjs Library.$addclass(kJavaObjsClassPathGroup,"c:\classes\myclass","c:\classes")
```

Error Handling

Java Objects makes full use of the #ERRCODE and #ERRTEXT Omnis variables. If an error occurs while executing a Java Object Method, the error variables identify the error on return from the method. In addition to this, each Java Object has an error status which you can examine using the $getlasterror() method.

The following example, which is in the example Omnis library, shows how to use $getlasterror() to retrieve the error status of an object.

#ERRCODE and #ERRTEXT can be used as follows:

Both #ERRCODE and $getlasterror will produce the value 1 (kJavaCoreOK) after a successful Java method call. If an error occurs while calling a Java method, an error code is generated.

Error Codes

Java Objects inherits all of the error codes that are supported by the *JavaCore*. The *JavaCore* is the Omnis Java Engine which is responsible for managing calls to the Java Virtual Machine. Both *Java Objects* and the *JDBC DAM* are clients of the *JavaCore*. All the error codes supported by the JavaCore are listed in the Omnis Catalog (F9) under 'JavaCore'. In addition, error codes specific to *Java Objects* are also listed.

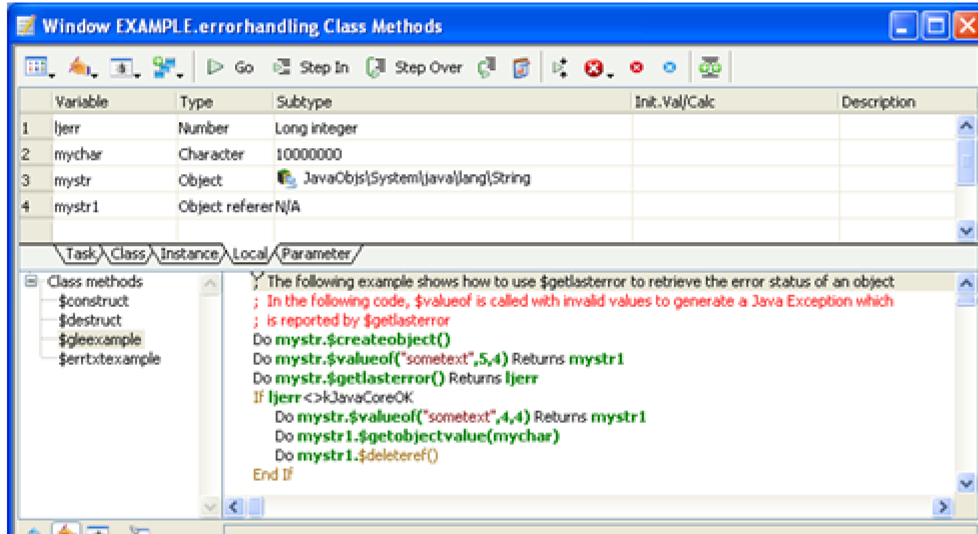


Figure 10:

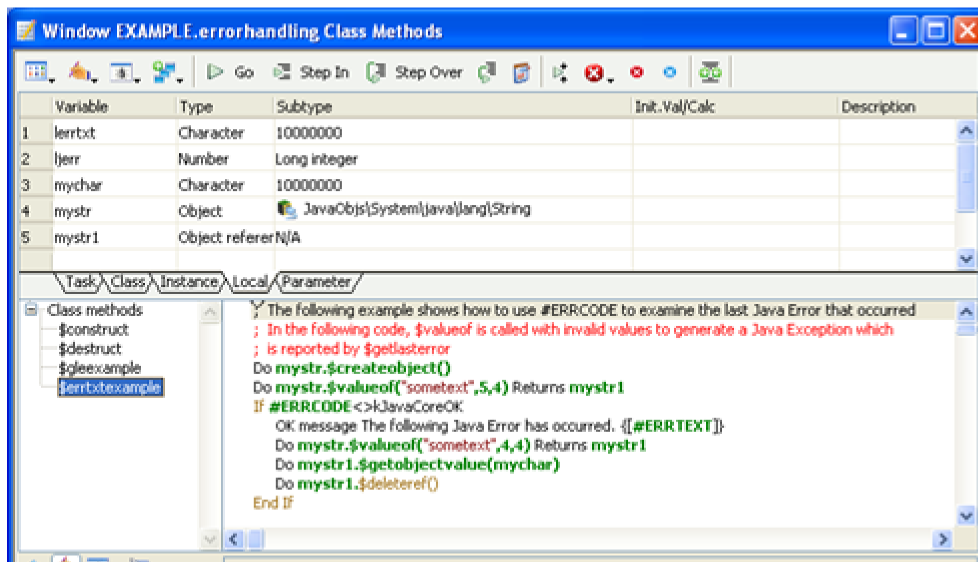


Figure 11:

Development Tips

The following information may be useful when developing with *Java Objects* in Omnis.

Ensure that your Java Code is working correctly before calling it from Omnis. Java Exceptions can be very difficult to track down as it is not possible to use a Java Debugger once a Java object is embedded in Omnis. *Java Objects* will report exceptions in Java Code. However, because *Java Objects* uses introspection to call functions in your Java Class, it will only be able to tell you that an Exception occurred while executing your method and the actual nature of the Exception will not be known.

Use Object References with care. Remember that any Java Object that is returned to Omnis is returned as an Object Reference and it is your responsibility to delete this reference using the $deleteref() method. Also remember that arrays of Java Objects are returned to Omnis as lists of Object References which also need to be deleted once they have been used.

It is possible to pass Nested Object Arrays to Java and have Java return Nested Object Arrays to Omnis. Nested Object Arrays are passed to Omnis as a list of Omnis lists which in turn contain Java Objects. Nested Object Arrays are returned from Java in the same format. See the *Nested Object Arrays* section for further information.

Method Overloading and Pattern Matching

Although Omnis Studio does not normally support the overloading of method names, *Java Objects* are an exception as method overloading is usually unavoidable when dealing with even the most basic of Java Classes. The following sections discuss how Omnis deals with overloaded methods and how it is possible to manually call an overloaded method directly from Omnis.

Pattern Matching

Omnis uses pattern matching in order to determine the overloaded method that you are trying to call. Put simply, Omnis examines the parameters that you pass to a method and attempts to call the method whose parameters most closely match the parameters that you are passing. For example, consider the following Java methods:

```
myfunc(int p1)
myfunc(float p1)
```

Calling myfunc from Omnis using an Omnis number type set to Long Integer will call myfunc(int p1). Calling myfunc from Omnis using an Omnis number type set to Floating dp will call myfunc(float p2).

Overloaded Data Types

An extra level of complexity is added to Pattern Matching with the support of Overloaded Data Types. An Overloaded Data Type is an Omnis Data Type that has more than one matching data type in Java. Overloaded Data Types are split into three groups as follows.

The Number Long Integer Type

The Omnis “Number Long Integer” type can be translated to the following Java Types.

- int
- long
- byte

If a Number Long Integer Omnis type is passed to an overloaded method, the following occurs:

- Omnis attempts to find an overloaded method that has the same number of “int” parameters that you are passing, i.e. if you are calling a method with (int,boolean,int), Omnis will try to find an overloaded method which matches this Parameter Search Pattern.

- If a match is not found, Omnis searches for a method that has a matching number of "long" parameters, i.e. Omnis will modify the Parameter Search Pattern in an attempt to find a match using combinations of int and long types. Therefore, if (int,boolean,int) is not found, Omnis will look for the following:

  (int,boolean,long)
  (long,boolean,int)
  (long,boolean,long)

- Finally, if this fails, Omnis searches for a method that has a matching number of "byte" parameters, i.e. Omnis will modify the Parameter Search Pattern in an attempt to find a match using combinations of int and byte types. Therefore, if 'b' does not find a compatible method, Omnis will look for the following:

  (byte,boolean,int)
  (int,boolean,byte)
  (byte,boolean,byte)

Note that any method that matches the "int" Parameter Search pattern will always be called if it is available. For example, consider the following Java methods:

myfunc(int p1)
myfunc(long p1)
myfunc(byte p1)
myfunc(boolean p1)

calling myfunc using a long integer type will always call myfunc(int p1) in Java. If we remove this method:

myfunc(long p1)
myfunc(byte p1)
myfunc(boolean p1)

calling myfunc using a Long Integer type will always call myfunc(long p1), and so on.

The Number Floating dp Type

The Omnis Number Floating dp type can be translated to the following Java Types.

- float
- double

If a Number Floating dp Omnis type is passed to an overloaded method, the following occurs:

- Omnis attempts to find an overloaded method that has the same number of "float" parameters that you are passing, i.e. if you are calling a method with (float,boolean,float), Omnis will try to find an overloaded method which matches this Parameter Search Pattern.
- If a match is not found, Omnis searches for a method that has a matching number of "double" parameters, i.e. Omnis will modify the Parameter Search Pattern in an attempt to find a match using combinations of float and double types. Therefore, if (float,boolean,float) is not found, Omnis will look for the following:

  (float,boolean,double)
  (double,boolean,float)
  (double,boolean,double)

Note that any method that matches the "float" Parameter Search pattern will always be called if it is available. For example, consider the following Java methods:

myfunc(float p1)
myfunc(double p1)
myfunc(boolean p1)

calling myfunc using a floating dp type will always call myfunc(float p1) in Java. If we remove this function:

myfunc(double p1)
myfunc(boolean p1)

calling myfunc using a floating dp type will always call myfunc(double p1).

The Java Object Type

The Omnis Java Object type can be translated to the following Java Types.

- < The object that you are attempting to pass as a parameter (e.g. java.lang.String)>
- java.lang.Object

When matching objects, Omnis will always try to find a method which takes an object parameter that matches the type of the object that you are passing. So if you pass a java.lang.String object to an overloaded method, Omnis will look for a match using java.lang.String as a search pattern. If Omnis fails to find a matching method, a method which takes the generic type java.lang.Object as a parameter will be called, if such a method is available, i.e. if Omnis searches for a method using the Parameter Search Pattern (String,boolean,String) and a matching method is not found, Omnis will look for (Object,boolean,String) followed by (String,boolean,Object) followed by (Object,boolean,Object).

Note that any Java method that has a parameter list which matches the Objects that you are using as parameters will always be called if it is available. For example, consider the following Java methods:

myfunc(String p1)
myfunc(Object p1)
myfunc(boolean p1)

calling myfunc using an Omnis object whose subtype is set to "JavaObjs\System\java\lang\String" will always call myfunc(String p1) in Java. If that function is removed:

myfunc(Object p1)
myfunc(boolean p1)

calling myfunc using an Omnis object whose subtype is set to "JavaObjs\System\java\lang\String" will always call myfunc(Object p1).

Char and Byte Pattern Matching

The following shows how Omnis converts Character and Binary data types to Java Data types when dealing with overloaded methods.

Omnis		Java
character types (length > 1)	*are converted to*	char[]
binary types (length > 1)	*are converted to*	byte[]
character types (length <= 1)	*are converted to*	char
binary types (length <= 1)	*are converted to*	byte

As a result of the above, you should be aware that it is not possible for Omnis to call an overloaded Java function that takes char[] as a parameter using a single character. This is demonstrated in the following example.

```
myfunc(char p1[])
myfunc(char p1)
```

Following the rules for data conversion, calling myfunc with a single character will call myfunc(char p1) while calling myfunc with a string of characters will call myfunc(char p1[]). Thus, in this situation it is not possible to call myfunc(char p1[]) with a single character as myfunc(char p1) will always be called if the length of the data that you are passing is 1. However, it is possible to force Omnis to call myfunc(p1[]) with a single character if you call the overloaded method directly. The procedure for doing this is discussed in the next section.

Note: If the length of an Omnis character type is 0, it will be converted into a Java char type whose value is 0. If the length of an Omnis binary type is 0, it will be converted into a Java byte type whose value is 0.

Calling Overloaded Methods Directly

In some rare cases, it may be necessary to call overloaded methods directly. Consider the following example.

You are trying to call overloaded functions in the java.lang.String Class. The functions are defined in Java as follows:

```
valueof(float f)
valueof(double d)
etc..
```

You have created an Omnis Object for the String class and you are attempting to call valueof(double d) by passing an Omnis Floating dp type as a parameter. However, because of the Pattern Matching and Data Type Matching rules previously discussed, valueof(float f) is always called.

In situations such as this, it is necessary to call valueof(double d) directly. In order to do this, the method can be called by its "real name". All overloaded methods have an associated real name. This consists of the method name followed by a string of alpha numeric characters and is used internally by Omnis for pattern matching purposes. The "real name" of an overloaded method can be found by opening the Interface Manager for the Java Object (o open the Interface Manager, right-click on the object in the method editor and select Interface Manager). From the Interface Manager, select the required overloaded method that you wish to call and click on the description tab. This will display the "real name" of the overloaded method as follows:

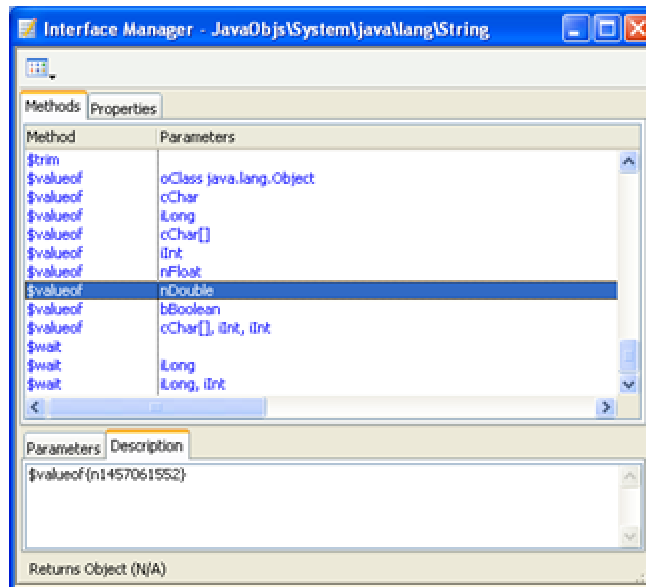


Figure 12:

The above shows the interface manager for the "java.lang.String" class. If you were to call the $valueof method using an Omnis Number floating dp type, Omnis would always call $valueof(nFloat) as this is the closest possible match to the type of data that you are passing to Java. If you wish to call $valueof(nDouble), you must use the "real name" of this method which in this case is $valueof{n1457061552}.

Nested Object Arrays

Java Objects support Nested Object Arrays as parameters to Java methods and can also interpret Nested Object Arrays that are returned from Java. A Nested Object Array is an array of type java.lang.Object that is capable of storing any Java Object or Array of Java Objects.

To use a nested Object array you must create an Omnis List variable and define it using a Binary variable. You can then add Java Objects and Lists of Java Objects to the list. Once your list is complete, you may pass it to any Java method that accepts java.lang.Object array as a parameter.

Nested Object Arrays can also be returned by Java methods. A Nested Object Array is returned to Omnis as a Binary list. This list may contain both object references and embedded lists.

An example of how to use Nested Object Arrays can be found in the example.lbs Omnis library that accompanies Omnis (in the Java\JavaCode folder). The window class *nestedobjects* demonstrates how to call a Java method using a Nested Object Array. The example begins by creating an Omnis list that has the following structure:

Line	Content
1	List of java.lang.String Objects
2	java.lang.Float Object
3	List of java.lang.Integer Objects

The above list is passed to the *exnested* Java method. This method copies the above list structure and adds another array of java.lang.String objects to the end of the list. The content of each element is displayed in the *nestedobjects* main window.

Modifying The System Package List

Before attempting the following, the Java Object Cache should be cleared and Omnis should be shutdown if it is running. Modifying the Omnis Studio System Package list may cause unpredictable behavior and is not supported by OmnisSoftware. The following is simply provided for those readers who wish to experiment with extra functionality. Users attempting the following procedure should make a copy of the "jfilter.txt" file before proceeding. This file can be found in the Java folder under the main Omnis folder.

Adding Extra System Packages

When Omnis starts up, the following default System Class packages are available for use with Java Objects:

java.lang
java.io
java.util
java.math

This list can be increased by editing the "jfilter.txt" file that can be found in the Java folder under the main Omnis folder. The default content of this file is as follows:

```
System\\java\\lang;System\\java\\io;System\\java\\util;System\\java\\math
```

The above is a list of paths to Java System Class packages. Each path is separated with a semicolon. To add a package to the list, simply insert it at the beginning as follows:

To add the java.net package to the list of available System Java Classes, replace the '.' with a '\' and prefix with "System\" so that

```
java.net
```

becomes

```
System\java\net
```

The above can then be added to the existing filters in the "jfilter.txt" file as follows:

```
System\java\net;System\java\lang;System\java\io;System\java\util;System\java\math
```

Removing System Packages

To remove a package, delete the relevant path from the path list. Note that deleting the "jfilter.txt" file completely will make all Java System packages available. However, this will significantly worsen performance and is not recommended.

Overloaded Types

The following sections provide detailed descriptions of the Overloaded Types used by *Java Objects*.

The Omnis Character Type

The Omnis Character type can be converted into the following java types:

char
char[]

This allows you to call any Java object that takes either a char or char[] as a parameter with an Omnis Character variable. You may also place Java return values that are of type char or char[] into an Omnis Character variable.

In the case of "char[]" parameters, the Omnis Character variable is automatically converted into a Java char array (or vice versa if you are dealing with return values). This allows objects such as the Java String to be created easily from Omnis. For example:

```
Calculate mychar as "hello this is a test"
Do mystring.$createobject(mychar)
```

The above will invoke the Java String Constructor **String**(char[] value)

If you attempt to call a Java function that takes a single character as a parameter and the Omnis variable that you are passing contains more than one character, Java will only receive the first character of the string.

The Omnis Binary Type

The Omnis Binary type can be converted into the following java types:

byte
byte[]

This allows you to call any Java object that takes either a byte or byte[] as a parameter with an Omnis Binary Variable. You may also place Java return values that are of type byte or byte[] into an Omnis Binary variable.

In the case of "byte[]" parameters, the Omnis Binary Type is automatically converted into a Java byte array (or vice versa if you are dealing with return values). This allows objects such as the Java String to be created easily from Omnis. For example:

```
Do mystring.$createobject(mybinval)
# where mybinval is a binary Omnis variable
```

The above will invoke the Java String Constructor **String**(byte[] bytes)

If you attempt to call a Java function that takes a single byte as a parameter and the Omnis variable that you are passing contains more than one byte, Java will only receive the first byte of the data that you are passing.

The Omnis Number Long Integer Type

The Omnis Number Long Integer type can be converted into the following java types:

Int
long
byte

This allows you to pass Omnis Number Long integer variables to Java functions which accept these types.

When dealing with return values, Java int and long types can be converted into the Omnis Number Long integer type. However, all byte return values are always converted to the Omnis Binary type.

The Omnis Number Floating dp Type

The Omnis Number Floating dp type can be converted into the following Java types:

Float
double

This allows you to pass Omnis Number floating dp variables to Java functions which accept these types.

Frequently Asked Questions

This section aims to answer some common questions about the Java installation and use with Omnis Studio.

Q. I have created an Omnis library which uses Java Objects and have moved this library to another machine. Now when I examine Objects in the method editor using the interface manager, no methods are displayed. Why is this?

A. The Java Virtual Machine has failed to start on the machine that you are now using. This can be confirmed by looking at Omnis Studio Trace Log. Make sure that you have correctly setup your OMNISJVM64 OR OMNISJVM32 environment variable. Also, if you are running under Linux or Solaris, make sure that you have modified both the PATH and the LD_LIBRARY_PATH environment variables. Further information on environment variable settings can be found at the beginning of this chapter.

Q. I have my own Java Package which I have placed on the class path. I have been using this package successfully with Java Objects but I have just compiled and updated my code and Omnis has not recognized the changes. What could be wrong?

A. Omnis Studio caches Java classes and method names to improve performance. Once a java class has been loaded, it will not be loaded again until Omnis is restarted. If you modify your Java Code while Omnis is running, the changes you make will not become apparent until you restart Omnis.

Q. Does Omnis recognize jar files?

A. Omnis will recognize and load any jar files that are located in a folder which has been specified in the CLASSPATH environment variable. All of the classes in your jar file will be available via the *Object Selection Tree* which is displayed when you create the object variable in the method editor.

Chapter 4—Omnis .NET Objects

Support for Microsoft .NET is provided in Omnis Studio using the .NET Objects external component. You should note that this component is now deprecated in Omnis Studio 10.x or above, and is currently only provided for backwards compatibility in existing Omnis Studio libraries that use .NET. You should not use the .NET Objects external component for new applications in Omnis Studio 10.x or above.

This chapter describes how you can access .NET Objects using the .NET Objects component available in some editions of Omnis Studio. It assumes that you have both a basic knowledge of Omnis Studio notation, creating Object references, and a basic knowledge of the .NET Language.

Introduction

The *.NET Objects* component for Omnis Studio allows you to integrate .NET functionality into your Omnis applications. You can create Omnis Objects based on .NET core functionality or third-party class libraries and call their methods in your Omnis code further extending the power and versatility of Omnis Studio. The Omnis .NET Objects component is available for the Microsoft Windows only at present.

Why use .NET

The Microsoft .NET Framework provides a library of prepackaged functionality, including base classes and various APIs, that can be reused and integrated into your Omnis applications. The base classes provide standard functionality such as input/output, string manipulation, security management, and network communications.

In addition to the base classes provided in the .NET Framework, there are thousands of class libraries available from third-party developers providing a broad range of functionality, available to Omnis developers via the .NET Objects component. These can be located in your .NET Framework folder and added into Omnis using a $addclass() method call.

Software Requirements

Omnis Studio

To use the Omnis .NET Objects component, you need Omnis Studio 4.3 or higher. The component is not supported in older versions of Omnis Studio.

Microsoft .NET Framework

In order to use the .NET Objects component in Omnis Studio, you must download and install the Microsoft .NET framework: the version of .NET framework must be appropriate for the version of Windows and Omnis Studio you are running.

See the Microsoft .NET web site for further details on system requirements for the .NET framework.

- https://dotnet.microsoft.com/

For further details about the .NET framework, for some useful tutorials, and to download the version you need, go to:

- https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet-framework

Setting up

The Omnis .NET Objects component is installed as part of the standard Development version of Omnis Studio (Windows only). The component itself is comprised of two DLLs:

1. **OMDOTNET.DLL**
   The Omnis .NET Objects component is located in the root of your Omnis Studio folder (in Program Files), but must be registered with .NET; see below.
2. **OMDNOBJS.DLL**
   This is an Omnis Studio external component and is located in the XCOMP folder within the Omnis Studio development tree.

Registering the component

The Omnis .NET Objects component (OMDOTNET.DLL) is installed into the root of your Omnis Studio folder, but *it must be registered with .NET* in order to work. If you have Admin rights this can be done from a command line prompt (DOS box) using the Regasm program and the following command:

```
REGASM "C:\Program Files\Omnis Software\<OS-version>\OMDOTNET.DLL"
```

If you have Admin rights you can run Regasm.exe from the Start>>Run command.

Regasm.exe is an application provided with the .NET Framework and you may be required to locate it, if it isn't on your PATH. It normally resides in a folder called Microsoft.NET\Framework\<Version Number> which is held in the operating system folder C:\Windows.

You should enclose path name to the OMDOTNET.DLL file in quotes, especially if the pathname has spaces in it. For example:

```
C:\Windows\Microsoft.NET\Framework\<version-no>\RegAsm.exe "C:\Program Files\Omnis Software\<OS-version>\OMDOT
```

Failure to have the Microsoft .NET Framework installed or failing to register OMDOTNET.DLL will result in an error, which will be reported in the Omnis trace log. You can open the Trace log from the Omnis Tools menu – if a component fails to load, it will be reported in the Omnis trace log.

Deployment

If you wish to deploy the Omnis .NET Objects component as part of your application, you must install both DLLs and register the OMDOTNET.DLL file on any client machine, as described above.

The .NET Example Library

There is an example library showing how the *.NET Objects* component works. The example library, which monitors RSS feeds from news web sites, is available under the Samples option in the HUB when you first launch Omnis (in the Studio Browser, press F2). Code from the example library is used later in this section to show you how to use the component.

Creating .NET Objects

Creating .NET Objects is a two-stage process that is similar to the process of creating Automation Objects or Java Objects in Omnis. A .NET Object is first declared and then constructed using the $createobject() method. This process is described in the following sections.

Defining .NET Objects

.NET Objects can be manipulated in a similar manner to other Omnis Objects. A .NET Object can be defined in a variety of ways:

- Declaring an Omnis Object variable in the Method Editor and setting the object subtype to a .NET Object class.
- Declaring an Omnis Object Reference variable and initializing it to point to a .NET Object class using notation.
- Declaring an Object variable in the Method Editor that does not have a subtype and initializing it to a .NET Object class using notation.

Defining .NET Objects via the Method Editor

To create a .NET Object in the Omnis Method Editor, simply declare an Object variable (e.g. an instance variable) and set its subtype to the desired .NET Objects Class. To do this, click the Subtype dropdown menu to open the “Select Object dialog, navigate the Object Selection Tree, highlight the class your require and click OK. The Object Selection Tree is split into one or more sections under the “NET” heading. These are “mscorlib” (basic .NET objects) and any classes that you have manually loaded (using $addclass).

The following shows a .NET Object being defined from a .NET Class which resides in the SOAP module which has been loaded manually.

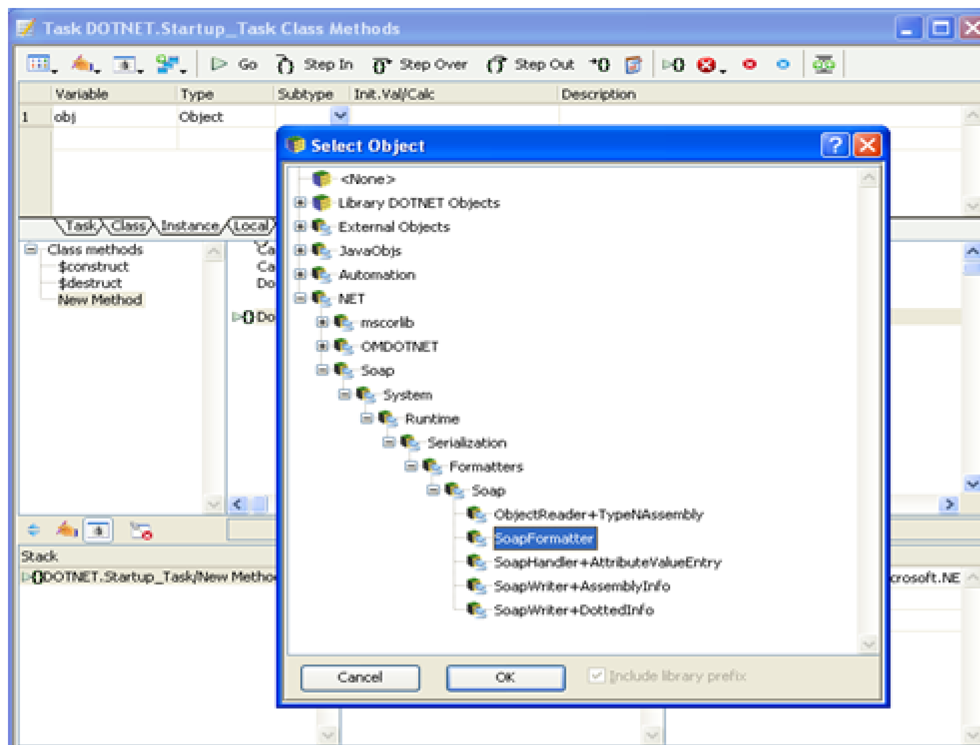


Figure 13:

Defining .NET Objects using Object References

Object Reference variables are a relatively new feature, introduced in Omnis Studio 4.0. Object Reference variables are similar to Object variables, but you are responsible for the allocation and de-allocation of the object concerned.

Object References are declared in the Method Editor, but do not have a subtype. To declare an Object Reference, simply declare a variable in the Method Editor and set its type to “Object Reference”. You can set a subtype in order to view the object’s methods in the Interface Manager, but the object associated with the reference has to be allocated using the $newref() method.

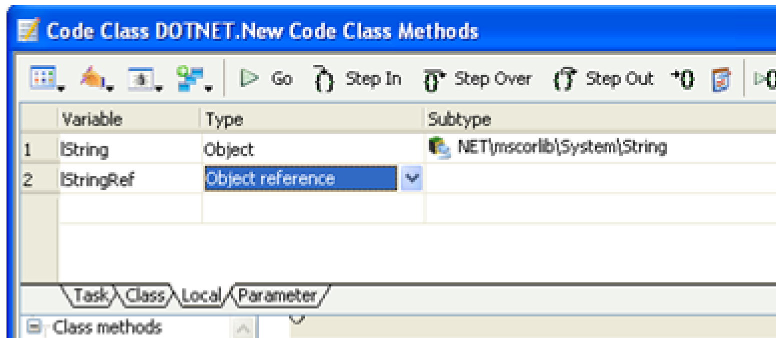


Figure 14:

The following shows both the *lString* object variable and the *lStringRef* Object Reference.

To allocate an Object to an Object Reference, you must use the notation. As notation strings for .NET Objects can be very long (due to the large hierarchies that are present in some .NET packages) the Notation Inspector can be displayed as a helper window when entering the notation for .NET Objects. You need to navigate the *$extobjects.DNet.$objects* group to find the .NET class you wish to reference. You can drag and drop the notation from the Notation Inspector to the Method Editor, and the full notation is created for you, such as the following:

Omnis will enclose the object string in double forward slashes since it does not recognize it as a variable or method name All .NET Object notation begins with *$root.$extobjects.DNet.$objects* (although you can omit $root and $extobjects), and the remainder of the notation string is dependant on the .NET class being used. The complete notation for a String object is as follows:

```
$root.$extobjects.DNet.$objects.//NET\mscorlib\System\String//
```

Now you have to add a call to the $newref() method manually. This method is responsible for allocating an Object to the Object Reference. Therefore, the complete method to allocate the String object would be as follows:

```
Calculate lStringRef as DNet.$objects.//NET\mscorlib\System\String//.$newref()
```

When the above method is executed, a String .NET Object will be created and assigned to the Object Reference *lStringRef*. It is your responsibility to ensure that this Object is deleted using the $deleteref() method once it has served its purpose. To de-allocate the variable, you can use:

```
Do lStringRef.$deleteref()
```

For further information about Object References, refer to *Omnis Programming* manual.

Defining .NET Objects via Notation

.NET Objects can be defined using the Object type via notation. To define a .NET Object via notation, simply follow the instructions in the previous section but use an *Object* type variable instead of an *Object Reference* type and call the *$new()* method instead of the *$newref* method.

Constructing .NET Objects

Once a .NET Object has been declared, it needs to be constructed. Declaring a .NET Object simply tells Omnis that the Object exists. Constructing a .NET Object using the $createobject() method actually performs the work of constructing the Object inside the .NET Virtual Machine.

By default all .NET Objects will have at least one $createobject() method which takes no parameters. This method can be used to create a .NET Object as follows:

```
Do lStringRef.$createobject()
```

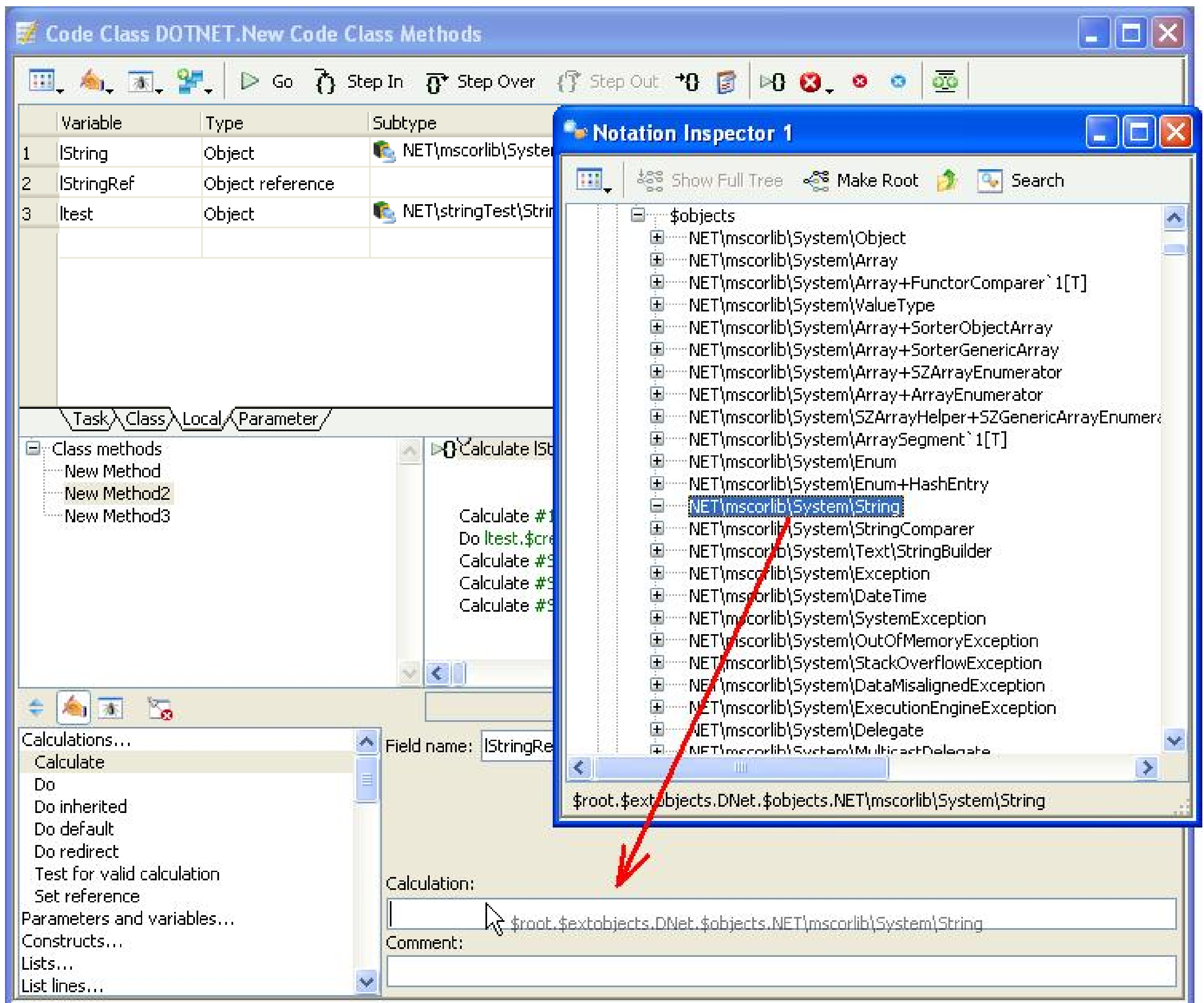


Figure 15:

Other .NET Objects may have more than one $createobject() method. In this case, each $createobject() method corresponds directly to a .NET Constructor for the class that is being used. For example, the .NET.System.String class has the following constructors:

String(char* value)
String(char[] value)
String(sbyte* bytes)
String(char c,int count)
String(char* value,int startindex,int length)
String(char[] value,int startindex,int length)
String(sbyte* value,int startindex,int length)
String(sbyte* value,int startindex,int length, int Encoding)

These are mirrored in Omnis by the following $createobject() methods:

Method	Parameters
$createobject	cValue, iStartIndex, iLength
$createobject	cValue
$createobject	cValue, iStartIndex, iLength
$createobject	cValue
$createobject	iValue, iStartIndex, iLength, oEnc
$createobject	iValue, iStartIndex, iLength
$createobject	iValue
$createobject	cC, iCount
$endswith	cValue, bIgnoreCase, oCulture
$endswith	cValue, iComparisonType
$endswith	cValue
$equals	oObj
$equals	cValue
$equals	cValue, iComparisonType
$equals	cA, cB
$equals	cA, cB, iComparisonType
$format	cFormat, oArgs
$format	cFormat, oArg0, oArg1, oArg2

Figure 16:

Calling any of above $createobject() methods will invoke the corresponding Constructor in .NET and create the .NET Object in the Virtual Machine. For example, calling:

```
Do mystring1.$createobject(mystring)
```

will invoke the String(char[] value).

Subclassing .NET Objects

.NET Objects can be subclassed by creating an Object class (using the "New Empty Class" option in the Studio Browser followed by the "Object" option) and setting its "Super Class" property ($superclass) via the Property Manager and the Object Selection Tree. For example, you can create an Object class and set its $superclass property to ".DNet.NET\mscorlib\System\String" via the Select Object dialog, as follows:

In this case, the *oString* object class becomes a subclassed object of *String* and inherits all of its methods as follows:

Subclassed .NET Objects can be used to define Object and Object Reference variables in the same manner as .NET Objects. Defining .NET Object variables is described in the previous section.

Using .NET Objects

The following sections describe how Omnis Data Types are converted to .NET Data Types, how you can pass parameters from Omnis to .NET, how you handle Return Values, and how you can retrieve data from .NET Objects. Other topics, such as Error Handling and Cache Control, are discussed later.

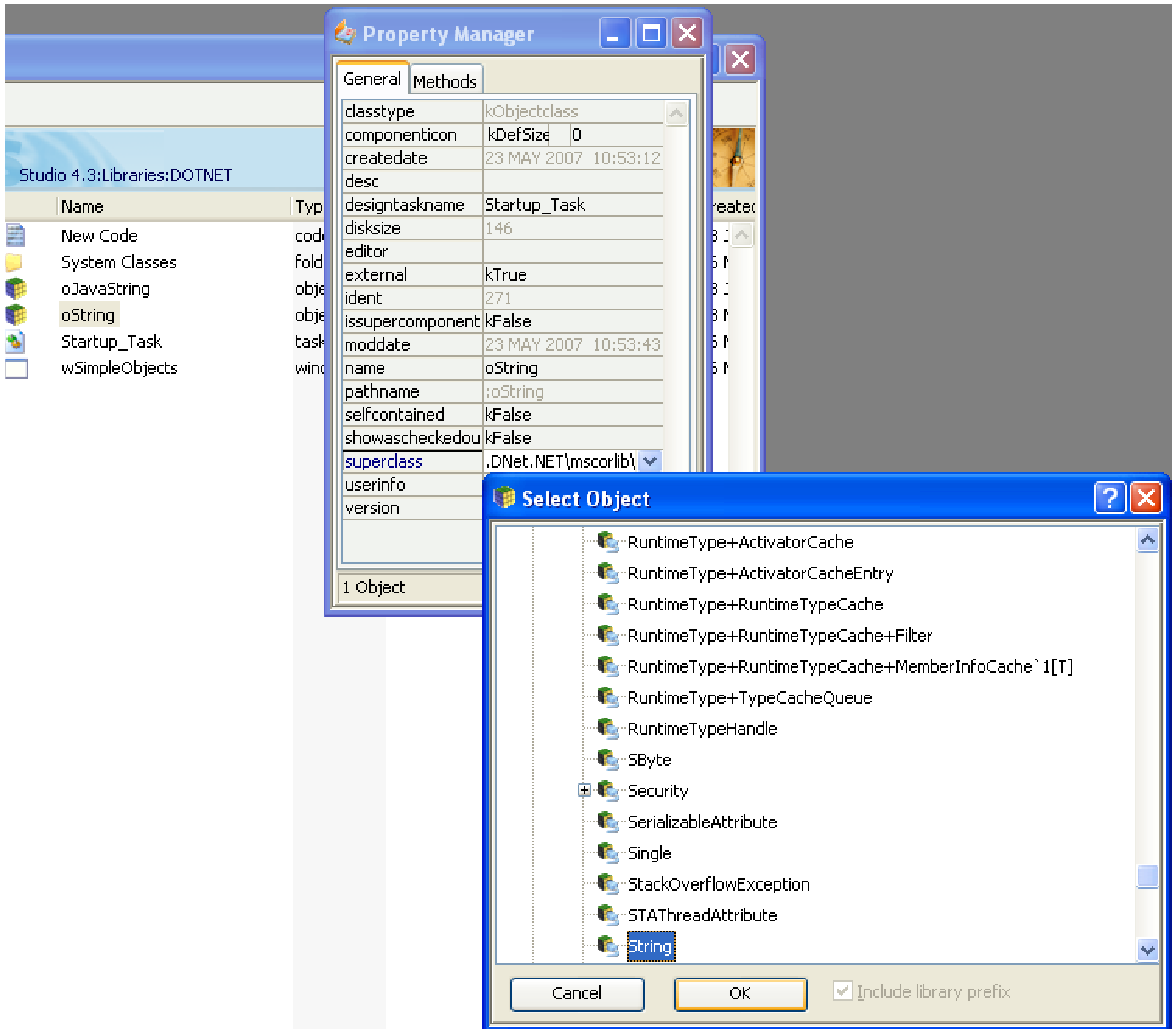


Figure 17:

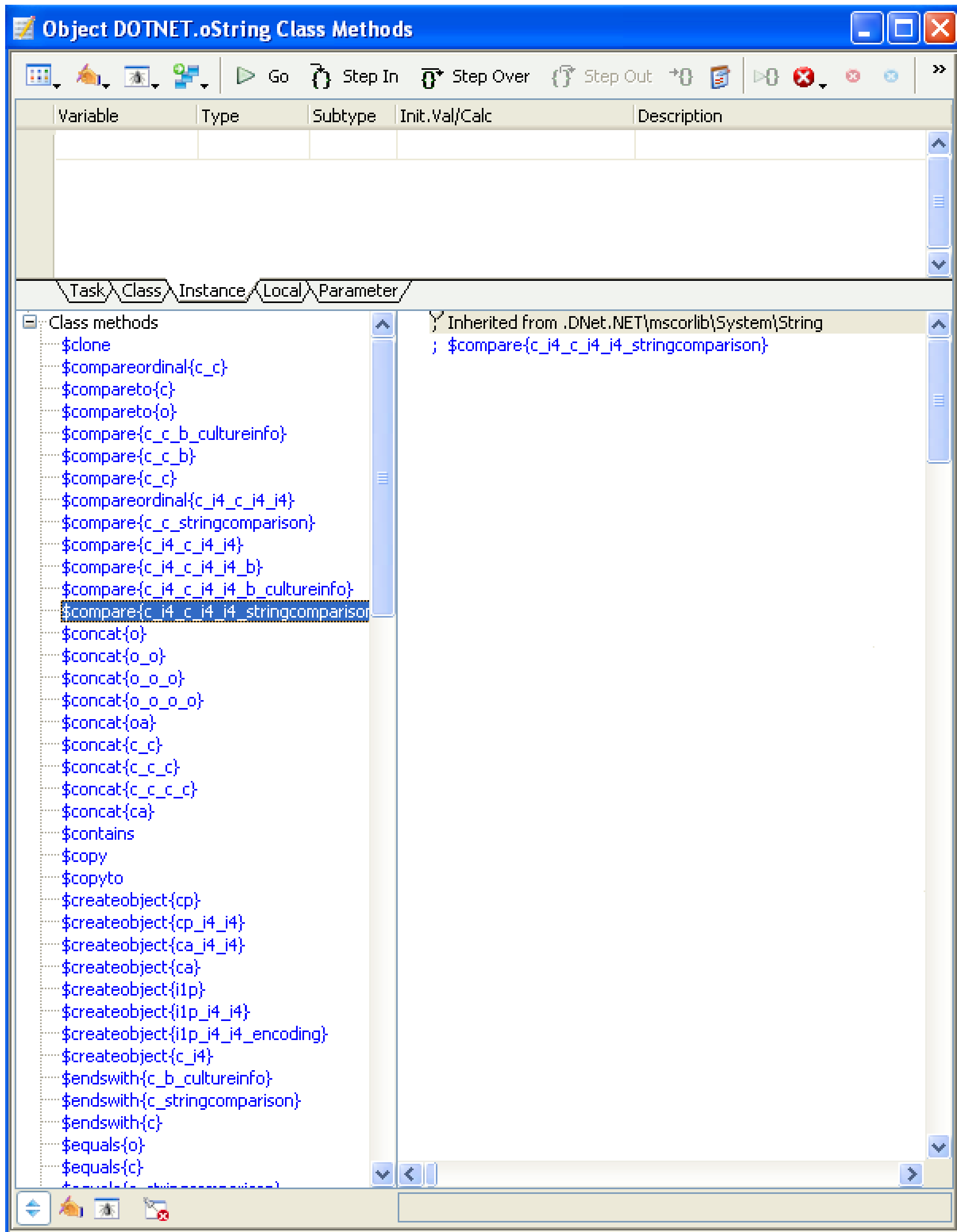


Figure 18:

Please check the Tech Notes on the Omnis web site (www.omnis.net) for supplementary information about using the .NET Objects component with complex data types.

Parameter Data Types

The following table shows how Omnis Data Types are converted to .NET Data Types when used as parameters to .NET Object methods. The Omnis Data Types marked with a '*' are known as *Overloaded Types* as they are compatible with more than one .NET Data Type.

Omnis Type	.NET Type
Simple types	
Character	char or char[]
Binary	byte or byte[]
Number Long integer	int, long or byte
Number Short integer	short
Number floating dp or Number <n> dp	float or double
Boolean	boolean
Complex types	
List	.NET Array
Object	.NET Object
Object Reference	.NET Object

As you can see from the above, most of the Data Types in Omnis can be coerced into multiple .NET Data Types. Although this coercion happens automatically, it is useful to be aware of *Overloaded Types* as they provide you with greater flexibility when programming with *.NET Objects*. For example, if you wish to construct a .NET.System.Byte object, you could define a Binary variable, load the byte into the variable and then call $createobject() with this variable. However, it is much quicker to simply call $createobject() using the integer value of the byte. For example:

```
Do mybyte.$createobject(65)
```

will create a .NET Byte object whose value is 65.

For a further explanation of *Overloaded Types*, see the Appendix.

Passing Parameters

In order to call .NET Objects effectively, it is necessary to know how Omnis Data Types are mapped to .NET Data Types when passing parameters. .NET Object Parameters can be categorized into two basic types:

- **Simple Parameters**
  Simple parameters are parameter types that are compatible with .NET base types
- **Complex Parameters**
  All other parameter types are considered to be Complex

Simple Parameter Types

- **Character**
  converts to .NET base type char/char[]
- **Binary**
  converts to .NET base type byte/byte[]
- **Number Long integer**
  converts to .NET base type int/long/byte
- **Number Short integer**
  converts to .NET base type short

- **Number floating dp or Number** <n> dp
  converts to .NET base type float/double
- **Boolean**
  converts to .NET base type boolean

Complex Parameter Types

- **List**
  can be converted to an array of any .NET type depending on list content
- **Object**
  can be converted to an Object of any .NET type depending on content
- **Object Reference**
  can be converted to an Object of any .NET Type depending on the content of the referenced Object

Passing Simple Parameter Types

Passing simple parameter types is relatively easy. You can call the .NET Object method with parameter(s), as you would call any method in Omnis. In the following example, the simple parameter type lpos is passed to the .NET.System.String charAt() function. This function returns the character value 'l' as this character resides at index 3 of the .NET String.

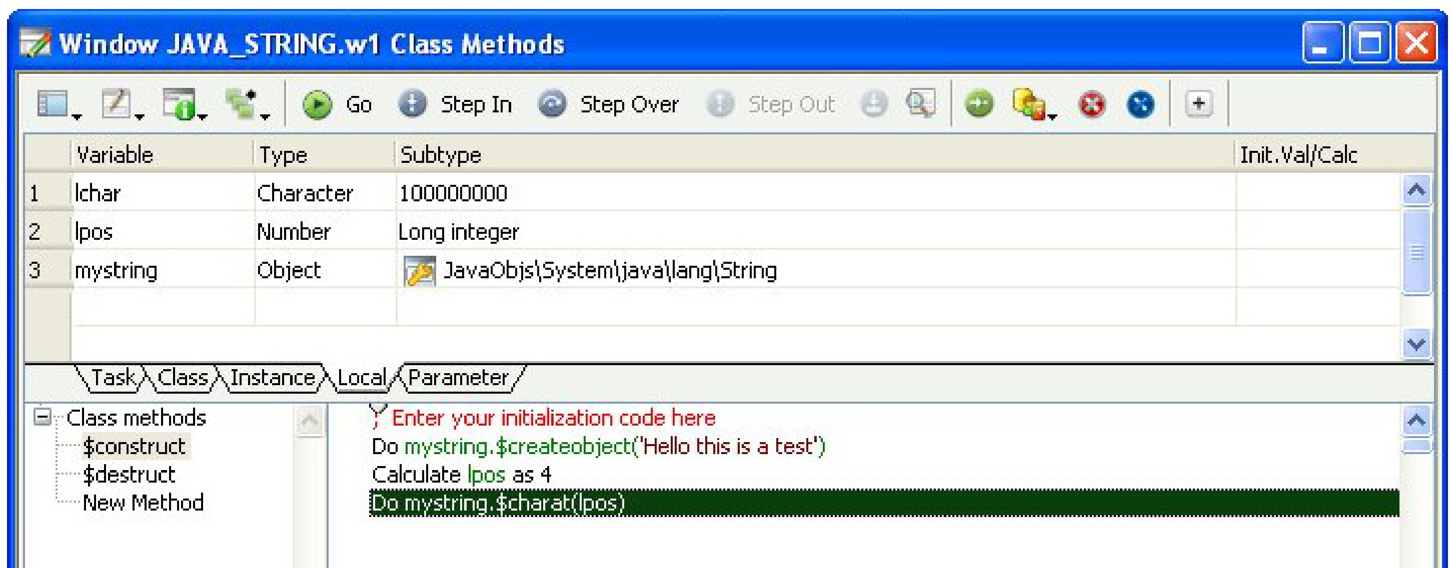


Figure 19:

Passing Complex Parameter Types

Complex Parameter types such as lists and Objects can be passed to .NET, however these Parameter types must be initialised before being used.

List Types

Omnis lists must contain valid data before being passed to a .NET Object. Note that if you use a list that has more than one column, only the first column will be used. Omnis lists which do not contain valid data will cause *.NET Objects* to return an error. For a further explanation of error handling, see the *Error Handling* section.

Omnis converts Single Column Lists to .NET Arrays based on the type of the data that is stored in the list. This conversion can be described by the following table:

Omnis List	.NET Array Type
Omnis List containing Number Long integer Types	int[] or long[]
Omnis List containing Number floating dp Types	float[] or double[]
Omnis List containing Number Short integer Types	short[]
Omnis List containing Boolean Types	boolean[]
Omnis List containing Object References	Array of .NET Objects

As you can see from the table above, *Overloaded Types* are also supported by Omnis Lists. For an explanation of *Overloaded Types*, see the Appendix. It should also be noted that char[] and byte[] are absent from the above table. This is because .NET char[] and byte[] types are created directly from Omnis Character and Binary variables.

Omnis will always attempt to determine the type of Array that you are trying to pass as a parameter by examining the data in the List. If a list contains .NET Objects, Omnis will attempt to determine the type of the objects in the list. If the type of all the objects is found to be the same, it is assumed that you are passing a parameter of that type. For example, if you create an Omnis list of .NET.System.String Objects, Omnis will assume that you are passing a String Array Parameter to .NET. However, if any of the Objects are found to be of different types, Omnis will assume that you are passing a generic Object Array (.NET.System.Object) to .NET.

When calling .NET methods that take arrays as parameters, you should make sure that the Omnis list that you pass to the method in question contains appropriate data. Lists which do not contain appropriate data will cause the function call to fail with an error (see the *Error Handling* section). For example, if you create an Omnis List with 12 .NET.System.String Objects and add an extra .NET.System.Byte object by mistake, your method call will fail if you are attempting to call a .NET function that accepts a .NET.System.String Array as a parameter.

Important Note: *When creating a list of .NET Objects, Object References should be used. This is because standard objects cannot be used in Omnis Lists. Once you have finished using your list of Object References, remember to delete each reference in the list by calling $deleteref for each reference. Please refer to the Omnis Programming manual for further information about Object References.*

Object Types

.NET Objects should be "created" using $createobject() before being passed as parameters to other .NET Objects. Passing an uninitialized object to a .NET function will return an error.

Returning Values From .NET Methods

The following table shows how .NET Return types are converted to Omnis data types.

.NET Type	Omnis Type
char[] or char	Character
byte[] or byte	Binary
int or long	Number Long integer
float or double	Number Floating dp or Number dp
short	Number Short Integer
Boolean	Boolean
.NET Arrays (except char[] and byte[])	List
Objects	Object References

As you can see from the table above, this is merely a reversal of the Omnis to .NET table shown in the previous section.

To return a value from a .NET object method, use the Method Editor to specify the variable that will be used to hold the return value. You must ensure that the Omnis variable that you are using is of the correct type to accept the value that will be returned from .NET. If this is not the case, an error will occur and the hash variables #ERRCODE and #ERRTEXT will be set accordingly (see the *Error Handling* section later in this manual).

Important: *If a .NET object returns a base object type, such as String, this will be returned as a base Omnis type. For example, the string object has a method called $substring which returns a character variable and not an object string. This is due to the fact that it would be otherwise impossible to get the contents of a string object if only a string object was returned.*

Obtaining constant (or Enum) values

Due to the numerous .NET classes available, constants (enums) are not added to the Omnis constants as shown in the Catalog (F9). To obtain the value of a particular constant, you can call the $getenum function with the *fully* qualified .NET class. For example, to open an existing stream object you can call:

```
Do oStream.$createobject( myFile,DNet.$getenum("System.IO.FileMode.Open"))
```

Adding .NET Classes

Additional .NET classes can be added by calling DNET.$addclass() with the location of the filename. For example, to add SOAP classes you can execute:

```
Calculate #S1 as con(DNet.$basefolder(),sys(9), "System.Runtime.Serialization.Formatters.Soap.dll")
Do DNet.$addclass(#S1) Returns #1
```

The first line of the method obtains the installation folder of the .NET framework and appends the soap filename. The second line adds the class (or classes if the string has more than one filename separated by commas) to the Omnis .NET classes which are available; the value returned is the number of files added, in this case 1. The .NET objects in the class are then available in the Select Object dialog when creating object variables.

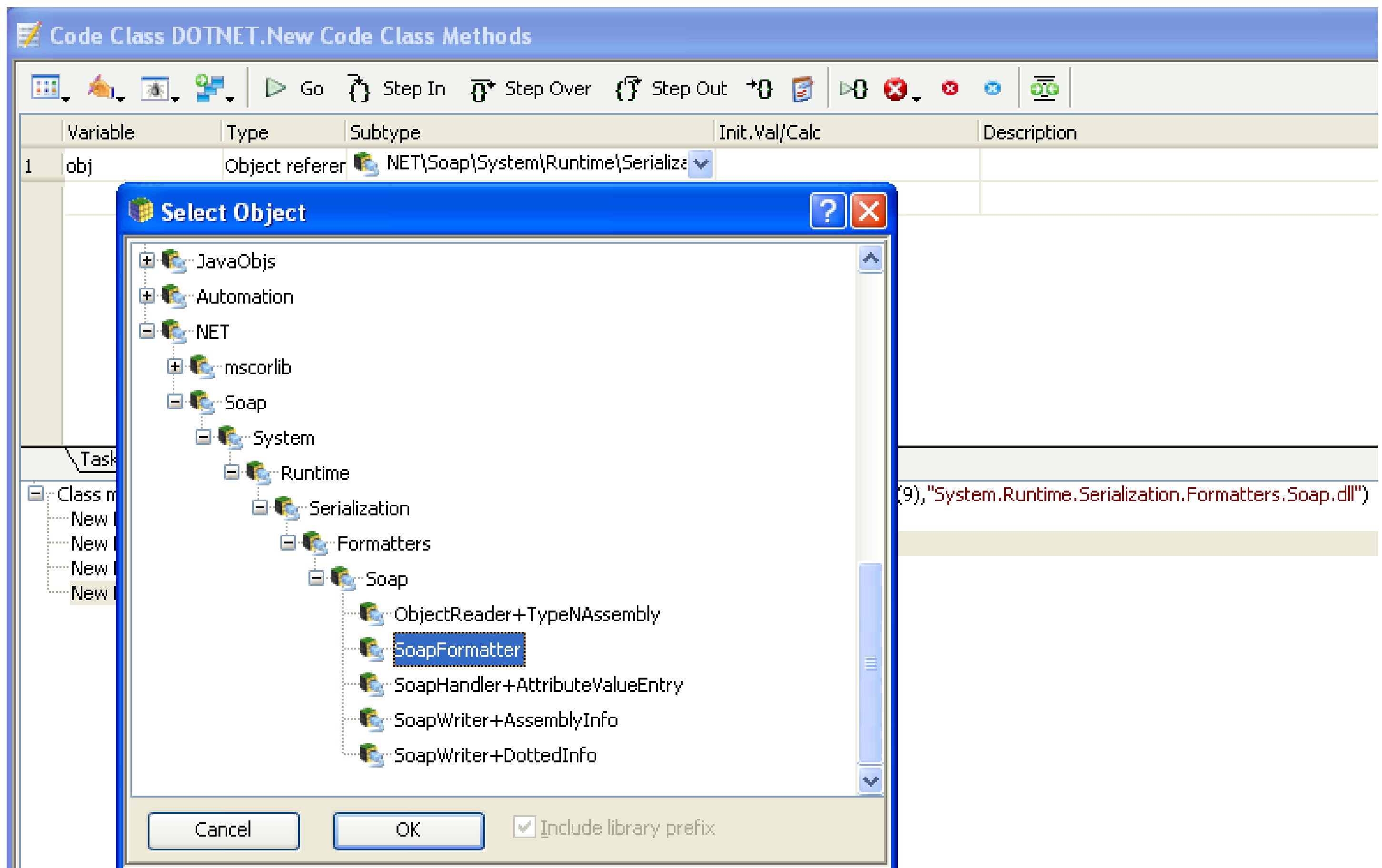


Figure 20:

It's not necessary to restart Omnis after adding classes, but remember that this information is not persistent between sessions, so you will need to add the classes into Omnis each time the application starts up or a window is opened.

Error Handling

.NET Objects makes full use of the #ERRCODE and #ERRTEXT variables in Omnis. If an error occurs while executing a .NET object method, these variables will report the error code and text on return from the method. #ERRCODE will produce the value 0 after a successful .NET method call.

.NET Objects example library

There is an example library showing how the *.NET Objects* component works. The example library, which monitors RSS feeds from news web sites, is available under the Examples link in the Welcome window when you first launch Omnis – you can open this window by clicking on the New Users button in the main Omnis toolbar.

Figure 21:

The example library has a number of pre-defined feeds from world-wide news organizations. After selecting a feed from the droplist, double-click a story headline to view the story detail. If you have Microsoft Internet Explorer ActiveX installed, this will be shown within an Omnis window, otherwise your default browser is used.

You can examine the code in the library to see how the *.NET Objects* component can be used in Omnis applications to access .NET functionality.

The RSS feed component

The RSS feed application comprises an Omnis library and a DLL which are located in the \welcome\examples folder under the main Omnis folder. The rssreader.dll contains the compiled C# code and provides Omnis with a single .NET object with three properties and two methods.

The first method in the RSS feed object, getdocument(), retrieves the XML source from the specified address in the rssUrl property. Thereafter it parses out the XML in the 'channel' node and sets it to the channelNode property. This is of type System.Xml.XmlNode and allows Omnis to call its member functions to determine title, language, and so on, of the RSS feed. Lastly it retrieves a list of all the 'item' or story XML nodes within the channel node itself. The property numberOfTags is derived from the number of 'item' nodes within the XML source.

The second method, getItemTags(), allows Omnis access to the nth story XmlNode determined by the function argument. Once again, calls to member functions can further extract XML from the XmlNode object and provide information such as Story Headline, Feed Description and Publish Date.

The RSS feed library

To discover how Omnis uses the .NET object you should examine the code in the RSS feed library. The Startup_Task in the library opens the RSS window and creates the object. The code for the $construct() method of the RSS window is as follows. The first part of the method loads the rssreader.dll and adds its .NET classes to Omnis using the $addclass() method.

```
Do FileOps.$splitpathname(sys(10),lDrive,lPath,,)
Calculate lPath as con(lDrive,lPath,'rssreader.dll')
Do DNet.$addclass(lPath)
```

Note the DNet object is part of the $root.$extobjects group. The next part of the method loads the System.Xml library and adds this to Omnis. You can use the $basefolder() method to return the folder containing your .NET Framework:

```
Do DNet.$basefolder Returns lPath
# lPath is something like C:\WINNT\Microsoft.NET\Framework\<version>
Calculate lPath as con(lPath,'\System.Xml.dll')
Do DNet.$addclass(lPath) Returns #F
```

The next part sets up and builds a list containing the list of RSS feeds and loads the first one in the list:

```
Do iItemList.$define(iItemTitle,iItemDes,iItemLink,iItemPubDate)
Do method $buildurllist
Do iUrlList.$first()
Do iUrlList.$loadcols()
```

The next part of the window $construct() method creates an instance of the getRssFeed object and constructs the object in .NET using the $createobject() method. Finally, a class method is called to retrieve the feed.

```
Do $root.$extobjects.DNet.$objects.//NET\rssreader\rssreader\getRssFeed//.$newref() Returns iFeedObject
Do iFeedObject.$createobject()
Do method $getfeed
```

You can use the Interface Manager to examine the properties and methods of a .NET object. For example, this is the iFeedObject object created in the RSS feed window.

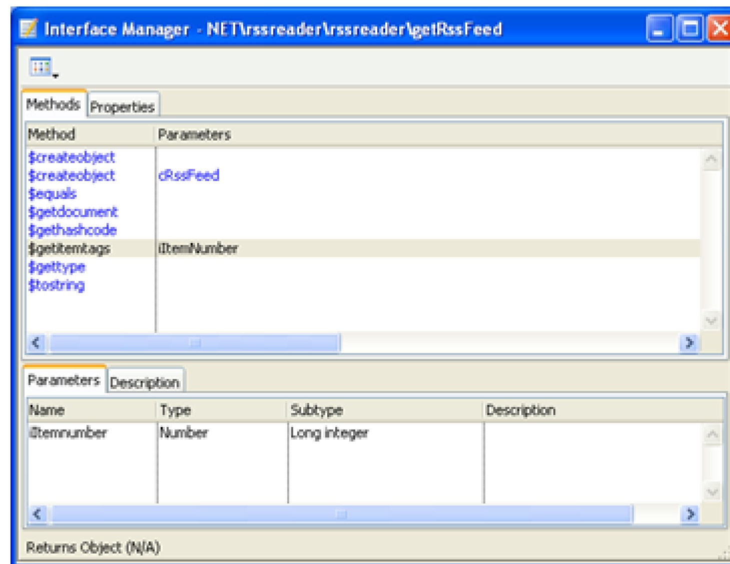


Figure 22:

The $buildurllist and $getfeed methods are contained in the window class and are called from the window $construct() method. The $buildurllist method builds a list containing a single column of RSS feed URLs which in this case are loaded from a static list of URLs, but could have been constructed from a database.

```
Do iUrlList.$define(iFeedLink)
Do iUrlList.$add('http://feeds.foxnews.com/foxnews/latest?format=xml')
# and so on...
```

The $getfeed method retrieves the XML source file for the current RSS feed and extracts the relevant information to display in the window.

```
# $getfeed method
Do iItemList.$clear()
Calculate iFeedObject.$rssurl as iFeedLink
Working message Waiting/-1073735814,-1073735810;50;0;60 {Retreiving RSS Feed} ## message while object is retri
Do iFeedObject.$getdocument() ## the XML source is retrieved
# Get the channel XmlNode and extract the channel information
Calculate lXmlNodeObject as iFeedObject.$channelnode
Do method $getchannelinfo (lXmlNodeObject)
For lItemNumber from 1 to iFeedObject.$numberofitemtags step 1
 Do iFeedObject.$getitemtags(lItemNumber) Returns lXmlNodeObject
 Do method $getiteminfo (lXmlNodeObject)
End For
```

The $getchannelinfo method retrieves the title, language, and description for the RSS document while the $getitemtags method returns the information about each news story listed in the RSS feed including the URL, title, description, and publication date.

```
# Local var: lSubNode (Object) subtype NET\Xml\System\Xml\XmlNode
Do pRssItemNode.$selectsinglenode("link") Returns lSubNode
Calculate iItemLink as lSubNode.$innertext
Do pRssItemNode.$selectsinglenode("title") Returns lSubNode
Calculate iItemTitle as lSubNode.$innertext
Do pRssItemNode.$selectsinglenode("description") Returns lSubNode
Calculate iItemDes as lSubNode.$innertext
# ignore all html tags
If pos('<',iItemDes)<>0
 Calculate iItemDes as left(iItemDes,pos('<',iItemDes)-1)
End If
Do pRssItemNode.$selectsinglenode("pubDate") Returns lSubNode
Calculate iItemPubDate as lSubNode.$innertext
Do iItemList.$add(iItemTitle,iItemDes,iItemLink,iItemPubDate)
Calculate iItemPubDate as ''
Calculate iItemDes as ''
```

When you double-click on a news story in the RSS feed window, Omnis uses the URL stored in iItemLink (loaded from the iItemList) to launch a browser window and display the news story.

Method Method Overloading and Pattern Matching

Although Omnis Studio does not normally support the overloading of method names, the *.NET Objects* component is an exception as method overloading is usually unavoidable when dealing with even the most basic of .NET classes. The following sections discuss how Omnis deals with overloaded methods and how it is possible to manually call an overloaded method directly from Omnis.

Pattern Matching

Omnis uses pattern matching in order to determine the overloaded method that you're trying to call. In simple terms, Omnis examines the parameters that you pass to a method and attempts to call the method whose parameters most closely match the parameters that you are passing. For example, consider the following .NET methods:

```
myfunc(int p1)
myfunc(float p1)
```

Calling myfunc from Omnis using an Omnis Number type set to Long Integer will call myfunc(int p1). Calling myfunc from Omnis using an Omnis Number type set to Floating dp will call myfunc(float p2).

Overloaded Data Types

An extra level of complexity is added to Pattern Matching with the support of Overloaded Data Types. An Overloaded Data Type is an Omnis data type that has more than one matching data type in .NET. Overloaded Data Types are split into three groups, described in the following sections.

Number Long Integer types

The Omnis Number Long Integer type can be translated to the following .NET types:

- int
- long
- byte

If a Number Long Integer type is passed to an overloaded method, the following occurs:

- Omnis attempts to find an overloaded method that has the same number of "int" parameters that you are passing. Therefore, if you are calling a method with (int,boolean,int), Omnis will try to find an overloaded method which matches this Parameter Search Pattern.
- If a match is not found, Omnis searches for a method that has a matching number of "long" parameters, that is, Omnis will modify the Parameter Search Pattern in an attempt to find a match using combinations of int and long types. Thus, if (int,boolean,int) is not found, Omnis will look for the following:

  (int,boolean,long)
  (long,boolean,int)
  (long,boolean,long)

- Finally, if this fails, Omnis searches for a method that has a matching number of "byte" parameters, that is, Omnis will modify the Parameter Search Pattern in an attempt to find a match using combinations of int and byte types. Thus, if '2' does not find a compatible method, Omnis will look for the following:

  (byte,boolean,int)
  (int,boolean,byte)
  (byte,boolean,byte)

Note that any method that matches the "int" Parameter Search pattern will always be called if it is available. For example, consider the following .NET methods:

```
myfunc(int p1)
myfunc(long p1)
myfunc(byte p1)
myfunc(boolean p1)
```

calling myfunc using a Long Integer type will always call myfunc(int p1) in .NET. If this method is removed:

```
myfunc(long p1)
myfunc(byte p1)
myfunc(boolean p1)
```

calling myfunc using a Long Integer type will always call myfunc(long p1), and so on.

Number Floating dp types

The Omnis Number Floating dp type can be translated to the following .NET types:

- double
- float
- decimal

If a Number Floating dp Omnis type is passed to an overloaded method, the following occurs:

- Omnis attempts to find an overloaded method that has the same number of "double" parameters that you are passing. Therefore, if you are calling a method with (double,boolean,double), Omnis will try to find an overloaded method which matches this Parameter Search Pattern.
- If a match is not found, Omnis searches for a method that has a matching number of "float" parameters, that is, Omnis will modify the Parameter Search Pattern in an attempt to find a match using combinations of double and float types. Thus, if (double,boolean,double) is not found, Omnis will look for the following:

  (double,boolean,float)
  (float,boolean,double)
  (float,boolean,float)

- Finally, if this fails, Omnis searches for a method that has a matching number of "decimal" parameters, that is, Omnis will modify the Parameter Search Pattern in an attempt to find a match using combinations of double and decimal types. Thus, if '2' is not found, Omnis will look for the following:

  (double,boolean,decimal)
  (decimal,boolean,double)
  (decimal,boolean,decimal)

Note that any method that matches the "double" Parameter Search pattern will always be called if it is available. For example, consider the following .NET methods:

```
myfunc(double p1)
myfunc(float p1)
myfunc(decimal p1)
myfunc(boolean p1)
```

calling myfunc using a Number Floating dp type will always call myfunc(double p1) in .NET. If this method is removed:

```
myfunc(float p1)
myfunc(decimal p1)
myfunc(boolean p1)
```

calling myfunc using a Floating dp type will always call myfunc(float p1), and so on.

.NET Object types

The Omnis .NET Object type can be translated to the following .NET Types:

- < The object that you are attempting to pass as a parameter, e.g. .NET.System.String >
- .NET.System.Object

When matching objects, Omnis will always try to find a method which takes an object parameter that matches the type of the object that you are passing. So if you pass a .NET.System.String object to an overloaded method, Omnis will look for a match using .NET.System.String as a search pattern. If Omnis fails to find a matching method, then Omnis will call a method which takes the generic type .NET.System.Object as a parameter, if such a method is available. Therefore, if Omnis searches for a method using the Parameter Search Pattern (String,boolean,String) and a matching method is not found, Omnis will look for (Object,boolean,String) followed by (String,boolean,Object) followed by (Object,boolean,Object).

Note that any .NET method that has a parameter list which matches the Objects that you are using as parameters will always be called if it is available. For example, consider the following .NET methods:

```
myfunc(String p1)
myfunc(Object p1)
myfunc(boolean p1)
```

calling myfunc using an Omnis object whose subtype is set to ".NETObjs\System\.NET\lang\String" will always call myfunc(String p1) in .NET. If this function is removed:

```
myfunc(Object p1)
myfunc(boolean p1)
```

calling myfunc using an Omnis object whose subtype is set to ".NETObjs\System\.NET\lang\String" will always call myfunc(Object p1).

Char and Byte Pattern Matching

The following table shows how Omnis converts Character and Binary data types to .NET data types when dealing with overloaded methods.

Omnis	.NET
character types (length > 1)	char[]
binary types (length > 1)	byte[]
character types (length <= 1)	char
binary types (length <= 1)	byte

As a result of these conversions, you should be aware that it is not possible for Omnis to call an overloaded .NET function that takes char[] as a parameter using a single character. This is demonstrated in the following example. Consider the following:

```
myfunc(char p1[])
myfunc(char p1)
```

Following the rules for data conversion, calling myfunc with a single character will call myfunc(char p1) while calling myfunc with a string of characters will call myfunc(char p1[]). Thus, in this situation it is not possible to call myfunc(char p1[]) with a single character as myfunc(char p1) will always be called if the length of the data that you are passing is 1. However, it is possible to force Omnis to call myfunc(p1[]) with a single character if you call the overloaded method directly. The procedure for doing this is discussed in the next section.

Note: If the length of an Omnis Character type is 0, it will be converted into a .NET char type whose value is 0. If the length of an Omnis Binary type is 0, it will be converted into a .NET byte type whose value is 0.

Calling Overloaded Methods Directly

In some rare cases, it may be necessary to call overloaded methods directly. Consider the following example: you may be trying to call overloaded functions in the .NET.System.String Class. The functions are defined in .NET as follows:

```
compareto(Character c)
compareto(Object o)
```

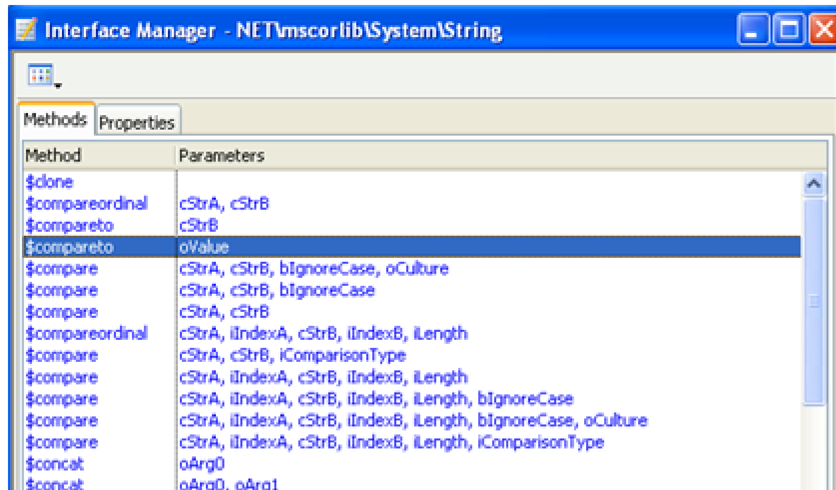


Figure 23:

If you want to call a particular method directly, you can do so from the Omnis Interface Manager, by selecting the desired overloaded method that you wish to call and clicking on the description tab. This will display the "real name" of the overloaded method.

The following screenshot shows the Omnis Interface Manager for the ".NET.System.String" class. If you were to call the $compareto() method using an Omnis "object" type, you must use the "real name" of this method which, in this case, is $valueof{o}.

Note: Each parameter is separated by an underscore (_).

The naming convention for these parameters are as follows:

<table>
<tr><th>.NET Type</th><th>Abbreviated to (Omnis type)</th></tr>
<tr><td>Char</td><td>“c”</td></tr>
<tr><td>boolean</td><td>“b”</td></tr>
<tr><td>Decimal</td><td>“d”</td></tr>
<tr><td>Float</td><td>“f4”</td></tr>
<tr><td>Double</td><td>“f8”</td></tr>
<tr><td>Unsigned Byte</td><td>“u1”</td></tr>
<tr><td>Signed Byte</td><td>“i1”</td></tr>
<tr><td>Unsigned Short</td><td>“u2”</td></tr>
<tr><td>Signed Short</td><td>“i2”</td></tr>
<tr><td>Unsigned Int</td><td>“u4”</td></tr>
<tr><td>Signed Int</td><td>“i4”</td></tr>
<tr><td>Unsigned Long</td><td>“u8”</td></tr>
<tr><td>Signed Long</td><td>“i8”</td></tr>
<tr><td>Object</td><td>“o”</td></tr>
<tr><td>Object of known type</td><td>“”</td></tr>
<tr><td>Enum</td><td>“ | | Void | “v” |</td></tr>
</table>

An additional 'p' character means pointer (or *) and an 'a' character means an array.

Nested Object Arrays

The *.NET Objects* component supports Nested Object Arrays as parameters to .NET methods, and it can also interpret Nested Object Arrays returned from .NET. A Nested Object Array is an array of type .NET.System.Object that is capable of storing any .NET Object or Array of .NET Objects.

To use a Nested Object Array, you must create an Omnis List variable and define it using a Binary variable. You can then add .NET Objects and lists of .NET Objects to the list. Once your list is complete, you can pass it to any .NET method that accepts .NET.System.Object array as a parameter.

Nested Object Arrays can also be returned by .NET methods. A Nested Object Array is returned to Omnis as a Binary list. This list may contain both Object References and embedded Lists.

Overloaded Types

The following sections provide detailed descriptions of the Overloaded Types used by the *.NET Objects* component.

Omnis Character types

The Omnis Character type can be converted into the following .NET types:

`Char`

`char[]`

This allows you to call any .NET object that takes either a char or char[] as a parameter with an Omnis Character variable. You may also place .NET return values that are of type char or char[] into an Omnis Character variable.

In the case of "char[]" parameters, the Omnis Character variable is automatically converted into a .NET char array (or vice versa if you are dealing with return values). This allows objects such as the .NET String to be created easily from Omnis. For example:

```
Calculate mychar as "hello this is a test"
Do mystring.$createobject(mychar)
```

The above will invoke the .NET String Constructor **String**(char[] value)

Note: If you attempt to call a .NET function that takes a single char as a parameter and the Omnis variable that you are passing contains more than one character, .NET will only receive the first character of the string.

Omnis Binary types

The Omnis Binary type can be converted into the following .NET types:

`Byte`

`byte[]`

This allows you to call any .NET object that takes either a byte or byte[] as a parameter with an Omnis Binary variable. You may also place .NET return values that are of type byte or byte[] into an Omnis Binary variable.

In the case of "byte[]" parameters, the Omnis Binary Type is automatically converted into a .NET byte array (or vice versa if you are dealing with return values). This allows objects such as the .NET String to be created easily from Omnis. For example:

```
Do mystring.$createobject(mybinval)
# where mybinval is a binary Omnis Variable
```

The above will invoke the .NET String Constructor **String**(byte[] bytes)

Note: If you attempt to call a .NET function that takes a single byte as a parameter and the Omnis variable that you are passing contains more than one byte, .NET will only receive the first byte of the data that you are passing.

Omnis Number Long Integer types

The Omnis Number Long Integer type can be converted into the following .NET types:

`Int`

`long`

`byte`

This allows you to pass Omnis Number Long integer variables to .NET functions which accept these types.

Note: When dealing with return values, .NET int and long types can be converted into the Omnis Number Long integer type. However, all byte return values are always converted to the Omnis Binary type.

Omnis Number Floating dp types

The Omnis Number Floating type can be converted into the following .NET types:

`Float`

`double`

This allows you to pass Omnis Number floating dp variables to .NET functions which accept these types.

Chapter 5—oXML

This chapter describes the Omnis XML object (oXML), an external component which allows you to parse and manipulate XML documents in Omnis using the standard Document Object Model (DOM) API. This chapter does not provide an exhaustive description of XML or the DOM, since this can be gained from many other sources.

For further information about XML, you can look up XML-related web sites, and read one or two of the many books available on the subject. Here are one or two resources we found very useful:

- **XML and DOM standards**
  www.w3.org has the official XML standards and a lot of good general information; also includes a full definition of the DOM API. A look at the DOM Level 2 definition is very useful, whereas studying the XML specification is not necessary.
- **Information and Tutorials**
  www.xml.com has background information on XML and news;
  www.w3schools.com/xml has some useful XML tutorials & information.
- **Omnis Tech Note:** you may also like to read the Omnis Technical Note TNXM0001: Creating XML documents with oXML which includes an example library to download.

About oXML

The interface to XML documents is implemented in Omnis Studio using the standard Document Object Model (DOM) API as an external component which must be instantiated via an Omnis *Object Variable*. The oXML component addresses the most basic XML requirement, namely the ability to parse and extract information from an XML document, and to generate new XML documents. The oXML component allows you to parse and manipulate XML documents using a standard set of methods provided by the DOM level 2 API, plus some additional methods that speed up the process of building a document. The oXML component also allows you to display an XML document in the tree list component, which is well suited to displaying the hierarchical structure contained in XML documents.

oXML Availability

oXML is available as a built-in component in most editions of Omnis Studio and is therefore installed and ready to use. In previous versions of Omnis Studio (pre Studio 6) the oXML component was available as a separate plug-in which had to be purchased and installed separately.

What is XML?

To use oXML to access your XML documents, you need a working knowledge of XML and the DOM. This section provides a short introduction to XML and the DOM. If you are already familiar with these technologies, and/or you have read one of the many sources of information about XML and DOM, then you may like to skip this section.

XML (eXtensible Markup Language) allows you to store, exchange and display data or information in a structured and efficient way. In this respect it is no different from most existing data formats, except that XML provides a higher degree of standardization and flexibility than many other proprietary technologies, opening up many new and exciting opportunities in business computing and information technology. XML has already revolutionized content management, information publishing, and news syndication, as well as other B2B markets, while the adoption of XML across many other industry sectors seems to be gathering pace.

XML allows you to store structured documents or data as text and provides you with a way of manipulating, transforming, and presenting your data in many different formats. For example, information or data stored in an XML document can be displayed in a web browser using a Cascading Style Sheet (CSS). In addition, when XML documents are stored in a database they can be queried and retrieved much like any other data source.

What are the Benefits of using XML?

The business benefits of using XML and XML-based systems are well documented in the IT industry and media. XML, or rather technologies that use XML as their basis, promise to provide the IT industry with greater standardization, interoperability, efficiency, and present the potential for many new technologies. If you are an application developer, you will no doubt be asked some time in the future to create applications that will "handle XML".

- **Platform Independent and Reusable**
  XML is machine and platform independent so it can be exchanged between one system or network and another. Plus, once information is in XML format it can be reused for many different purposes for digital and printed publication.
- **Worldwide Standard**
  XML is a standard language defined and ratified by the W3C consortium so it is not controlled or owned by any one company. This ensures the future of XML as an open standard employed by the whole IT industry.
- **Information exchange**
  Since XML is an agreed standard it affords a high degree of information exchange, in particular between networks, businesses and other interdependent organizations.
- **New Business opportunities**
  The standardization and flexibility of XML mean that many existing business problems can be solved more efficiently, while many new business opportunities will arise that take advantage of XML. For example, XML has already revolutionized Content management, publishing & news syndication, and will transform many other areas of business, particularly those suited to automation.

Elements

XML is very much like HTML, but it differs in one or two important ways. Like HTML, XML uses tags to define the "elements" (content or data holders) within a document, but unlike HTML, XML tags only describe the data or content, they do not contain any information about the display or formatting of the content or document as a whole. Each element must have a start and an end tag, and tag names are case-sensitive.

HTML conforms to a standard set of tags, whereas XML element names can be anything you like providing a better description of each piece of data or the content in your document. For example, to create a file to store the contents of a bookstore you can create an element called <bookstore> to contain the information about all the books. XML documents are often described as having "metadata" since the information in the tags describes the data within the tag itself. In this case, someone looking at the file containing the tag <bookstore> can see immediately that the information relates to a bookstore.

Elements within a document are often nested in a hierarchical structure, building a more detailed or structured picture of the thing or things being described in the document. Therefore, individual elements are referred to as "nodes". The top level element in a document is called the "root node", which has an ID of 0, and all elements inside it are called "child nodes" which have unique IDs identifying them. Carrying on the book example, the <bookstore> element or root node could contain elements for <book>, <title>, <author>, <publisher>, and <isbn>, plus you can further describe the <author> element using <firstname> and <lastname> child nodes. An XML document with these elements or nodes would have the following structure:

```
<?xml version="1.0"?>
<bookstore>
  <book>
    <title>Essential XML for Web Professionals</title>
    <author>
      <firstname>Dan</firstname>
      <lastname>Livingstone</lastname>
    </author>
    <publisher>Prentice Hall PTR</publisher>
    <isbn>0130662542</isbn>
    <price>34.99</price>
  </book>
  <book>
    <title>Professional XML (Programmer to Programmer)</title>
    <author>
      <firstname>Mark</firstname>
```

```
      <lastname>Birbeck</lastname>
    </author>
    <publisher>Wrox Press, Inc</publisher>
    <isbn>1861005059</isbn>
    <price>59.99</price>
  </book>
</bookstore>
```

Note the <firstname> and <lastname> elements are nested inside the <author> element, while all sub-nodes are contained in the <bookstore> root node. Also note the obligatory XML declaration at the beginning of the document which defines the XML version of the document; this is a processing instruction that gets sent to the XML parser.

Attributes

Like HTML, elements can have "attributes" (properties) that further describe the element, but again they do not provide any information about the display of the data. For example, the <book> element in our sample xml above could have the attribute "genre" which is written like this:

```
<book genre="Computing">
```

Note that genre is a general characteristic of a book and is therefore considered an attribute of a book (i.e. many books may be in the same genre), whereas the title of a book is unique to each book and is therefore described in an element as part of an individual book.

Entities

Entities let you represent a single character, a number of characters, or a string of words using a short alias name. There is a range of ISO approved entities that have reserved name and number codes to represent specific characters, such as & for ampersand, < for lesser than, > for greater than, " for double quotation mark, ' for apostrophe, and € for the Euro symbol, and so on. You can also define your own custom entities in your XML documents in the document template, either internally quoted in the DTD (Document Type Definition: see below) or they can be listed in an external file. For example, you could represent a publisher name by declaring <!ENTITY ph "Prentice Hall PTR">, therefore writing the publisher name as &ph; in the body of your document.

Entities that hold text, like those described above, are called *parsed entities*. You can also create entities for non-text data or files, such as image files, video, binary files, or even other applications, and these are called *unparsed entities*, but they are also referred to as *notations*.

DTDs

When a document has all the correct start and end tags and is properly nested, it is described as "well-formed". XML documents are processed through an XML parser which checks for correct syntax or "well-formedness". Documents can be further *validated* against a template or *Document Type Definition* (DTD), or a schema. A DTD is itself a text document which contains a description of the elements and entities for a particular type of XML document, in other words, it specifies what type of data or content the document can contain. The DTD would contain a list of elements allowed in the XML document, defining the name and data type for each element. The DTD for an XML document can be included inline, as part of the XML document itself, or it can be a separate file referenced in the XML document.

```
<!DOCTYPE BOOKS [
<!ENTITY PH "Prentice Hall PTR">
<!ELEMENT book (TITLE,AUTHOR,PUBLISHER,ISBN)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (FIRSTNAME,LASTNAME)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT isbn (#PCDATA)>
]>
```

Most of the time you will need to use an industry standard DTD, rather than creating your own. Using standard DTDs ensures the portability of your data or documents across many different applications and between organizations.

Schemas

Increasingly, DTDs are being superceded by *schemas*, or *w3c schemas* as they are sometimes called. oXML supports the use of schemas to validate XML documents. Like DTDs, schemas define the structure and types of data allowed in documents but they provide greater control over the structure and types of data in your XML documents. Schemas use XML *namespaces* to define the elements and attributes in your XML documents. Namespaces are unique names that identify element types and attribute names.

Schemas are themselves written in XML so you can create and manipulate them using oXML. Schemas are more powerful than DTDs since you can define the type and constraints on the data in your documents. Schemas can contain a number of built-in data types, such as, xs:string, xs:decimal, xs:date, xs:anyURI, and you can create your own custom types. Like DTDs, schemas can be referenced externally in your XML documents: see below.

The differences between DTDs and Schemas become apparent when you compare one with the other, for example, the following DTD called note.dtd describes the structure of a very simple XML document.

```
<!ELEMENT note (to, from, heading, body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

The DTD defines the 'note' root node as having four child nodes (to, from, heading, body). The DTD is placed in the XML document itself or referenced externally.

A simple schema called note.xsd can be used to define the same structure:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

The schema defines the 'note' *complex element* that contains four *simple elements* (to, from, heading, body).

Using Schema Files for Validation

The XML DOM Document object has some properties to allow you to specify and use an external schema file (XSD) for validation.

- $fullschemavalidation
  If true, and $parservalidates is also true, and the parser will validate against a schema, the parser performs additional checks against the schema.

You should set $fullschemavalidation to true unless performance is an issue.

The other new properties allow an external schema to be specified:

- $nonamespaceschemalocation
  if specified, this property becomes the noNamespaceSchemaLocation attribute for the document being parsed.
- $schemalocation
  if specified, this property becomes the schemaLocation attribute for the document being parsed.

The schemas specified in these properties need to be referenced by a pathname to the schema file.

For example, to use an external schema, turn on $fullvalidation (without this, the absence of the schema file is an unreported and ignored warning), and set $schemalocation to:

```
"urn:books c:\dev\studio60orfc\oxml\test\books.xsd"
```

where the second component is the path to the schema file on your system.

If an XSD is in the same directory as the XML, you can use:

```
"urn:books books.xsd"
```

XML Parser

An XML parser or processor is a software module that checks your documents for “well-formedness” and performs validation against a DTD. The XML parser provided with oXML, called Xerces (called ‘xerces-c_3_1.dll’ under Windows), is a validating parser that allows you to read and write documents as well as perform validation against a DTD or schema. The Xerces parser is in the root of the Omnis Studio folder, in the same location as the Omnis.exe program. Under macOS, the parser library is called Xerces.Classic.Lib and it is bound into the oXML component.

Displaying XML Documents

Most recent browsers will display XML documents in a collapsible/expandable format. For most purposes though, you need to extract data from your XML documents for subsequent data processing, or enumerate an entire document in order to build a list for display in an Omnis tree list.

The display or transformation of the XML data is handled at the time of delivery when the document is retrieved from a document store and displayed on a client machine. The idea of XML is to store your data in a raw but structured state, allowing you to query and present it in many different ways as and when required.

What is the DOM?

To use oXML to access your XML documents, you need some knowledge of the DOM. Like XML, the DOM API is well documented in print and on the web so consult these external sources for further information.

The Document Object Model (DOM) is an API that allows you to build documents, navigate their structure, and add, modify, or delete elements and content. To quote from www.w3c.org, the “Document Object Model (DOM) is an application programming interface (API) for XML [and HTML] documents. It defines the logical structure of documents and the way a document is accessed and manipulated. In the DOM specification, the term “document” is used in the broad sense - increasingly, XML is being used as a way of representing many different kinds of information that may be stored in diverse systems, and much of this would traditionally be seen as data rather than as documents. Nevertheless, XML presents this data as documents, and the DOM may be used to manage this data.”

The oXML component uses the DOM Level 2 API to access XML documents, as defined on the W3C web site. It is built upon source code provided by the Xerces project, available as part of the Apache XML project: please see their web site for background information (http://xml.apache.org). For our purposes, the DOM provides a platform-independent interface to XML documents so that Omnis developers can use Omnis code and the notation to navigate and manipulate XML documents. DOM treats an XML document as a hierarchy of nodes, arranged in a tree structure, and accessible via its API and its methods.

The methods available in oXML closely match those defined by the DOM, so for a fuller explanation of the DOM API and its interfaces and methods you should consult the www.w3.org web site, or a good XML book or reference guide. The following URL has a definition of the DOM Level 2 API:

http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html

Documents and Nodes

The DOM represents a document as a tree of nodes or objects. Each node represents a different part of the document, hence a node can be one of a number of different types of node. In addition, each node or object type can have children, but only certain types of children are allowed for each type of node. The following table shows you what objects are returned (if any) when you query the children of a node in the document tree.

Node type	Possible Children
Document	Element (maximum of one), ProcessingInstruction, Comment, DocumentType
DocumentFragment	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
DocumentType	no children
EntityReference	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Element	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
Attr	Text, EntityReference
ProcessingInstruction	no children
Comment	no children
Text	no children
CDATASection	no children
Entity	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Notation	no children

See the Reference section later in this manual for a complete list of properties and methods for each type of node object.

Creating a Document Object

The oXML component is an e*xternal object* which is a type of external component that contain *methods* that you can use by instantiating an *Object Variable* based on the oXML external object. The oXML component is stored in a component library, called Oxml.dll under Windows, and is in the XCOMP folder under the main Omnis folder. The oXML component is always loaded by default so there is no need to load it via the External Components option in the Component Store.

Creating an Object Variable

You can add a new XML object in the method editor by inserting a variable of type Object and using the subtype column to select the XML Document object. You can click on the subtype droplist and select the XML object from the Select Object dialog. You can ignore the group of DOM Types: you must create a document object to access the whole of an XML document and use the object's methods to return the different parts or elements in the document.

An object icon plus the type "DOM Document" will appear in the variable subtype cell showing the type of object.

Inspecting an Object Variable

When an instance of the external object has been constructed (in an open window or form), you can inspect its properties and methods during development using the Interface Manager.

You can drag a method of the XML object from the Interface Manager into the Code Editor for the *Do command*. For example, when you inspect a DOM Document object (as shown), the Interface Manager will list the methods of a document object as well as the general properties of a node, since the root node is at the head of the document. To load a specified XML document you can use the $loadfile() or $loadbinary method (see below in the section Loading an XML document for the code). While debugging your code, you can inspect an object variable by right-clicking on the variable and selecting the Variable [varname] option. The following shows an object variable containing a document.

If true, the $useobjectrefs property ensures oXML returns object references rather than objects, that is, object return values are object references and object parameters must be object references. This property is automatically set in new returned objects to the value of $useobjectrefs in the object returning the new object.

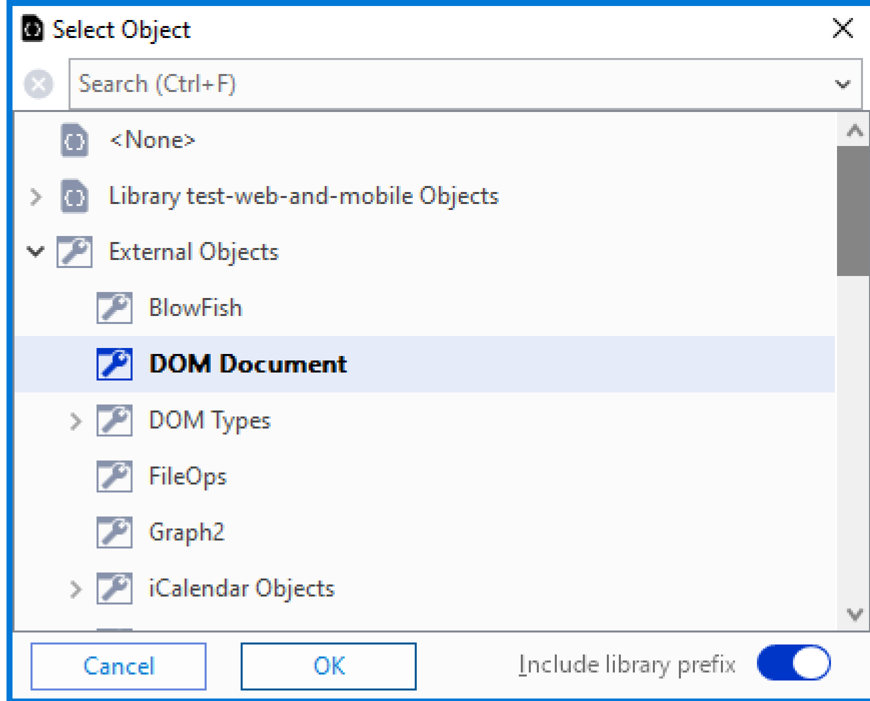


Figure 24:

NewWindow methods

View Modify Debug Stack Breakpoints

	Variable	Type	Subtype	Init.Val/Calc	Description
1	iEnt	Boolean	N/A		
2	iValidate	Boolean	N/A	kTrue	
3	path	Character	100000000		
4	xml	Object	DOM Document		

Figure 25:

Interface Manager - DOM Document

Search (Ctrl+F)

Methods | Properties

Method	Parameters
$loadbinary	xData, &cErrorText [, cSearchDir]
$loadfile	**cPath, &cErrorText**
$loadlist	lList, &cErrorText [, cDocTemplatePath]
$nextsibling	
$ownerdocument	
$parentnode	
$previoussibling	

Parameters | Description

Name	Type	Subtype	Description
cPath	Character	100000000	The pathname of the XML file
&cErrorText	Character	100000000	The error text returned from the XML e...

Returns Boolean (N/A)

Figure 26:

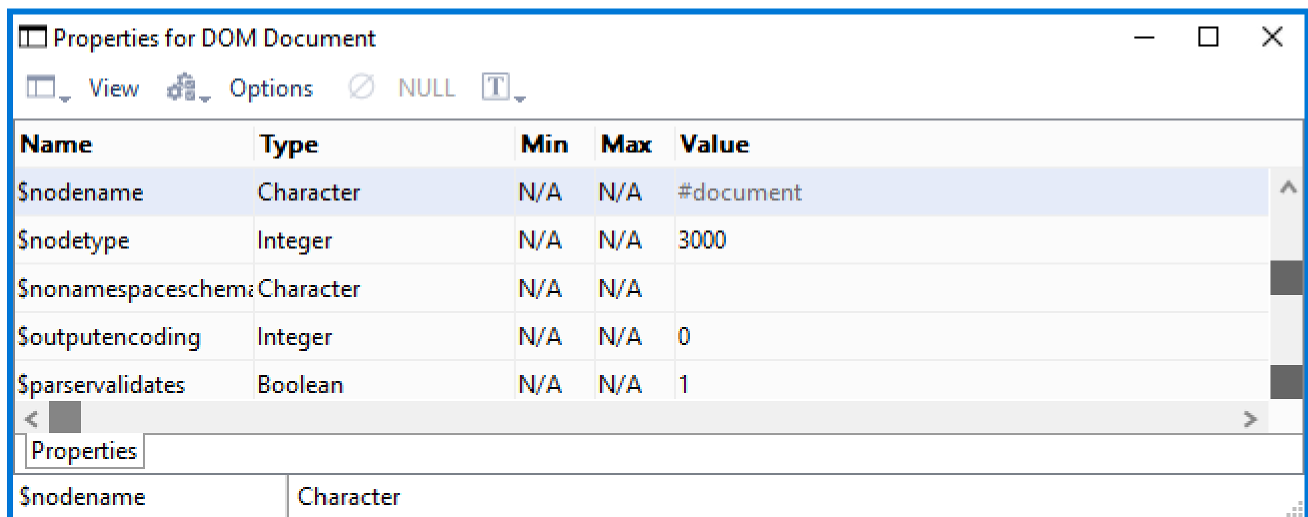


Figure 27:

Document Objects in oXML

In the DOM, documents have a logical structure which is like a tree, that is, all the elements and objects in the document are arranged in a hierarchical structure. Each object in a document is treated as a "node". The objects or nodes in a document can be one of a number of different types: an element, a comment, an attribute, text, or an entity. The oXML component contains a number of different objects, to support the different objects defined by the DOM.

The key object in oXML is the "DOM Document" object, which represents an XML document. You can use DOM Document as the subtype of an object variable, in order to use oXML and access its methods to manipulate your documents.

Having returned an XML document into your object variable, you can use various methods to return the objects within the document. For example, the $documentelement() method returns a DOM Element object that, in this particular case, represents the root of the XML document (see below the section Getting the document root element for the code). You can return and manipulate the following objects:

- **DOM Attribute**
  an attribute or property of an element
- **DOM CDATASection**
  the CDATA section of an attribute definition
- **DOM Comment**
  provides access to the content of a comment
- **DOM DocumentFragment**
  allows you to load part of a document into memory rather than the whole document
- **DOM DocumentType**
  a list of document types within a document
- **DOM Element**
  an element within a document; provides access to an element's attributes
- **DOM Entity**
  an entity within a DTD
- **DOM EntityReference**
  the representation of an entity
- **DOM NamedNodeMap**
  a collection of unordered nodes within a document
- **DOM NodeList**
  an ordered list of nodes within a document
- **DOM Notation**
  an unparsed or non-textual entity within a DTD, or the formal declaration of Processing Instruction target
- **DOM ProcessingInstruction**
  a special element or tag that provides instructions parsed to an external application, e.g. the XML version declaration at the beginning of a document
- **DOM Text**
  the text or content of an element or attribute

These objects are never used directly as the subtype of an object variable. Instead, they are returned by methods of the oXML object.

Manipulating XML Documents

Having created and instantiated an object variable, based on oXML, you can use its methods to load an XML document, enumerate the different parts or elements of the document, and display it in an Omnis tree list component. The following sections show how you can do this using Omnis methods.

Loading an XML Document

You can use the $loadfile() or $loadbinary() method to load an XML document into the document object (variable).

- **$loadfile**(cPath,&cErrorText)
  loads the XML file with pathname cPath into the document object, and returns true for success, or false and cErrorText for failure.
- **$loadbinary**(xData,&cErrorText[,cSearchDir])
  loads a document stored in the binary variable xData into the document object; returns true for success, or false and cErrorText for failure; when parsing from memory, cSearchDir is the directory in which the parser looks for externally referenced files.

When the $loadfile() method is executed the specified document is loaded into the object object/variable. In effect, this method loads the whole document, that is, a DOM Document representing the document. The document object has the general properties of a node ($nodename=#Document and $nodetype=kXMLNodeDocument) together with the properties of a document object $parservalidates, $replaceentityreferences, and $outputencoding, as well as many other methods for manipulating or traversing the document tree.

The following method can be placed behind a button and be used to prompt the user to select an XML document. The method then calls a class method to build the tree corresponding to the structure of the document selected by the user.

```
# Method '$event' for Load XML button
# create instance vars: path (Char), xml (Object, DOM Document), iValidate (Bool), iEnt (Bool), error (Num, Lo
On evClick
  Calculate path as
  Do FileOps.$getfilename(path,"Select XML file to parse")
  # prompts the user for an XML file name and location
  If path<>''
    Calculate $cinst.$title as path
    Calculate xml.$parservalidates as iValidate ## optional
    Calculate xml.$replaceentityreferences as iEnt ## optional
    Do xml.$loadfile(path,errorText) Returns error
    If error=0
      OK message Parser Error {[errorText]}
    Else
      Do method $buildtree ## see below
    End If
  End If
```

Note that the variables iValidate and iEnt can be added to your window or application (these preferences can be assigned to check boxes on a window) and used to force the XML parser to validate your document and resolve all entities.

Getting the Document Root Element

Having loaded the XML document into the document object, you can get the root element using the $documentelement() method; the method has no parameters.

- **$documentelement()**
  returns the DOM Element object representing the root of this XML document.

The following method prepares the window tree list, gets the root element from the document object and calls another method to build up a complete tree containing all the sub-nodes in the document.

```
# Method '$buildtree'
# create local variables nodetext (Char), obj (Object), tree (Item ref), treenode (Item ref)
Set reference tree to $cinst.$objs.tree
Do tree.$clearallnodes()
Calculate obj as xml.$documentelement()
# the root element is returned and placed in 'obj'
Do method $getelementtext (obj) Returns nodetext ## see below
Set reference treenode to tree.$add(nodetext)
Do method $addchildren (obj,treenode) ## see below
```

Getting the Attributes of an Element

When you have placed an element, such as the root element, into an object variable you can get its text value held in the $tagname property and access its attributes, if the element has any, using the $attributemap() method.

- **$attributemap**(&cErrorText)
  returns a named node map object listing the attributes of the element, or NULL and cErrorText if an error occurs; the named node map contains an unordered list of attributes belonging the element

Having created the list of attributes (a named node map) for an element you can step through the list using the $item() method in a For loop to extract each attribute.

- **$item**(iIndex)
  returns an attribute object referenced by iIndex from the name node map; indexing starts at zero; a bad index results in a NULL return value.

The following method gets the text value of the element passed to it and, assuming the element has attributes, adds the attributes in 'name=value' pairs. The properties $attname and $attvalue give you the name and value of an attribute.

```
# Method '$getelementtext'
# create parameter var element (Object)
# create local vars att (Object), attlist (Object), k (Long int), nodetext (Char)
Calculate nodetext as con('<',element.$tagname,'>')
# returns the element or tag name in nodetext
Calculate attlist as element.$attributemap
# builds a 'namednodemap' or list of object attributes: this is empty if there are no attributes and code skip
For k from 0 to attlist.$length-1 step 1
  # $length is the number of attributes in the named node map
  Calculate att as attlist.$item(k)
  # $item() returns the specified attribute in the list
  Calculate nodetext as con(nodetext,' ',att.$attname,' = ',att.$attvalue)
  # $attname and $attvalue are properties of an attribute
End For
Quit method nodetext
```

Adding Children to a Node

Since XML documents are highly structured it is relatively easy to step through the node tree and enumerate all its nodes and sub-nodes (children and grandchildren). You can use the $childnodes() method to get a list of children for a node, and then construct each child node according to its type by querying its $nodetype property.

- **$childnodes**(&cErrorText)
  returns a node list object listing the children of this object, or NULL and cErrorText if an error occurs
- **$haschildnodes**()
  returns true if the object has children; note no parameters

The following method steps through the node tree passed to it and adds the text value of each node to a tree list. Note the switch statement branches on the $nodetype of the current node and constructs the nodetext accordingly.

```
# Method '$addchildren'
# create parameter vars pObj (Object) and pTree (Item ref)
# create local vars att (Object), attlist (Object), child (Object), j (Long int), k (Long int), nl (Object), n
Calculate nl as pObj.$childnodes()
For j from 0 to nl.$length-1 step 1
  Calculate child as nl.$item(j)
  Switch child.$nodetype
    Case kXMLNodeComment
```

```
      Calculate nodetext as con('// ',child.$textdata)
    Case kXMLNodeElement
      Do method $getelementtext (child) Returns nodetext
    Case kXMLNodeProcessingInstruction
      Calculate nodetext as con('PI: ',child.$pitarget,
      ' = ',child.$pidata)
    Case kXMLNodeText
      Calculate nodetext as con(child.$textdata)
    Case kXMLNodeCDATASection
      Calculate nodetext as con('CDATA: ',child.$textdata)
    Case kXMLNodeAttribute
      Calculate nodetext as con(child.$attname,  ' = ',child.$attvalue)
    Case kXMLNodeEntityReference
      Calculate nodetext as 'ER'
    Default
      Calculate nodetext as con('Unexpected node type: ',child.$nodetype)
  End Switch
  Set reference treenode to pTree.$add(nodetext)
  If child.$haschildnodes()
    Do method $addchildren (child,treenode)
  End If

End For
```

Saving a Document

You can save an XML document to a file on disk using the $savefile() method or to a binary variable using $savebinary().

- **$savefile**(cPath,&cErrorText[,bStripDT=kFalse,iFmt=kXMLformatNone])
  saves XML to pathname cPath; returns true for success, or false and cErrorText; strips prolog DOCTYPE if bStripDT is true; kXMLFormat... constant iFmt controls formatting
- **$savebinary**(&xXML,&cErrorText[,bStripDT=kFalse,iFmt=kXMLformatNone])
  saves XML to binary variable xXML;r eturns true for success, or false and cErrorText; strips prolog DOCTYPE if bStripDT is true; kXMLFormat... constant iFmt controls formatting

```
# Method for Save button
# create vars path (Char), errorText (Char), error (Long Int)
On evClick ## Event Parameters - pRow( Itemreference )
  Calculate path as
  Do FileOps.$putfilename(path,"Specify name of output XML file")
  If path<>''
    Do xml.$savefile(path,errorText) Returns error
    If error=0
      OK message Parser Error {[errorText]}
    End If
  End If
```

The format parameter for the $savefile() and $savebinary() methods is an integer and can take one of a number of constants, as follows:

- **kXMLFormatNone**
  The output XML is not formatted; the default if iFmt is omitted (that is, no tabs and carriage-return linefeed sequences are inserted)
- **kXMLFormatBasic**
  The output XML is formatted by the insertion of tabs and carriage-return linefeed sequences
- **kXMLFormatFull**
  The output XML is formatted by the insertion of tabs and carriage-return linefeed sequences; in addition, text nodes are formatted by removing all leading and trailing spaces, as well as tabs, carriage returns and linefeeds

Using Lists with XML

You can pass the contents of a document object into an Omnis list variable using the $savelist() method. Conversely, you can transfer the contents of an Omnis list, assuming it is in the correct format, to a document object using the $loadlist() method. You can therefore manipulate the contents of an XML document via an Omnis list.

- **$savelist**(&lList,&cErrorText [,bSkipWhiteSpace])
  saves the XML specified by the object into the list lList, skipping the whitespace within elements if bSkipWhiteSpace is true; returns true for success, or false and cErrorText for failure.
- **$loadlist**(lList,&cErrorText [,cDocTemplatePath])
  loads list lList defining an XML document into the object, returns true for success, or false and cErrorText for failure. cDocTemplatePath is optional and can specify a DTD document template.

The following methods show how you can read an XML document into and out of an Omnis list. The Save List method prompts the user for an XML document, loads the file into the document object, and saves the contents of the object into the Omnis list xmllist.

```
# method for save list button
On evClick
  Calculate path as
  Do FileOps.$getfilename(path,"Select XML file to parse")
  If path<>''
    Calculate xml.$parservalidates as iValidate
    Calculate xml.$replaceentityreferences as iEnt
    Do xml.$loadfile(path,errorText) Returns error
    If error=0
      OK message Parser Error {[errorText]}
    Else
      Do xml.$savelist(xmllist,errorText,iSkip) Returns error
    End If
  End If
```

The Load List method prompts the user for a file name for the output XML document using the FileOps method $putfilename(), prompts the user to identify a DTD for validation, transfers the contents of the Omnis list to the document object, and saves the contents of the document object to the XML disk file.

```
# method for load list button
On evClick
  Calculate path as ''
  Do FileOps.$putfilename(path,"Specify name of output XML file")
  If path<>''
    Calculate template as
    Do FileOps.$getfilename(template,"Select XML document template for output")
    Do xml.$loadlist(xmllist,errorText,template) Returns error
    If error=0
      OK message {Load list failed: [errorText]}
      Quit method
    End If
    Do xml.$savefile(path,errorText) Returns error
    If error=0
      OK message Parser Error {[errorText]}
    End If
  End If
```

Using Tree Lists with XML

You can save or transfer a document object containing an XML document to an Omnis list variable and display the document in a tree list object using the $savetree() method. The tree list can be either a standard window tree control or a web component displayed on a remote form.

- **$savetree**(&lList, &cErrorText, bExpanded [,bIgnoreDocumentElement, bSkipWhiteSpace])
  saves the XML specified by the document object to a list suitable for displaying in a data bound tree list object with the dataname lList, and returns true for success, or false and cErrorText for failure;

If true, bExpanded specifies that the tree list is expanded when displayed, bIgnoreDocumentElement specifies that the document root is ignored and not displayed, and bSkipWhiteSpace specifies whether or not white spaces within elements are ignored.

The $savetree() method can be used behind a button or FormFile object to transfer an XML document into a tree list for display. For example, the following remote form allows the user to view an XML document in a web browser.

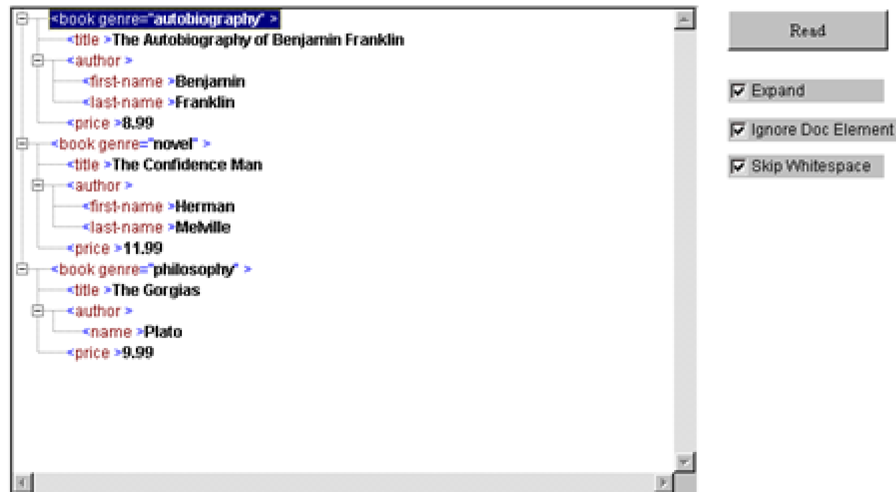


Figure 28:

The following method is behind a Read button (FormFile object) on the remote form. The method prompts the user to locate an XML document and displays the file in a web tree.

```
# create vars iXml (Object), iList (List)
# iExpanded, iIgnoreDocElem, iSkipWhitespace are linked to the check boxes on the form
# pFileData is passed to the method from the FormFile component and in this case contains the data from the XM
On evFileRead
  Calculate iXml.$parservalidates as kFalse
  Do iXml.$loadbinary(,cErrorText,cSearchDir) Returns #1
  If #1=0
    Do $cinst.$showmessage(cErrorText)
  Else
    Do iXml.$savetree(iList, cErrorText, iExpanded, iIgnoreDocElem, iSkipWhitespace) Returns #1
    If #1=0
      Do $cinst.$showmessage(cErrorText)
    End If
    Do $cinst.$redraw()

  End If
```

The first parameter of the $savetree() method is an Omnis list variable. When the method is executed the list is populated with the XML data from the document object. The list has a number of columns that are required to draw the tree list. The columns are: nodeType, path, value, attributes, iconid, ident, canedit, flags, and textcolor which are required to draw the tree list.

Document Templates

If you wish to build a document containing an XML construct such as a DTD or something that cannot be built using oXML, then provided that this information is fixed for each XML document, you can handle this by using a document template. The approach to building a document becomes:

1. Load the document template; this may contain an inline DTD or link to external DTD file, for example.

type	path	value	attributes	iconid	ident	canedit
3004	/bookstore		(Not empty)	0	1	False
3004	/bookstore/book		(Not empty)	0	2	False
3004	/bookstore/book/title	The Autobiography of Benjamin Franklin	(Not empty)	0	3	False
3004	/bookstore/book/author		(Not empty)	0	4	False
3004	/bookstore/book/author/first-name	Benjamin	(Not empty)	0	5	False
3004	/bookstore/book/author/last-name	Franklin	(Not empty)	0	6	False
3004	/bookstore/book/price	8.99	(Not empty)	0	7	False
3004	/bookstore/book		(Not empty)	0	8	False
3004	/bookstore/book/title	The Confidence Man	(Not empty)	0	9	False
3004	/bookstore/book/author		(Not empty)	0	10	False
3004	/bookstore/book/author/first-name	Herman	(Not empty)	0	11	False
3004	/bookstore/book/author/last-name	Melville	(Not empty)	0	12	False

Figure 29:

2. Add information to the document (elements, text etc.).

3. Save the document.

Character Sets and Unicode

XML documents can contain characters from any language including those represented by Unicode. oXML only works with documents that contain characters that can be converted to the local code page of the environment in which Studio is running, for example, under Windows the ANSI character set is used. Documents containing other characters can be loaded, but will not have the correct data when used in Omnis Studio.

Removing Invalid Characters

You can use the static function $removeinvalidcharacters to remove invalid characters from XML data.

- **$removeinvalidcharacters**
  $removeinvalidcharacters(&xData,iEncoding,iReplaceChar,&cErrorText) discards or replaces invalid XML characters in xData and returns the number of characters discarded or replaced, or NULL and cErrorText if an error occurs.
  xData: The data to check for invalid XML characters.
  iEncoding: The encoding of the xData parameter. One of kUniTypeAuto (defaults to kUniTypeUTF8 if encoding cannot be determined), kUniTypeUTF8, kUniTypeUTF16[BE|LE], kUniTypeUTF32[BE|LE], and kUniTypeNativeCharacters.
  iReplaceChar: Either -1 meaning discard invalid XML characters or the value of the character (0-255) used to replace the invalid XML characters.
  cErrorText: Receives error text if an error is returned.

Invalid XML characters are deemed to be characters less than space, that are not tab, carriage return or linefeed.

Creating XML documents

The oXML external component makes the handling of XML documents simple within an Omnis application. It treats each node as a separate object, enabling easy searching and manipulation of these nodes within the document. To create an XML document you need to create an Omnis Object with the subtype DOM document object and add different elements to the object. You can do this using the methods built into the document object.

Creating an XML Document

To create an XML document, first you must create a DOM document object (in the method editor Variable pane, Type is Object, and Subtype is under external objects -> DOM document, as described earlier in this manual). This is the master object of your document, and allows you to create, search, edit and delete all types of nodes within your document.

Adding an Element

Your XML document will be made up of several ‘elements’. Each element must have a name, but may also have several other properties associated with it, such as attributes, text, and comments. These will be discussed later.

Each element may also contain other elements (known as its children), thereby creating the tree type structure associated with the XML document.

Elements (and all other objects) are created by the DOM document object using its $createXXX() methods. For example, the following method returns an element object, oRoot, named ‘Root’:

```
# Define oXML as Object, with subtype DOM document object
# Define lError as Character
# Define oRoot as Object, with subtype DOM Element object
# Define oObj as Object, no subtype
Do oXML.$createelement('Root',lError) returns oRoot
```

Once you have created an element object, you must insert it into the document. This must be done from the element object that will become the parent of the element you are about to add. The element object has two methods for this:

- $appendchild()
  inserts the element at the end of its list of children.
- $insertbefore()
  inserts the element before the stated object.

As there are no elements when you create your first element, you must use the DOM document object as its parent: this creates what is known as the ‘Root’ element, of which there can only be one, and all other elements are descendants of this.

```
Do oXML.$appendchild(oRoot ,lError)
Do oXML.$createelement('Element1' ,lError) returns oObj
Do oRoot.$appendchild(oObj, lError)
```

The above method inserts the Root element into the document, then creates another element (Element1), which it returns in oObj (since oObj has no subtype, its type is defined when it has an object assigned to it; this keeps the number of variables down). The element is then inserted as a child of oRoot.

Properties of Elements

Although you have now added some elements, they contain no information. An element may have various properties associated with it. These are all added as children of the element, in the same way that elements are added as children of their parent elements. These may be added before or after inserting the element.

Adding Text

There are two possible ways of displaying text, parsed and unparsed. The usual method is parsed, which means your XML parser will evaluate the text. For example, “Apples & Pears” will be equated to “Apples & Pears”. Using unparsed will not evaluate the text and so will express it literally, in this case, “Apples & Pears”.

```
# to create PARSED text
Do oXML.$createtextnode("Apples & Pears", lError) Returns oObj

# to create UNPARSED text
Do oXML.$createcdatasection("Apples & Pears", lError) Returns oObj
```

Will create a text node containing the text, then to add it to the element oElement:

```
Do oElement.$appendchild(oObj, lError)
```

Adding Attributes

Attributes are added in a very simple manner. They require just a name and a value, and are added to the element with its $setattribute() method, as follows:

```
Do oElement.$setattribute("Colour","red",lError)
```

This will add the attribute Colour = "red" to the element oElement. You can use this method many times to add multiple attributes to the same element. Attributes can also be added using the usual method, such as oXML.$createattribute(), then the attribute is added as a child of element.

Adding Comments

Comments are not processed by XML parsers, but are present only to improve readability of the XML document. They follow the general form:

```
Do oXML.$createcomment("Your comment here", lError) Returns oObj
Do oElement.$appendchild(oObj, lError)
```

Processing Instructions

Processing instructions are used in XML as a way to keep processor-specific information in the text of the document. They store a 'target' and a value to pass that target. Again, these are created in the same way as the other objects, that is, oXML.$createprocessinginstruction(), then the processing instruction is added as a child of an element.

Entities

Entities are declared in the DOM document's DocumentType object. The oXML component is based on DOM level 2, which does not support the editing or creation of DocumentType objects. Therefore, the oXML component only allows the reading of entities already defined in an existing XML document.

Saving the XML File

Once you have created your DOM document in Omnis, with all the elements and so on in place, you need to save the document object to an .xml file. To do this, you can use the $savefile() method in the DOM document object.

```
Do oXML.$savefile("C:\MyFolder\MyXML.xml", lError, kFalse, XMLFormatFull)
```

The last argument of the $savefile() method allows you to specify the formatting of the output XML file. The formatting options let you specify whether or not to add carriage returns and line feeds, and to remove spaces, etc. Different parsers may require different formatting settings to display your XML file.

Chapter 6—oProcess

About oProcess

oProcess is a Worker Object (external component) providing a simple interface to launch and manage other processes, executables and applications, thereby providing you with greater interoperability from within Omnis Studio.

You can interact with oProcess using the standard worker methods, e.g. $init(), $run(), etc, which are described below, plus the external component has the common worker properties which are described here.

Properties

The oProcess object has the following properties:

Property	Description
$callbackinst	Sets the instance that will receive a worker’s callbacks
$cancancel	If kFalse, you can only cancel the worker forcefully. Defaults to kTrue
$elapsed	Seconds elapsed since the worker’s process was launched. Stops counting when the process returns an exit code
$pid	The worker’s process identification
$exitcode	The worker’s process exit code
$timeout	Seconds the worker’s process is allowed to run before getting cancelled. Defaults to 0 (no timeout)
$eol	End-of-line character which when encountered, a callback to the appropriate stream the worker’s process wrote to is executed. Defaults to kLf for Linux and macOS and kCr,kLf for Windows. Setting $eol to an empty string i.e. eol.assign(“”) will cause an immediate callback to the stream the worker’s process has written to
$state	Returns the worker’s current state
$errortext	Returns the error text associated with the last action
$threadcount	Returns the number of active background threads for all worker instances
$errorcode	Error code associated with the last action

Methods

$init()

$init(cProcess [,rArguments, cInitialDirectory, lEnvironment])

Initialises the worker to launch process in *cProcess* parameter. Use the *rArguments* row parameter to pass arguments to the process. *cInitialDirectory* can be used to launch the process with a different current directory. *lEnvironment* is a two-column list of environment variables and their values to be used during the process’ runtime. For example, launching the following process:

proc.$init(‘/bin/echo$TEST’,,,list(row(“TEST”,“Hello world!”)))

will result in callback to $stdout with “Hello world!” in the stdout column.

$run()

Runs the process on the worker’s main thread, therefore blocking code execution until the process returns. Should be avoided, unless there are specific synchronous requirements.

$start()

Starts the process on the worker’s background thread (non-blocking).

$cancel()

$cancel([bForce=kFalse])

Cancels the worker’s process. Pass kTrue for bForce parameter to close the process forcefully (currently supported only on Linux and macOS, sends SIGTERM signal). If bForce is kFalse, a SIGINT signal is sent. Note: if $cancancel property is kFalse and bFore is kFalse, the call to $cancel will be ignored: use kTrue for bForce to override the $cancancel property.

$completed()

$completed(wResults)

Callback method when the worker has finished running. wResults is a row with a retcode column containing the return code of the process and runtime_seconds column containing the seconds the process was alive for.

$cancelled()

Callback method when the worker's process has been cancelled.

$started()

Callback method when the worker's process has started and can now write to stdin. You can use this callback to work with processes that expect input as soon as they start running, e.g. when they prompt for a password.

$isrunning()

Returns kTrue if the worker's process is running, meaning that is has a PID greater than 0.

$stdout()

$stdout(wResults)

Callback method when worker's process writes to the stdout stream. wResults is a row with a stdout column containing the text the worker's process has written.

$stderr()

$stderr(wResults)

Callback method when worker's process writes to the stderr stream. wResults is a row with a stderr column containing the text the worker's process has written.

$write()

$write(cCharacters)

Writes cCharacters to the stdin stream of the worker's process.

$readlines()

$readlines(iStream [,nLines=0])

Returns a list containing all the lines written to kOProcessStd... stream, starting from the beginning of the stream. Use optional parameter nLines to limit the number of lines returned. For example:

$readlines(kOProcessStdin, 3)

will return the first 3 lines of the stdin stream.

iStream for $readlines() and $readtail() can be one of the following constants:

Constant	Description
kOProcessStdin	Identifier for the stdin stream
kOProcessStdout	Identifier for the stdout stream
kOProcessStderr	Identifier for the stderr stream

$readtail()

$readtail(iStream [,nLines=0])

Returns a list containing all the lines written to kOProcessStd... stream, starting from the end of the stream. Use optional parameter nLines to limit the number of lines returned. For example:

$readlines(kOProcessStdout, 3)

will return the last 3 lines of the stdout stream.

Using oProcess

Using the $init method you can run multiple bash commands as follows:

```
proc.$init("/bin/bash",row("-c","echo hey && echo hey2"))
```

or without using the arguments parameter

```
proc.$init("/bin/bash -c 'echo hey && echo hey2'")
```

On macOS and Linux, you can run processes as the root user as follows:

```
proc.$init("/usr/bin/sudo",row("-S","/usr/bin/whoami"))
```

and when the $started callback is received, call proc.$write(con("password",kLf)) to respond with the password. In this case, you will receive a call to $stdout with the stdout column containing "root", indicating that process is running with higher privileges.

On Windows, you cannot elevate the currently running process since the underlying APIs that make use of RunAs cannot redirect the stdout and stderr streams, suggesting that you cannot directly capture the output streams of an elevated process from a non-elevated process. Although the best way to ensure the elevated privileges are transferred to the process launched is to run Omnis with elevated privileges, you could do:

```
proc.$init('powershell.exe start powershell -Verb runAs -ArgumentList \"net session\" -WindowStyle hidden -Wai
```

to execute something as admin when Omnis is not running as admin, but you will not get the $stdout or stderr callbacks and you will not be able to use $write to the elevated process, making it a run-and-forget process.

Chapter 7—OW3 Worker Objects

You can build "low-level" Web- and Email-based communications into your Omnis applications using a number of different techniques or commands: this includes support for HTTP, SMTP, POP3, IMAP, and FTP communications and protocols; with the addition of JavaScript (Node.js), CRYPTO, HASH, and OAUTH2 in Studio 10, and LDAP and Python in Studio 11.

IMPORTANT: We recommend you use the OW3 Worker Objects (OW3) for all new development, since the older Web Worker Objects (OWEB) and the External commands are no longer supported.

The technique you choose to implement such support will depend on the breadth of support you require and the version of Omnis Studio you are using. The following techniques are available:

External package	Supported protocol	Omnis Studio version	Implem
OW3 Worker Objects (OW3)	HTTP, SMTP, FTP & SFTP, IMAP, JavaScript (Node.js), POP3, CRYPTO, HASH(1) OAUTH2(2) LDAP, Python(3)	(1) Studio 10(2) Studio 10.2(3) Studio 11	Objvar.$
OW3 Worker Objects (OW3)	HTTP, SMTP, FTP (not secure), IMAP	Studio 8.1	Objvar.$
Web Worker Objects (OWEB)	HTTP, SMTP	Studio 6.1.2	Objvar.$

External package	Supported protocol	Omnis Studio version	Implem
External Commands(Note these are obsolete in Studio 10 or above and are no longer available)	HTTP, FTP, SMTP, POP3, and IMAP	Studio 1.x (previously called *Web Enabler* in Omnis 7)	*Externa* prefixed e.g. HTT

Example Apps

There is an example app for most of the **OW3 Worker Objects** available in the **Samples** group under the **Hub** in the Studio Browser to demonstrate the use of the OW3 Worker Objects; search for “worker” in the search box (in the window title bar/toolbar) under **Samples** in the Studio Browser. There are examples for: Crypto, FTP, Hash, HTTP, IMAP, POP3, SMTP, JS Worker, and LDAP.

Using the OW3 Workers

Using the OW3 Worker Objects you can execute a potentially long-running task on a background thread, such as running a large mailshot, that reports back to the main thread when the task is complete. Indeed, the OW3 workers allow you to execute multiple tasks simultaneously allowing you to increase the efficiency and workload of your app.

The OW3 worker objects use the open-source *CURL library*, and native secure connection implementations for Windows and macOS, so they should have fewer deployment issues than the implementations available in previous versions.

The web and email commands in any of the OW3 Workers are accessed via one of the *Worker Objects* available under the *OW3 Worker Objects* group in the **Object Selection** dialog in the Method Editor (do not use the *Web Worker Objects* group which contains the old OWEB worker objects). To use the web and email commands, you need to create an **Object variable** and set its **subtype** to one of the OW3 worker objects, such as HTTPClientWorker or FTPClientWorker, under the OW3 Worker Objects group. Alternatively, you can create an **Object class** and set its superclass to one of the OW3 worker objects, then create an Object variable or Object reference variable and set its subtype to the object class name. Having created the variable you can call the web or email commands (methods) using *OBJECTVAR.$methodname*.

All OW3 Worker Objects share the same base functionality, plus they have additional functions specific to their respective web or email protocol.

HTTP/2 support

From Studio 11, the OW3 Workers support HTTP/2 which is more secure than HTTP as it uses binary protocols instead of plaintext, and is generally faster and more efficient for web communication.

The nghttp2 open source library is included to accommodate HTTP/2 support, and various libraries have been updated including: zlib, mbedTLS, libssh2, and libcurl; if your application uses OW3 your product licensing should include the appropriate third-party licensing.

Base Worker Properties

All OW3 worker objects have the following properties:

Property	Description
$callprogress	If true, and the worker is invoked to execute asynchronously using $start, the worker periodically generates a notification to $progress as it executes. Must be set before calling $start.The $progress method is described in the Methods section below.
$curloptions	Use this property to set internal CURL options not otherwise exposed by the worker. A two-column list, where column 1 is a number (the CURL easy option number) and column 2 is a string. The internal option must use either an integer or string value.Normally, you would not use this property, but if you do use it, you will need to consult the libcurl header files and documentation to obtain easy option numbers and values. You should use this option with care, as there is a chance you could cause Omnis to crash by passing an incorrect option value.

Property	Description
$errorcode	Error code associated with the last action (zero means no error)
$errortext	Error text associated with the last action (empty means no error)
$protocollog	If non-zero, the worker adds a log of protocol activity as a column named log to its wResults row. The desired value must be set before calling $run or $start. Defaults to kOW3logNone. Otherwise, a sum of kOW3log… constants; see below
$state	A kWorkerState… constant that indicates the current state of the worker object, one of the following: kWorkerStateClear, kWorkerStateInit, kWorkerStateCancelled, kWorkerStateRunning, kWorkerStateComplete
$threadcount	The number of active background threads for all instances of this type of worker object. In this case, type means the type of the subclass of the common base class e.g. HTTP
$timeout	The timeout (in seconds) for requests. Zero means requests do not time out. The desired value must be set before calling $run or $start. Defaults to 10

Request Completion

The **$alwaysfinish** property allows asynchronous requests to continue to completion after the instance containing the OW3 object destructs; the property only applies to the HTTP, IMAP, SMTP, POP3 and FTP workers.

When the instance containing an OW3 worker closes, and the OW3 worker is executing via a call to $start(), the worker thread continues executing until completion in the background: in this case, no notifications will be generated, as there is not a suitable instance to receive them.

Note that even if $alwaysfinish is true, if you shut down Omnis before the request has completed, OW3 will cancel the request so that shutdown works correctly.

Base Worker Constants

Protocol Logging

OW3 worker objects can all use these constants to control protocol logging. Sum the constants to select the desired logging.

Constant	Description
kOW3logNone	No protocol logging occurs. Obviously, this value needs to be used on its own
kOW3logBasic	Basic protocol information such as headers is logged
kOW3logData	Application data sent or received is logged up to a maximum of 16k for each direction. If the data is not consistent with UTF-8 encoding, it is logged as a binary dump format rather than character
kOW3logSecureData	Secure connection data is logged
kOW3logHTML	The content of the generated log is HTML rather than plain text. This can be written to a file and displayed using OBROWSER on Windows and macOS platforms

Base Worker Methods

All OW3 worker objects have the methods described in this section. There are normal methods that you call, and callback methods that you override to receive a notification.

Normal methods

$run

The $run method runs the worker on the main thread. Returns true if the worker executed successfully. The callback $completed will be called with the results of the request.

$start

The $start method runs the worker on a background thread; can be called multiple times to run different threads simultaneously to perform different tasks at the same time. Returns true if the worker was successfully started. The callback $completed will be called with the results of the request, or alternatively $cancelled will be called if the request is cancelled.

$cancel

The $cancel method cancels execution of worker on a background thread. Will not return until the request has been cancelled.

$getsecureoptions

The $getsecureoptions method gets the options that affect how secure connections are established.

```
OW3.$getsecureoptions([&bVerifyPeer,&bVerifyHost,&cCertFile,&cPrivKeyFile,&cPrivKeyPassword])
```

Returns Boolean true for success, or returns false and sets $errorcode and $errortext if an error occurs.

The parameters are:

Parameter	Description
bVerifyPeer	If true, the worker verifies the server certificate. The default is true, and this results in greater security
bVerifyHost	If true, the worker verifies that the server certificate is for the server it is known as. The default is true, and this results in greater security
cCertFile	For macOS, the pathname of the .p12 file containing the client certificate and private key, or its keychain name.For other platforms, the pathname of the client certificate .pem file. Empty if a client certificate is not required
cPrivKeyFile	Ignored on macOS.For other platforms, the pathname of the private key .pem file. Empty if a client certificate is not required
cPrivKeyPassword	The private key password. Empty if a client certificate is not required

$setsecureoptions

The $setsecureoptions method sets the options that affect how secure connections are established (call $setsecureoptions before calling $run or $start).

```
OW3.$setsecureoptions([bVerifyPeer=kTrue,bVerifyHost=kTrue,cCertFile='',cPrivKeyFile='',cPrivKeyPassword=''])
```

Returns Boolean true for success, or returns false and sets $errorcode and $errortext if an error occurs.

The parameters are:

Parameter	Description
bVerifyPeer	If true, the worker verifies the server certificate. The default is true, and this results in greater security

Parameter	Description
bVerifyHost	If true, the worker verifies that the server certificate is for the server it is known as. The default is true, and this results in greater security
cCertFile	For macOS, the pathname of the .p12 file containing the client certificate and private key, or its keychain name. For other platforms, the pathname of the client certificate .pem file. Empty if a client certificate is not required
cPrivKeyFile	Ignored on macOS. For other platforms, the pathname of the private key .pem file.Empty if a client certificate is not required
cPrivKeyPassword	The private key password. Empty if a client certificate is not required

$parserfc3339

The $parserfc3339() static method returns an Omnis date-time value from a RFC3339 formatted time.

```
OW3.$parserfc3339(cRfc3339[,bUTC=kTrue,&iOffset,&cErrorText])
```

Parses a date and time value conforming to RFC3339 and returns an Omnis date-time value and optionally the time zone offset in minutes.
Returns #NULL if the string cannot be parsed.

The parameters are:

Parameter	Description
cRfc3339	a date and time string conforming to RFC3339
bUTC	If true, the returned date-time value is in UTC rather than the local time zone of the RFC3339 date-time value
iOffset	If the RFC3339 date and time string is parsed successfully this receives the time zone offset in minutes
cErrorText	If supplied, receives text describing the error that caused $parserfc3339 to return #NULL

$splitmultipart

Allows you to split multipart content of a rest call, plus the MIME list returned by the OW3 methods that contain body part headers.

```
OW3.$splitmultipart(cContentType, xContent, &lMIMEList [,iDefCharSet=kUniTypeUTF8, &cErrorText])
```

Splits MIME-encoded multi-part xContent into lMIMEList.cContentType must include a boundary parameter. Returns true if successful. The parameters are:

Parameter	Description
cContentType	The content type header (must contain a boundary parameter)
xContent	The binary content to split
lMIMEList	Receives the MIME list created by splitting the MIME content. See the documentation for the MailSplit command to see how a MIME list is structured; however note that the charset in the OW3 MIME list is a kUniType… constant
iDefCharSet	The default character set used to convert character data when there is no charset specified for a MIME text body part. A kUniType… constant (not Character/Auto/Binary)
&cErrorText	If supplied, receives text describing the error that caused $splitmultipart to return false

The MIME list (for this call and for the other OW3 calls that generate a MIME list) now contains an additional column named bodypart-headers. This is a row containing a column for each non-empty header present for the body part. In addition, it has a column named "name" which contains the content-disposition header name parameter. All header names are normalized in the same way as those passed to RESTful services, that is, lower-case with any - characters removed.

Callback methods

$completed

When a worker is started using either $run or $start, it reports its completion by calling $completed. Override the $completed method of the worker object to receive this notification. It is called with a single row variable parameter. The columns of the row are specific to each type of worker object, so they are described in each specific worker object section.

$cancelled

To receive a notification that a request has been cancelled using $cancel, override the $cancelled method of the worker object. It is called with no parameters.

$progress

To receive progress notifications, override the $progress method of the worker object. OW3 worker objects generate notifications to $progress as and when some data has been transferred. Progress notifications will not be generated any more than once a second. Each notification receives a row variable parameter. The row has 4 columns.

Column	Description
downloadTotalBytesExpected	The total number of bytes expected to be downloaded from the server. This may always be zero, for example when the server is using chunked HTTP transfer encoding
downloadBytesSoFar	The number of bytes downloaded from the server so far
uploadTotalBytesExpected	The total number of bytes expected to be uploaded to the server
uploadBytesSoFar	The number of bytes uploaded so far

For the OW3 FTP worker, $progress can be called for synchronous operations.

OAUTH2 Worker Object

Support for OAUTH2 authorization has been added to Studio 10.2 by adding a new OAUTH2 Worker Object in the OW3 Worker object package, while the HTTP, IMAP, POP3, and SMTP workers have been modified to support OAUTH2 authorization via the new dedicated worker object.

Why use OAUTH2

OAUTH 2.0 provides much improved security over and above simple username and password schemes. It is commonly used by many different service providers, such as Google mail, for which its use will become mandatory in 2020 (meaning less secure apps will no longer be supported). You can read about OAUTH 2.0 in RFC 6749 (https://tools.ietf.org/html/rfc6749)

An application wishing to use a service (using HTTP, IMAP, POP3, or SMTP) requires an ***Access Token*** of type "bearer". The application needs to be registered with the service, so it can identify itself to the service, and the registration process provides the application with a Client Id and a Client Secret, that identify the application to the service.

As an initial step, the user of the service must authorize the application to use the service. To do this:

- The application opens a Web browser at the Authorization Endpoint (a URL) of the service.
- The authenticated user agrees that the application can access the service.

- The server hosting the Authorization URL redirects the browser to a URL supplied when opening the Web browser. This request contains an Authorization Code.
- The application makes a request to the Token Endpoint (a URL) sending it the Authorization Code.
- The server hosting the Token URL returns various pieces of information to the application, including: Access Token, Expiry of Access Token (recommended but not mandatory), Refresh Token (optional).

At this point, the application can use the Access Token to make requests to the service. Access Tokens *are short-lived, typically being valid for about an hour.* If the Token URL server also returned a Refresh Token, the application can use that after the Access Token has expired to obtain a new Access Token, without any further interaction with the user. Refresh Tokens typically have a long lifetime, but may be invalidated for various reasons, depending on the service implementation.

Obtaining Authorization

The OAUTH2 Worker allows you to obtain an Authorization Code, exchange it for tokens, and refresh tokens, using the $authorize() method on a background thread. The worker also contains methods to save and load the tokens and other related information to and from an encrypted binary block of data, which helps to protect key pieces of information such as the Refresh Token and Client Secret.

Note that you must always use an Object Reference to create the OAUTH2Worker object – this eliminates potential issues with the way Omnis uses the OAUTH2Worker as a property value.

The object reference to an OAUTH2Worker object containing the authorization information can be passed to the $oauth2 property in the HTTP, IMAP, POP3, and SMTP Workers to provide authorization.

The $addclientdetailstotokenrequest boolean allows you to include or remove client credentials from the body of a token request (when Omnis exchanges the authorization code for the access token).

OAUTH2 Properties

These properties are specific to OAUTH2.

Property	Description
$accesstoken	The access token to be used with HTTP, IMAP, POP3 and SMTP connections
$accesstokenexpiry	The expiry date and time of the access token (in UTC time). #NULL means no access expiry date and time is available
$addclientdetailstotokenrequest	If kTrue (default), the client id and client secret are added to the body of the request to get the token. If kFalse, the client id and client secret are not added to the token request
$authorizeurl	The URL of the OAUTH2 authorization endpoint
$clientid	The Client Id used in conjunction with the client secret to identify the application to the OAUTH2 authorization server
$clientsecret	The Client Secret used in conjunction with the Client Id to identify the application to the OAUTH2 authorization server

Property	Description
$redirecturiserveraddress	If not empty (the default value), this property overrides localhost in the redirect URI server address, replacing localhost with the value of this property. The default is localhost rather than 127.0.0.1 when generating redirect URIs when running in a thick client remote task. If using the default redirect URI, Omnis will pass localhost[:$serverport]/api/__oauth2/omnis to the service | | $refreshtoken | The Refresh Token to be used to request a new access token after the access token has expired | | $scope | A string identifying the type of access required. Used as part of the URL used to open the Web Browser at the authorization endpoint. For example, when using Google to access GMAIL, specify the scope as “https://mail.google.com/” | | $tokenurl | The URL of the OAUTH2 token endpoint | | asof 35659 $oauth2state | Custom content to be appended to the 32-character UUID in the state query string parameter |

HTTP and General Properties

In addition to the OAUTH2 properties, the OAUTH2Worker also has various HTTP and general OW3 properties, that for example affect how the HTTP requests it makes are executed: the OAUTH2Worker makes two different HTTP requests: a request to exchange an Authorization Code for an Access Token, and a request to obtain a new Access Token using the Refresh Token.

Property	Description
$errorcode	Error code associated with the last action (zero means no error)
$errortext	Error text associated with the last action (empty means no error)
$followredirects	If true, the HTTP request will follow a server redirect in order to complete the request. Defaults to false
$proxyserver	The URI of the proxy server to use for all requests from this object e.g. http://www.myproxy.com:8080. Must be set before executing $run or $start. Defaults to empty (no proxy server)
$proxytunnel	If true, and $proxyserver is not empty, requests are tunnelled through the HTTP proxy
$proxyauthtype	The type of HTTP authentication to use when connecting to $proxyserver. A kOW3httpAuthType… constant (see below)
$proxyauthusername	The username used to authenticate the user when connecting to $proxyserver using $proxyauthtype
$proxyauthpassword	The password used to authenticate the user when connecting to $proxyserver using $proxyauthtype
$state	A kWorkerState… constant that indicates the current state of the worker object
$threadcount	The number of active background threads for all instances of this type of worker object
$timeout	The timeout (in seconds) for requests. Defaults to 60 with a minimum value of 10
$protocollog	If non-zero, the worker adds a log of protocol activity as a column named log to its wResults row. The desired value must be set before calling $run or $start. Defaults to kOW3logNone. Otherwise, a sum of kOW3log… constants

OAUTH2 Standard Methods

$authorize

```
$authorize([iAuthFlow=kOW3OAUTH2authFlowCodeWithPKCE])
```

Starts the OAUTH2 authorization flow iAuthFlow on a background thread. Returns true if the thread was successfully started. Properties of the object cannot be assigned while $authorize() is running.

$authorize() opens a Web Browser at the authorization URL, passing the URL various parameters in the query string, such as the Client Id using the value of the $clientid property.

How the Web Browser is opened depends on the context in which $authorize() is called.

When executed within the context of a ***thick client*** (non-remote) task, $authorize() uses the $webbrowser property to control which browser it opens (note that you cannot use $authorize() with a thick client task when running in the headless server). It should be noted that when running in the thick client, $authorize() always uses a web browser rather than an embedded obrowser control due to best practice considerations documented in RFC 8252: https://www.rfc-editor.org/rfc/rfc8252.txt

To use $authorize() in the Runtime version of Omnis, you must set the **disableInRuntime** item in the 'server' section of the config.json file to *false*.

When executed within the context of a ***remote task,*** $authorize() will only work if the remote task is a JavaScript Client remote task. In this case, it uses the $showurl() mechanism of the JavaScript Client to open a browser window or tab. Note that in this case, you cannot execute both $authorize() and $showurl() in response to the same JavaScript client event.

When using the authorization flows that redirect the browser to a URI, $authorize() determines the redirect URI as follows.

For the *thick client,* it uses a loopback URI, to 127.0.0.1. Note that if the version of Omnis is not a server version, Omnis will still open a server port with limited support for OAUTH2 only, to allow the Authorization Code to be received via the redirect URI.

For the *JavaScript client,* $authorize() uses the RESTful URI determined from the Omnis server configuration. Note that this means that if you are using a Web Server to handle requests for your Omnis server, you need to set up the Omnis Web Server plugin for both the JavaScript client and RESTful requests.

$authorize() takes a single parameter, iAuthFlow, which can have one of the following constant values (kOW3OAUTH2authFlowCodeWithPKCE is the default):

Constant	Description
kOW3OAUTH2authFlowCode	The normal OAUTH2 authorization flow, where the authorization code will be received by redirecting the browser to a URI served by Omnis
kOW3OAUTH2authFlowCodeWithPKCE	Identical to kOW3OAUTH2authFlowCode, except that the worker uses PKCE to further secure its requests for an authorization code; the default iAuthFlow (see https://tools.ietf.org/html/rfc7636)
kOW3OAUTH2authFlowManualCode	Like kOW3OAUTH2authFlowCode, except that the redirect URI is urn:ietf:wg:oauth:2.0:oob. This means that instead of the authorization code arriving at Omnis via the redirect URI, the user must copy the authorization code to the clipboard from the browser window, and paste it into Omnis or the JavaScript client; after pasting, the Omnis application must call the method $setauthcode() (described below)
kOW3OAUTH2authFlowManualCodeWithPKCE	Like kOW3OAUTH2authFlowManualCode, but also uses PKCE

Note that you would normally use PKCE unless the service does not support it.

Manual code support, via the clipboard, is provided in case you do not want to open up a port for the redirect URI when running in the thick client; however, note that not all services support the redirect URI urn:ietf:wg:oauth:2.0:oob.

When $authorize() completes (which if successful means that is has opened the browser, received the Authorization Code, and exchanged it for an Access Token etc) it generates a call to the callback method $completed().

$setauthcode

```
$setauthcode(cAuthCode)
```

Returns Boolean true for success.

Only applicable to kOW3OAUTH2authFlowManualCode and kOW3OAUTH2authFlowManualCodeWithPKCE, when the $authorize() thread is waiting for the Authorization Code. Called from the application to supply the pasted Authorization Code using the cAuthCode parameter.

$save

```
$save(&xOAUTH2[,xKey])
```

Saves the properties ($clientid, $clientsecret, $authorizeurl, $tokenurl, $scope, $accesstoken, $refreshtoken and $accesstokenexpiry) to the encrypted binary buffer xOAUTH2.

xKey is a 256 bit AES encryption key. If you omit xKey, OW3 uses a hard-coded default key.

Returns Boolean true for success.

$save provides a convenient way to save all of the OAUTH2 parameters required for communicating with a service. In particular, it lets you safely store the Refresh Token, so you can minimise the number of occasions on which a user needs to authorize access using $authorize().

You can further protect your client secret, by including the encrypted buffer generated by $save in your release tree.

$load

```
$load(xOAUTH2[,xKey])
```

Loads the properties ($clientid, $clientsecret, $authorizeurl, $tokenurl, $scope, $accesstoken, $refreshtoken and $accesstokenexpiry) from the encrypted binary buffer xOAUTH2 previously generated using $save().

xKey is a 256 bit AES encryption key. If you omit xKey, OW3 uses a hard-coded default key. You must use the same key as that used when calling $save().

Returns Boolean true for success.

Grant Types

From Studio 11, the OAUTH2 Worker supports multiple grant types: authorization_code, password, and client_credentials (as per RFC 6749 sections 4.1, 4.3 and 4.4).

The **$granttype** property takes one of the following grant types:

- **kOW3OAUTH2grantAuthorizationCode** (the default)
  the Authorization Code grant type (behaves as previous versions)
- **kOW3OAUTH2grantPassword**
  the Password grant type requires the new $username and $password properties to be specified
- **kOW3OAUTH2grantClientCredentials**
  the Client Credentials grant type requires $clientid and $clientsecret

The $granttype property is set to kOW3OAUTH2grantAuthorizationCode by default which corresponds to behavior versions prior to Studio 11, so existing code should run as before.

When $granttype is set to kOW3OAUTH2grantPassword, the **$password** and **$username** properties can be used to retrieve an authorization token. However, this is deemed to be insecure, when compared to more secure methods, and should not be used (unless a legacy system requires it).

When $granttype is set to kOW3OAUTH2grantClientCredentials, the properties **$clientid** and **$clientsecret** are used to obtain the authorization token. Note that when using this grant type, the OAUTH2 server may not return a refresh token (as per RFC 6749 section 4.4.3).

Adding custom content to OAUTH2 state parameter

The **$oauth2state** property can contain custom content to be appended to the 32-character UUID in the state query string parameter of the request, allowing you to identify requests sent from multiple instances of Omnis.

If you are handling this on a reverse proxy, you will have to URL-decode and look for your value after the first 32 characters, but it is important when proxying off the request to keep the UUID in the state, otherwise Omnis will not be able to match the callback to the initiated request.

OAUTH2 Callback Methods

$tokensrefreshed

The OAUTH2Worker has one non-standard callback method, $tokensrefreshed. The OAUTH2Worker generates a call to this method after it has successfully refreshed the tokens while it is being used in conjunction with the HTTP/IMAP/POP3/SMTP worker.

$tokensrefreshed() is called with no parameters; at this point, the worker has been updated with the new Access Token, Access Token Expiry and Refresh Token. A typical implementation of $tokensrefreshed() would use $save() to save the current tokens etc and then write the encrypted buffer to disk. It should be noted that calling the server to refresh tokens can result in a different updated Refresh Token - this needs to be used to refresh tokens the next time a refresh is required.

HTTP and General Methods

The OAUTH2Worker supports the normal methods $cancel(), $getsecureoptions() and $setsecureoptions(). The latter two relate to how secure connections to the Token URL are established.

HTTP Callback Methods

The OAUTH2Worker generates calls to the standard callback methods $cancelled() and $completed(). These correspond to a call to $authorize() to start the authorization code flow. The completion row passed as a parameter to $completed() has columns as follows:

Column	Description
errorCode	An integer error code indicating if the request was successful. Zero means success. If successful, $accesstoken, $accesstokenexpiry and $refreshtoken have been updated using the content received from the server; if no Access Token Expiry was received, $accesstokenexpiry is #NULL; if no Refresh Token was received, $refreshtoken is empty
errorInfo	A text string providing information about the error if any
scope	The scope returned from the server when requesting the Access Token, if different to the requested scope
log	If you used $protocollog to generate a log, this column contains the log data, either as character data, or UTF-8 HTML. Otherwise, the log column is empty

HTTP, IMAP, POP3, and SMTP Workers

Once you have used $authorize() to obtain an Access Token, you need to make the Access Token available to the worker with which OAUTH2 authorization is required. You do this by assigning a new $oauth2 property of the HTTP, IMAP, POP3, or SMTP worker:

- **$oauth2**
  Property that is an object reference to an OAUTH2Worker object containing the authorization information required to make requests to the server. Clear this property by assigning #NULL to it. $authorize() cannot run while the OAUTH2Worker is assigned to $oauth2

The supported workers use $oauth2 to obtain the Access Token for the request. To do this, it uses the following logic:

- If there is no Refresh Token ($refreshtoken is empty), it uses $accesstoken.
- If the $accesstokenexpiry is #NULL (there is no expiry date and time), it uses $accesstoken.
- If the expiry date time is more than 5 seconds away, it uses $accesstoken
- Finally, it uses $refreshtoken to refresh the token(s). If successful, it generates a call to $tokensrefreshed() in the OAUTH2Worker and it uses the new $accesstoken

You should note that there is a chance the request will fail when it is made near to the 5 second window before the Access Token expires. You should be prepared to handle this type of error in $completed, possibly retrying the request.

HTTP

After assigning $oauth2, the parameters iAuthType, cUserName, and cPassword passed to $init() are ignored in favour of using the Access Token stored in $oauth2.

IMAP, POP3, SMTP

After assigning $oauth2, the cPassword parameter passed to $init() is ignored in favour of using the Access Token stored in $oauth2. Note that cUserName is still required.

HTTP Worker Object

The HTTPClientWorker provides client HTTP support. For example, you can POST data to a server, execute a RESTful request, or download a file from a server.

The HTTP Worker supports the following externals:

- curl version 7.84.0
- libssh2 version 1.9.0
- mbedTLS version 2.16.2

Properties

The HTTPClientWorker has the following properties in addition to the base worker properties described earlier:

Property	Description
$followredirects	If true, the HTTP request will follow a server redirect in order to complete the request. The desired value must be set before calling $run or $start. Defaults to false
$proxyserver	The URI of the proxy server to use for all requests from this object e.g. http://www.myproxy.com:8080. Must be set before executing $run or $start. Defaults to empty (no proxy server)
$proxytunnel	If true, and $proxyserver is not empty, requests are tunnelled through the HTTP proxy
$proxyauthtype	The type of HTTP authentication to use when connecting to $proxyserver. A kOW3httpAuthType… constant. kOW3httpAuthType constants are described in the Constants section below
$proxyauthusername	The user name used to authenticate the user when connecting to $proxyserver using $proxyauthtype
$proxyauthpassword	The password used to authenticate the user when connecting to $proxyserver using $proxyauthtype
$responsepath	If not empty, the worker writes response content to the file with this path rather then adding it to the wResults row. The file must not already exist. The desired value must be set before calling $run or $start. Defaults to empty

Property	Description
$oauth2	An object reference to an OAUTH2Worker object containing the authorization information required to make requests to the server: see OAUTH2 Worker Object

Constants

The HTTPClientWorker uses the following constants, specified in the iMethod parameter in the $init method, in addition to the base worker constants described earlier:

Constant	Description
kOW3httpMethodDelete	Sends a DELETE method
kOW3httpMethodGet	Sends a GET method
kOW3httpMethodHead	Sends a HEAD method
kOW3httpMethodOptions	Sends a OPTIONS method
kOW3httpMethodPatch	Sends a PATCH method
kOW3httpMethodPost	Sends a POST method
kOW3httpMethodPut	Sends a PUT method
kOW3httpMethodTrace	Sends a TRACE method
kOW3httpAuthTypeNone	Indicates that no HTTP authentication is required (in this case a user name and password do not need to be supplied)
kOW3httpAuthTypeBasic	Indicates that basic HTTP authentication is required
kOW3httpAuthTypeDigest	Indicates that digest HTTP authentication is required
kOW3httpMultiPartFormData	Indicates that HTTP multipart/form-data content is to be sent (this is described below)

Methods

HTTPClientWorker has the methods described in this section in addition to the base worker methods described earlier.

Normal methods

$init

$init(cURI [,iMethod=kOW3httpMethodGet, lHeaders=#NULL, vContent='', iAuthType=kOW3httpAuthTypeNone, cUserName='',cPassword=''])

Called to prepare the object to execute a request, before calling $run or $start; the URI is the only required parameter.

Returns Boolean true for success, or returns false and sets $errorcode and $errortext if an error occurs.

The parameters are:

Parameter	Description
cURI	The URI of the resource, optionally including the URI scheme (http or https) e.g. http://www.myserver.com/myresource. If you omit the URI scheme e.g. www.myserver.com/myresource,the URI scheme de- faults to http. You can also include query string parameters if desired e.g. http://www.myserver.com/myresource?param1=test¶m2=test
iMethod	A kOW3httpMethod... constant that identifies the HTTP method to perform

Parameter	Description
lHeaders	A two-column list where each row is an HTTP header to add to the HTTP request Column 1 is the HTTP header name e.g. 'content-type' and column 2 is the HTTP header value e.g. 'application/json'. If you do not supply the header "accept-encoding" the worker automatically decompresses content compressed using gzip or deflate; however, if you supply this header, the worker does not perform automatic decompression
vContent	kOW3httpMultiPartFormData or a binary, character or row variable containing content to send with the request. kOW3httpMultiPartFormData means send the content built using the $multipart... methods described below. The worker sends binary data as it is. The worker converts character data to UTF-8 and sends the UTF-8. A row must have a single column containing the path of the file containing the content to send. If you do not specify a content-type header in lHeaders, the worker will generate a suitable type if it recognises the file extension when using a row, or when using a character value it will use text/plain;charset=utf-8. Otherwise, it will use application/octet-stream. In addition, the worker will automatically add a content-length header, so there is no need to pass this in lHeaders.From Studio 11, you can pass a raw List or Row which the worker will subsequently transform into JSON before executing the request. If a row is passed and it contains only one column, which is a path to a file that exists, the contents of the file will be used. If the file does not exist, the contents of the row will be converted to JSON and sent with the request. Code which passes JSON in a binary variable is not affected. The lists can contain sub-lists as the worker supports both JSON arrays of arrays, and arrays of objects
iAuthType	A kOW3httpAuthType... constant that specifies the type of authentication required for this request. If you omit this and the remaining parameters, authentication defaults to kOW3httpAuthTypeNone
cUserName	The user name to use with authentication types kOW3httpAuthTypeBasic and kOW3httpAuthTypeDigest
cPassword	The password to use with authentication types kOW3httpAuthTypeBasic and kOW3httpAuthTypeDigest

NOTE: If you call $init when a request is already running on a background thread, the object will cancel the running request, and wait for the request to abort before continuing with $init.

Example

The following code could be used to prepare an HTTPClientworker object using $init, and then $run can be used to execute the HTTP method. The method returns the content of the web page stored in iURI, e.g. ww.omnis.net.

```
# $execute method
Do method checkHttpObject ## sets up the HTTP object ref var
Do method setupLogging ## sets up logging based on user choice
If len(iTempContent)
    Do iHttp.$init(iURI,iMethodList.iMethod,iHeaderList,iTempContent,iAuthList.iAuthType,iUser,iPassword) Retur
Else
    If iSendContentMode=1
        Do iHttp.$init(iURI,iMethodList.iMethod,iHeaderList,row(iContentPath),iAuthList.iAuthType,iUser,iPassw
    Else If iSendContentMode=2
        Do iHttp.$buildmultipart(iContentPath)
        Do iHttp.$init(iURI,iMethodList.iMethod,iHeaderList,kOW3httpMultiPartFormData,iAuthList.iAuthType,iUse
    Else If iSendContentMode=0
```

```
        Do iHttp.$init(iURI,iMethodList.iMethod,iHeaderList,iContent,iAuthList.iAuthType,iUser,iPassword) Retu
    End If
End If
If not(lOk)
    OK message {Error [iHttp.$errorcode]: [iHttp.$errortext]}
    Quit method kFalse
End If
If pRun
    Do iHttp.$run() Returns lOk
Else
    Do iHttp.$start() Returns lOk
    If lOk
        Calculate $cinst.$objs.ScrollBox.$objs.cancel.$enabled as kTrue
        Calculate $cinst.$objs.ScrollBox.$objs.execute.$enabled as kFalse
        Calculate $cinst.$objs.ScrollBox.$objs.executethencancel.$enabled as kFalse
    End If
End If
If not(lOk)
    OK message {Error [iHttp.$errorcode]: [iHttp.$errortext]}
    Quit method kFalse
End If
Quit method kTrue
```

$multipartclear

$multipartclear()

Frees any previously generated multipart/form-data content. Note that calling $run or $start with kOW3httpMultiPartFormData results in the multipart/form-data content being automatically freed after use.

Returns Boolean true for success, or returns false and sets $errorcode and $errortext if an error occurs.

$multipartaddfield

$multipartaddfield(cName, cFieldData [,lPartHeaders])

Adds a field part to the multipart/form-data content stored in the worker object. To send this content specify kOW3httpMultiPartFormData as the vContent parameter to $init().

Returns Boolean true for success, or returns false and sets $errorcode and $errortext if an error occurs.

The parameters are:

Parameter	Description
cName	The name of the multipart/form-data field part
cFieldData	The value of the multipart/form-data field
lPartHeaders	A two-column list where each row is a header to add to the part. Column 1 is the header name and column 2 is the header value

$multipartaddfile

$multipartaddfile(cName, vFileData [,cFileName='', lPartHeaders])

Adds a file part to the multipart/form-data content stored in the worker object. A file part indicates to the server that a file is being uploaded. To send this content specify kOW3httpMultiPartFormData as the vContent parameter to $init().

Returns Boolean true for success, or returns false and sets $errorcode and $errortext if an error occurs.

The parameters are:

Parameter	Description
cName	The name of the multipart/form-data file part

Parameter	Description
vFileData	A binary, character or row variable containing the file data for the part. The worker sends binary data as it is. The worker converts character data to UTF-8 and sends the UTF-8. A row must have a single column containing the path of the file containing the content to send. If you do not specify a content-type header in lPartHeaders, the worker will generate a suitable type if it recognises the file extension when using a row, or when using a character value it will use text/plain;charset=utf-8. Otherwise, it will use application/octet-stream
cFileName	The filename of the part. Must be specified if vFileData is binary or character. If vFileData is a row (identifying a file) then this overrides the default filename (the name of the file)
lPartHeaders	A two-column list where each row is a header to add to the part. Column 1 is the header name and column 2 is the header value

Callback methods

$completed

The standard $completed callback is passed a row variable parameter with the following columns:

Column	Description
errorCode	An integer error code indicating if the request was successful. Zero means success i.e. the HTTP request was issued and a response received - you also need to check the httpStatusCode to know if the HTTP request itself worked
errorInfo	A text string providing information about the error if any
httpStatusCode	A standard HTTP status code that indicates the result received from the HTTP server
httpStatusText	The HTTP status text received from the HTTP server
responseHeaders	A row containing the headers received in the response from the HTTP server.The header values are stored in columns of the row.The column name is the header name converted to lower case with any - characters removed, so for example the Content-Length header would have the column name contentlength.If the client receives multiple headers with the same name, it combines them into a single header with a comma separated list of the received header values. This is consistent with the HTTP specification
responseContent	If you have not used $responsepath to write the received content directly to a file, this is a binary column containing the content received from the server
log	If you used $protocollog to generate a log, this column contains the log data, either as character data, or UTF-8 HTML. Otherwise, the log column is empty

WebSocket Client Support

$init

To initialise the OW3 HTTP worker object so that it is ready to create a WebSocket client connection, call $init with parameters as follows:

Parameter	Description
cURI	The URI of the WebSocket server, which must include the URI scheme (ws or wss) e.g. wss://demos.kaazing.com/echoYou cannot omit the URI scheme, because the HTTP worker defaults to using http
iMethod	Must be kOW3httpMethodGet

Parameter	Description
lHeaders	A two column list where each row is an HTTP header to add to the HTTP request.The worker automatically adds these headers when connecting to a WebSocket server, so do not add these headers:connection: upgrade upgrade: websocketsec-websocket-version: 13sec-websocket-key:
vContent	Not used
iAuthType	A kOW3httpAuthType… constant that specifies the type of authentication required for this request. If you omit this and the remaining parameters, authentication defaults to kOW3httpAuthTypeNone
cUserName	The user name to use with authentication types kOW3httpAuthTypeBasic and kOW3httpAuthTypeDigest
cPassword	The password to use with authentication types kOW3httpAuthTypeBasic and kOW3httpAuthTypeDigest

The standard OW3 properties $state, $errortext, $errorcode, $threadcount, $protocollog, $timeout, $callprogress and $curloptions all apply when connecting to a WebSocket server.

The standard OW3 methods $getsecureoptions and $setsecureoptions apply when connecting to a WebSocket server.

$run and $start

You cannot use $run to establish a WebSocket connection, since multiple threads are required to make it usable. So if you try to use $run, the worker returns kFalse and sets $errorcode and $errortext.

To establish a WebSocket connection, call $start. If $start succeeds, then the worker attempts to establish the connection to the WebSocket server in another thread.

If the connection cannot be established, the worker generates a notification to $completed, with a non-zero value in the errorCode member of the notification row parameter.

If the connection is established successfully (and is therefore open and ready for data transfer), the worker generates a notification to the new method $ws_onconnect. Override this method to receive this notification. $ws_onconnect receives a single row variable as its parameter. This row variable has a single column, responseHeaders, which is a row with a single column for each response header received from the server in the final HTTP protocol exchange resulting in the 101 (web socket protocol handshake) HTTP status code.

As soon as you have received the $ws_onconnect notification, the WebSocket is ready to send and receive data.

$wssend

After you have received the $ws_onconnect notification, you can send data using the method $wssend:

$wssend(vMessage)

Sends the supplied message on a connected web socket. Returns true if successful, which means the message has been queued for sending.

If vMessage is a character value, the worker converts it to UTF-8 before sending it as a text message; otherwise the value is treated as binary and sent as a binary message.

Receiving Data

Each message received from the WebSocket server generates a $ws_onmessage notification. Override this method to receive these notifications. $ws_onmessage receives a single row variable parameter, with 2 columns (named data and utf8). Column data is the binary data and column utf8 is boolean true if the data is UTF-8.

$wsclose

The client can close the WebSocket connection by calling the method $wsclose:

$wsclose([bDiscardUnsentMessages=kFalse,iStatusCode=1000,cReason=''])

Closes the connection to the web socket server. $completed() will be notified when the connection has closed. The row passed to $completed has closeStatus and closeReason columns that receive the values sent in the close frame to the server.

Pass bDiscardUnsentMessages as kTrue, to discard any completely unsent queued messages before sending the close frame to the server.

iStatusCode is an integer status code that indicates the reason for closure (one of the values in section 7.4 of RFC6455).

cReason is some optional text that indicates the reason for closure.

Server close

The server can send a close frame to the client, telling the client it is closing the connection. The client responds with a close frame, before the connection closes. Again, $completed is notified to tell the object that the connection has closed.

The row passed to $completed has closeStatus and closeReason columns that receive the values sent in the close frame from the server.

$cancel

You can use $cancel to terminate the connection in a non-graceful manner.

Ping-pong

If the client receives a Ping from the server, it automatically responds with a Pong. You can also set up the client to automatically Ping the server, and generate an error (closing the connection) if a Pong is not received. To do this, use these two properties:

$wspinginterval: If non-zero, and the connection is to a web socket server, the HTTP worker sends a Ping frame to the server every $wspinginterval seconds of inactivity, and closes the connection if a Pong frame is not received after $wspongtimeout seconds. Defaults to zero.

$wspongtimeout: The number of seconds (1-60, default 5) the client waits to receive a Pong frame after sending a Ping frame as a result of the $wspinginterval timeout expiring.

Timeout

The object restarts the $timeout timer each time it sends or receives some data.

SMTP Worker Object

The SMTPClientWorker provides client SMTP support, allowing you to use the worker to send emails, including bulk emails via a mailshot. The following sections describe the SMTP worker properties, constants and methods.

Properties

The SMTPClientWorker has the following properties in addition to the base worker properties described earlier:

Property	Description
$requiresecureconnection	If true, and the URI is not a secure URI, the connection starts as a non-secure connection which must be upgraded to a secure connection (using the STARTTLS command). If it cannot be upgraded then the request fails. Defaults to false
$keepconnectionopen	If true, the worker can leave the connection to the server open when it completes its request. Defaults to false. Note that even when this property is set to true, a protocol error may cause the connection to close. Use true if you are likely to send more emails using the same server fairly soon

Property	Description
$callmailshotprogress	If true, and the worker is sending a mailshot asynchronously via $start, the worker generates notifications to $mailshotprogress as it executes. $callmailshotprogress must be set before calling $start. Defaults to false
$oauth2	An object reference to an OAUTH2Worker object containing the authorization information required to make requests to the server: see OAUTH2 Worker Object
asof 35659 $allowpathinuri	If true, the SMTP Worker will accept paths in the URI used in the $init method, e.g. smtp://my.smtp.server:587/my.helo.address. Defaults to false

Constants

The SMTPClientWorker uses the following constants in addition to the base worker constants described earlier:

Constant	Description
kOW3msgPriorityLowest	The message has the lowest priority
kOW3msgPriorityLow	The message has low priority
kOW3msgPriorityNormal	The message has normal priority
kOW3msgPriorityHigh	The message has high priority
kOW3msgPriorityHighest	The message has the highest priority

Methods

SMTPClientWorker has the methods described in this section in addition to the base worker methods described earlier.

Normal methods

$init

$init(cURI, cUser, cPassword, vFrom, lTo, lCc, lBcc, cSubject, cPriority, lHeaders, vContent [,bMailshot=kFalse])

Called to prepare the object to execute a request, before calling $run or $start.

Returns Boolean true for success, or returns false and sets $errorcode and $errortext if an error occurs.

The parameters are:

Parameter	Description
cURI	The URI of the server, optionally including the URI scheme (smtp or smtps), e.g. smtp://test.com. If you omit the URI scheme e.g. smtp.myserver.com the URI scheme defaults to smtp. If the server uses a non-standard port, you can include it in the URI like this example smtp://smtp.myserver.com:2525
cUser	The user name to be used to log on to the SMTP server
cPassword	The password to be used to log on to the SMTP server
vFrom	The email address of the message sender. Either a character value e.g. user@test.com or a row with 2 columns where column 1 is the email address e.g. user@test.com and column 2 is descriptive text for the sender, typically their name

Parameter	Description
lTo	A one or two column list where each row identifies a primary recipient of the message. Column 1 contains the email address e.g. user@test.com and column 2 if present contains descriptive text for the recipient, typically their name
lCc	Empty if there are no CC recipients, or a one or two column list where each row identifies a carbon copied recipient of the message. Column 1 contains the email address e.g. user@test.com and column 2 if present contains descriptive text for the recipient, typically their name
lBcc	Empty if there are no BCC recipients, or a single column list where each row contains the email address of a blind carbon copied recipient of the message e.g. user@test.com. Unlike lTo and lCc, lBcc does not allow more than 1 column, as blind carbon copied recipients are not added to the message header and therefore the descriptive text is not required
cSubject	The subject of the message
iPriority	A kOW3msgPriority... constant that specifies the priority of the message
lHeaders	A two-column list where each row is an additional SMTP header to send with the message. Column 1 is the header name e.g. 'X-OriginalArrivalTime' and column 2 is the header value e.g. '23:02'
vContent	Message content. Either binary raw content (which the worker sends exactly as it is), or a list to be sent as MIME. See the documentation for the MailSplit command to see how a MIME list is structured; however, note that the charset in the worker MIME list is a kUniType... constant rather than a character string
bMailshot	Allows a mailshot to be sent (default is kFalse). If true, the worker sends a separate copy of the message to each recipient in the lTo list (so that each recipient cannot see the addresses of the others); only lTo is used, and lCc and lBcc must be empty

NOTE: If you call $init when a request is already running on a background thread, the object will cancel the running request, and wait for the request to abort before continuing with $init.

Example

The $init method can be used to prepare the SMTPClientWorker object to be executed using the $run or $start method. In this case, iSmtp is an object reference variable with its subtype set to an object class, which has its $superclass set to the SMTPClientWorker in the OW3 worker objects group.

```
# $start method
Do method setupLogging ## set up logging
Calculate iSmtp.$timeout as iTimeout ## set properties via window fields
Calculate iSmtp.$callprogress as iCallProgress
Calculate iSmtp.$keepconnectionopen as iKeepConnectionOpen
Calculate iSmtp.$requiresecureconnection as iRequireSecureConnection
Calculate iSmtp.$callmailshotprogress as iCallMailshotProgress
Do method $splitaddressentry (iFrom,lFromAddress,lFromDescription) Returns lOk
If not(lOk)
    OK message {From "[iFrom]" is invalid}
    Quit method kFalse
```

```
End If
Calculate lOk as $cinst.$makerecipientlist(lToList,iTo)
If not(lOk)
    OK message {From "[iTo]" is invalid}
    Quit method kFalse
End If
Calculate lOk as $cinst.$makerecipientlist(lCcList,iCc)
If not(lOk)
    OK message {From "[iCc]" is invalid}
    Quit method kFalse
End If
Calculate lOk as $cinst.$makerecipientlist(lBccList,iBcc)
If not(lOk)
    OK message {From "[iBcc]" is invalid}
    Quit method kFalse
End If
If iCallMailshotProgress
    Set reference lMailshotProgressItem to $clib.$windows.wMailshotProgress.$openmodal("*",kWindowCenterRelati
End If
Do iSmtp.$setMailshotProgressInst(lMailshotProgressItem)
If iNoMIME
    If len(lFromDescription)
        Do iSmtp.$init(iServerURI,iUser,iPassword,row(lFromAddress,lFromDescription),lToList,lCcList,lBccList,
    Else
        Do iSmtp.$init(iServerURI,iUser,iPassword,lFromAddress,lToList,lCcList,lBccList,iSubject,iPriorityList
    End If
Else
    If len(lFromDescription)
        Do iSmtp.$init(iServerURI,iUser,iPassword,row(lFromAddress,lFromDescription),lToList,lCcList,lBccList,
    Else
        Do iSmtp.$init(iServerURI,iUser,iPassword,lFromAddress,lToList,lCcList,lBccList,iSubject,iPriorityList
    End If
End If
If not(lOk)
    OK message {$init error [iSmtp.$errorcode]: [iSmtp.$errortext]}
    Quit method kFalse
End If
If pRun
    Do iSmtp.$run() Returns lOk
Else
    Do iSmtp.$start() Returns lOk
End If
If not(lOk)
    OK message {$run error [iSmtp.$errorcode]: [iSmtp.$errortext]}
    Quit method kFalse
Else If not(pRun)
    Calculate $cinst.$objs.scrollbox.$objs.cancel.$enabled as kTrue
    Calculate $cinst.$objs.scrollbox.$objs.start.$enabled as kFalse
    Calculate $cinst.$objs.scrollbox.$objs.startthencancel.$enabled as kFalse
End If
Quit method kTrue
```

Callback methods

$completed

The standard $completed callback is passed a row variable parameter with the following columns:

Column	Description
errorCode	An integer error code indicating if the request was successful. Zero means success i.e. the message was successfully sent
errorInfo	A text string providing information about the error if any
log	If you used $protocollog to generate a log, this column contains the log data, either as character data, or UTF-8 HTML. Otherwise, the log column is empty

$mailshotprogress

If the request is a mailshot, and $callmailshotprogress is kTrue, the worker generates a notification to $mailshotprogress each time it sends (or fails to send) the message to a recipient. $mailshotprogress is passed a row variable parameter with the following columns:

Column	Description
address	The email address of the recipient.
sent	Boolean, true if the message was successfully sent to this recipient

FTP Worker Object

The FTPClientWorker provides client FTP support, allowing you to transfer files. The following sections describe the FTP worker properties, constants and methods.

Properties

The FTPClientWorker has the following properties in addition to the base worker properties described earlier:

Property	Description
$requiresecureconnection	If true, and the URI is not a secure URI, the connection starts as a non-secure connection which must be upgraded to a secure connection (using the STARTTLS command). If it cannot be upgraded then the request fails. Defaults to false
$keepconnectionopen	If true, the worker can leave the connection to the server open when it completes its request. Defaults to false. Note that even when this property is set to true, a protocol error may cause the connection to close. Use true if you are likely to use the same server quite soon
$servercharset	The character set used by the server to encode file names in commands and file lists. Default kUniTypeAuto (meaning UTF-8 if the server supports it or kUniTypeNativeCharacters if not). Otherwise a kUniType... constant for 8-bit character sets
$responsepath	If not empty, the worker writes response content to the file with this path rather then adding it to the wResults row. The file must not already exist. The desired value must be set before calling $run or $start. Defaults to empty

Constants

The FTPClientWorker uses the following constants in addition to the base worker constants described earlier. These constants are all actions specified in the iAction parameter used with the $init method to indicate the action to perform, so this section should be read in conjunction with the section describing the $init method:

Constant	Description
kOW3ftpActionPutFile	Upload file data to file cServerPath on FTP server.vParam is file data (binary, character or row).Worker converts character to server character set.Row must have one column (path of file containing data to upload).Note that all file transfers use FTP binary mode
kOW3ftpActionPutFileMulti	Upload multiple files to the FTP server. vParam is a 2 column list (col1: full local pathname, col2: full server pathname). cServerPath must be empty; see below
kOW3ftpActionAppendFile	Identical to kOW3ftpActionPutFile except the action appends the file data to an existing file on the FTP server, or creates a new file containing the supplied data if the file does not exist on the FTP server
kOW3ftpActionGetFile	Download file cServerPath from FTP server. Downloaded file data is either written to $responsepath (if not empty) or returned in the wResults row. vParam is not required.Note that all file transfers use FTP binary mode
kOW3ftpActionGetFileMulti	Download multiple files from the FTP server. vParam is a 2 column list (col1: full local pathname, col2: full server pathname). cServerPath must be empty; see below
kOW3ftpActionDelete	Delete directory or file cServerPath from the FTP server. vParam is Boolean true if cServerPath is a directory, false if it is a file
kOW3ftpActionCreateDirectory	Create directory cServerPath on the FTP server. vParam is not required
kOW3ftpActionListDirectory	List the contents of directory cServerPath on the FTP server returned to wResults.resultList. vParam is Boolean true to list file names only (single column list), or false to get a detailed list with 8 columns: see below
kOW3ftpActionSetPermissions	Set the permissions of file or directory cServerPath on the FTP server. vParam is a character string specifying the new permissions of the file or directory.Note that not all servers support the SITE CHMOD command used by this
kOW3ftpActionExecute	If cServerPath is not empty, CWD cServerPath.Then execute FTP control connection commands in vParam.vParam is either a character string or a single column list of commands. E.g. to rename a file, you could use:RNFR oldname.txtRNTO newname.txtas two lines in the command list.wResults.resultList has a row for each command response
kOW3ftpActionMove	(Studio 11) Moves or renames a file on the FTP server. In this case, cServerPath in $init is the pathname of the file or directory to be moved. vParam is the new server path name. The action works with FTP, FTPS and SFTP (the latter uses a different command), and can be used to rename a file, or move it to a new location

Directory list for kOW3ftpActionListDirectory

FTP does not have a standard syntax for the data returned by the LIST command, so the FTP worker attempts to parse the results of the ListDirectory action, based on some typical syntaxes supported by many servers. The detailed list has 8 columns, as follows:

- The full text returned by the server. This maintains compatibility with previous versions of the OW3 FTP worker, and may contain additional information not extracted by the parser.
- The file name.
- Boolean. True if the entry is probably a directory.
- Boolean. True if the entry is probably a file.

- File size in bytes.
- Modification date of the file.
- Boolean. True if the modification date is in the local time zone of the client. False means the time zone of the modification date is unknown.
- If not empty, the server id of the file or directory. A character string.

Uploading or downloading multiple files

When using kOW3ftpActionGetFileMulti or kOW3ftpActionPutFileMulti, to upload or download multiple files, the row passed to $progress has an extra column (requestNumber) which corresponds to the line number in the vParam list currently being transferred.

The $completed method is called with a successful status if all transfers are completed successfully. If at least one failed, the error code is 10312 (at least one transfer during a kOW3ftpActionGetFileMulti or kOW3ftpActionPutFileMulti action failed). In addition, the resultList column contains a list with a line for each transfer, containing error code, error info and FTP status code.

Methods

FTPClientWorker has the methods described in this section in addition to the base worker methods described earlier. $progress can be called for synchronous (as well as asynchronous) operations for the FTP worker.

Normal methods

$init

$init(cURI, cUser, cPassword, iAction, cServerPath, vParam)

Called to prepare the object to execute a request, before calling $run or $start.

Returns Boolean true for success, or returns false and sets $errorcode and $errortext if an error occurs.

The parameters are:

Parameter	Description
cURI	The URI of the server, optionally including the URI scheme (ftp or ftps) e.g.ftp://ftp.myserver.com.If you omit the URI schemee.g. ftp.myserver.comthe URI scheme defaults to ftp
cUser	The user name to be used to log on to the FTP server
cPassword	The password to be used to log on to the FTP server
iAction	A kOW3ftpAction... constant that specifies the action to perform
cServerPath	A pathname on the FTP server. Paths are relative to the current working directory on the FTP server. The worker only changes directory if you supply a non-empty cServerPath parameter to kOW3ftpActionExecute, so unless you do this, paths are relative to the root.After changing working directory, if you supply cServerPath prefixed with // then the path is relative to the root, e.g./myfile ormyfileis a path relative to the current working directory, whereas//myfileis a path relative to the root.
vParam	A parameter specific to the action. See the constant descriptions for details of vParam for each action

NOTE: If you call $init when a request is already running on a background thread, the object will cancel the running request, and wait for the request to abort before continuing with $init.

Example

You could create an FTP client window with various fields for FTP host name, username, password, timeout setting, server character set, and a list of FTP commands or actions as they are defined in the $init() method. A button could initiate the FTP command, executing the appropriate action depending on the one chosen by the end user, using the following code:

```
# start() method
# iFtp is an Object reference variable with the FTPClientWorker as Subtype
# iActionList (List) variable assigned to list of actions on the window
Do method setupLogging
Calculate iFtp.$timeout as iTimeout ## fields on the FTP window
Calculate iFtp.$callprogress as iCallProgress
Calculate iFtp.$keepconnectionopen as iKeepConnectionOpen
Calculate iFtp.$requiresecureconnection as iRequireSecureConnection
Calculate iFtp.$servercharset as iServerCharsetList.C2
Calculate iFtp.$responsepath as iResponsePath
If iActionList.C2=kOW3ftpActionPutFile.C2=kOW3ftpActionAppendFile
    If iSendContentMode=0
        ReadBinFile (iContentPath,iContent)
        Do iFtp.$init(iServerURI,iUser,iPassword,iActionList.C2,iServerPath,iContent) Returns lOk
    Else
        Do iFtp.$init(iServerURI,iUser,iPassword,iActionList.C2,iServerPath,row(iContentPath)) Returns lOk
    End If
Else If iActionList.C2=kOW3ftpActionSetPermissions
    Do iFtp.$init(iServerURI,iUser,iPassword,iActionList.C2,iServerPath,iPermissions) Returns lOk
Else If iActionList.C2=kOW3ftpActionExecute
    Do iFtp.$init(iServerURI,iUser,iPassword,iActionList.C2,iServerPath,iCommandList) Returns lOk
Else If iActionList.C2=kOW3ftpActionListDirectory
    Do iFtp.$init(iServerURI,iUser,iPassword,iActionList.C2,iServerPath,iNamesOnly) Returns lOk
Else If iActionList.C2=kOW3ftpActionDelete
    Do iFtp.$init(iServerURI,iUser,iPassword,iActionList.C2,iServerPath,iPathIsDirectory) Returns lOk
Else
    Do iFtp.$init(iServerURI,iUser,iPassword,iActionList.C2,iServerPath) Returns lOk
End If
# then $run or $start is called
If not(lOk)
    OK message {$init error [iFtp.$errorcode]: [iFtp.$errortext]}
    Quit method kFalse
End If
If pRun
    Do iFtp.$run() Returns lOk
Else
    Do iFtp.$start() Returns lOk
End If
If not(lOk)
    OK message {$run error [iFtp.$errorcode]: [iFtp.$errortext]}
    Quit method kFalse
Else If not(pRun)
    Calculate $cinst.$objs.scrollbox.$objs.cancel.$enabled as kTrue
    Calculate $cinst.$objs.scrollbox.$objs.start.$enabled as kFalse
    Calculate $cinst.$objs.scrollbox.$objs.startthencancel.$enabled as kFalse
End If
Quit method kTrue
```

Callback methods

$completed

The standard $completed callback is passed a row variable parameter with the following columns:

Column	Description
errorCode	An integer error code indicating if the request was successful. Zero means success i.e. the message was successfully sent
errorInfo	A text string providing information about the error if any
ftpResponseCode	The FTP response code from the last FTP command executed when performing the action. An integer
fileData	Used for kOW3ftpActionGetFile only. If you have not used $responsepath to write the file data directly to a file, this is a binary column containing the file data received from the server
resultList	For kOW3ftpActionList:A single character column list, containing the list entries received from the server.For kOW3ftpActionExecute:A 2 column list, containing an entry for each command supplied in vParam that was successfully executed. Command execution stops as soon as a command fails; the status of the failed command becomes the main error information in the row passed to $completed.Each row of the list contains the ftpResponseCode for the command, and the response text that was received from the server
log	If you used $protocollog to generate a log, this column contains the log data, either as character data, or UTF-8 HTML. Otherwise, the log column is empty

Example

Following on from the $init example above, you could create code in the $completed method to handle the response from the FTP server returned in the pResults parameter: the code writes the log to an HTML file and displays it in the oBrowser object.

```
# $completed method
Calculate iResponse as pResults
Calculate iErrorCode as pResults.errorCode
Calculate iErrorText as pResults.errorInfo
If iUsingLogBrowser
    Do FileOps.$deletefile(iLogHTMLPath)
    WriteBinFile (iLogHTMLPath,iResponse.log)
    Calculate iLogBrowser.$urlorcontrolname as con("file://",replaceall(iLogHTMLPath," ","%20"))
Else
    Calculate iLog as iResponse.log
End If
Do $cinst.$redraw()
Calculate $cinst.$objs.tabpane.$currenttab as 3
```

Secure FTP (SFTP)

The FTP Worker Object supports Secure FTP (SFTP). There are some differences in functionality when using SFTP:

- You use URLs of the form SFTP:// to request an SFTP connection.
- You must explicitly select a server character set - kUniTypeAuto will cause the worker to return an error.
- The append file action is not supported.
- If you have written code that uses the execute action, be aware that SFTP servers support different commands to FTP servers.
- The remaining actions work as expected.

In addition, there are some properties and methods, primarily related to how a connection is authenticated. SFTP does not use TLS, so the secure options related to that only affect FTPS and FTP connections to be upgraded to TLS.

The FTP worker object has the properties:

- **$sshenablecompression**
  If true, SSH compression is enabled for SFTP connections, resulting in a request to the server to enable compression; the server may ignore the request. Defaults to false

- **$sshknownhostsfile**
  The full pathname of the SSH known hosts file used for SFTP. Defaults to the path of clientserver/client/ow3_sftp_known_hosts in the Studio tree. Set this to empty to allow connections (insecurely) to any host

- **$sshknownhostsaction**
  A sum of kOW3sshKHAction… constants (default kOW3sshKHActionReject) specifying what occurs if $sshknownhostsfile is present, and a connection is to be made to a server not in the file, or a server in the file with a host key mismatch

- **$sshauthtypes**
  A sum of kOW3sshAuthType… constants specifying the allowed authentication types when establishing a connection to the server; the default is kOW3sshAuthTypePublicKey + kOW3sshAuthTypePassword + kOW3sshAuthTypeHost + kOW3sshAuthTypeAgent

The FTP worker object has the methods:

- **$getsshoptions()**

  $getsshoptions([&cServerPublicKeyMD5,&cClientPublicKeyFile,&cPrivKeyFile,&cPrivKeyPassword)]) gets the options that affect how SSH connections are established for SFTP

- **$setsshoptions()**

  $setsshoptions([cServerPublicKeyMD5='',cClientPublicKeyFile='',cPrivKeyFile='',cPrivKeyPassword='']) sets the options that affect how SSH connections are established for SFTP. The parameters are:
  cServerPublicKeyMD5:The 128 bit MD5 checksum of the server's public key (supplied as a 32 character ASCII hex string).If not empty,the SFTP client will reject the connection to the server unless the MD5 checksums match
  cClientPublicKeyFile:The pathname of the client's public key. If empty,the client will try to compute the public key from the private key
  cPrivKeyFile:The pathname of the client's private key file
  cPrivKeyPassword:The private key file password

Some or all of the SSH options may be optional, depending on the authentication type chosen.

Keys

When using SFTP, the public and private keys need to be in OpenSSH format. If you have a differently formatted key, such as SSH2 from RFC4716, perhaps generated with PuTTY, you can convert the key using a terminal.

For example, converting a public key with ssh-keygen:

```
ssh-keygen -i -f [path to .pub key]
```

will print out the OpenSSH-format public key which can be copied into its own new .pub file.

Converting a private .ppk key:

```
puttygen [path to .ppk key] -O private-openssh -o [path to new .pem file]
```

will convert the .ppk key generated from PuTTY to a private OpenSSH-format private key.

IMAP Worker Object

The IMAPClientWorker provides client IMAP support, allowing you to use the worker to manage emails stored on an IMAP server. The following sections describe the IMAP worker properties, constants and methods.

Properties

The IMAPClientWorker has the following properties in addition to the base worker properties described earlier:

Property	Description
$requiresecureconnection	If true, and the URI is not a secure URI, the connection starts as a non-secure connection which must be upgraded to a secure connection (using the STARTTLS command). If it cannot be upgraded then the request fails. Defaults to false
$keepconnectionopen	If true, the worker can leave the connection to the server open when it completes its request. Defaults to false.Note that even when this property is set to true, a protocol error may cause the connection to close. Use true if you are likely to use the same server fairly soon
$splitfetchedmessage	If true, the worker splits the fetched message into headers and a MIME list for any content. Defaults to true. If false, the worker simply returns the raw fetched message data
$defaultcharset	Used when kOW3imapActionFetchMessage splits the message, and no character set is specified for a MIME text body part. The character set used to convert to character. Default kUniTypeUTF8. A kUniType... constant (not Character/Auto/Binary)
$removemessageid	If true, the worker removes the Message-id header from the message when performing the action kOW3imapActionAppendMessage. Defaults to true.Duplicating message ids may cause the IMAP server to discard messages with duplicate ids, hence this property
$oauth2	An object reference to an OAUTH2Worker object containing the authorization information required to make requests to the server: see OAUTH2 Worker Object

Constants

The IMAPClientWorker uses the following constants in addition to the base worker constants described earlier. These constants are all actions used with the $init method to indicate the action to perform, so this section should be read in conjunction with the section describing the $init method:

Constant	Description
kOW3imapActionListMailboxes	List mailboxes in reference name cMailboxName.vParam1 specifies the names to list. This becomes the “mailbox name with possible wildcards” parameter of the IMAP LIST or LSUB command (see RFC 3501).vParam2 (optional, default false) is Boolean true to list subscribed mailboxes only
kOW3imapActionListMessages	Selects mailbox cMailboxName and lists the messages it contains.vParam1 (optional), if present, it is a single column list of additional mail header names to retrieve in addition to the standard mailbox list information e.g. the list could have 2 rows, “Subject” and “X-Priority” to retrieve the message subject and priority for each message.vParam2 (optional) is an IMAP search query selecting messages to list, e.g. UNSEEN to fetch the unread messages
kOW3imapActionFetchMessage	Selects mailbox cMailboxName and fetches the message with UID vParam1.vParam2 (optional, default false) is Boolean true to fetch message headers only

Constant	Description
kOW3imapActionSetMessageFlags	Selects mailbox cMailboxName and sets flags for message with UID vParam1.vParam2 is a row of flags with values kFalse, kTrue or kUnknown (leave flag unchanged):row(answered, deleted, draft, flagged, seen)
kOW3imapActionAppendMessage	Selects mailbox cMailboxName and appends a message to the mailbox.You can either:Supply the entire message as binary data in vParam1 orSupply a 2 character column list in vParam1 (columns are header name and header value) with binary raw content in vParam2orSupply a 2 character column list in vParam1 (columns are header name and header value) with the content specified by a MIME list in vParam2. See the documentation for the MailSplit command to see how a MIME list is structured; however note that the charset in the worker MIME list is a kUniType... constant rather than a character string
kOW3imapActionExecute	If cMailboxName is not empty select mailbox cMailboxName.Then execute IMAP commands in vParam1. vParam1 is either a binary value or a single column list of binary values. wResults.resultList has a row for each command response.Each binary value is an IMAP command to execute, e.g. EXAMINE. You can generate binary values using the correct character set required by the IMAP protocol using the $chartoutf7 method of the IMAPClientWorker.The sequence of actions will stop as soon as an error occurs
kOW3imapActionSelect	(Studio 11) Executes an IMAP SELECT on the mailbox given to $init. As part of the SELECT, IMAP returns the number of emails in the mailbox, which are returned to the $completed method in resultList in column EXISTS. Note that an IMAP SELECT will cause the current mailbox to be changed, so you may prefer to execute this action on a different connection

Methods

IMAPClientWorker has the methods described in this section in addition to the base worker methods described earlier.

Normal methods

$init

```
$init(cURI, cUser, cPassword, iAction, cMailboxName, vParam1, vParam2)
```

Called to prepare the object to execute a request, before calling $run or $start.
Returns Boolean true for success, or returns false and sets $errorcode and $errortext if an error occurs.

The parameters are:

Parameter	Description
cURI	The URI of the server, optionally including the URI scheme (imap or imaps), e.g. imap://ftp.myserver.com. If you omit the URI scheme, e.g. imap.myserver.com, the URI scheme defaults to map
cUser	The user name to be used to log on to the FTP server
cPassword	The password to be used to log on to the FTP server
iAction	A kOW3imapAction... constant that specifies the action to perform

Parameter	Description
cMailboxName	The IMAP mailbox name (ignored for kOWEimapActionListMailboxes). If you are using non-ASCII characters in mailbox names, you may need to normalise the name using the Omnis nfd() or nfc() function before passing it to $init()
vParam1	A parameter specific to the action. See the constant descriptions for details of vParam1 for each action
vParam2	A parameter specific to the action. See the constant descriptions for details of vParam2 for each action

NOTE: If you call $init when a request is already running on a background thread, the object will cancel the running request, and wait for the request to abort before continuing with $init.

$chartoutf7

```
$chartoutf7(cChar)
```

Returns a binary value (containing 7 bit characters) that is the IMAP UTF-7 representation of cChar (note that IMAP uses a special variant of UTF-7, and this method generates that variant).

The parameters are:

Parameter	Description
cChar	A character string to be converted to IMAP UTF-7

$utf7tochar

```
$utf7tochar(xUtf7[,bAllowCRLF=kTrue])
```

Converts IMAP UTF-7 xUtf7 to character and returns the result. Optionally allows CRLF sequences in the data and replaces them with CR in the result (note that IMAP uses a special variant of UTF-7, and this method expects that variant in xUtf7).

The parameters are:

Parameter	Description
xUtf7	A binary value containing IMAP UTF-7 to be converted to character.
bAllowCRLF	If true, CRLF sequences are to be expected in the UTF-7 stream - they are replaced with CR.

Callback methods

$completed

The standard $completed callback is passed a row variable parameter with the following columns:

Column	Description
errorCode	An integer error code indicating if the request was successful. Zero means success i.e. the message was successfully sent
errorInfo	A text string providing information about the error if any
resultList	This column receives a list, the content of which depends on the action. The action-specific lists returned here are described below (actions kOW3imapActionSetMessageFlags and kOW3imapActionAppendMessage do not return any data in this column)

Column	Description
log	If you used $protocollog to generate a log, this column contains the log data, either as character data, or UTF-8 HTML. Otherwise, the log column is empty
<action-specific>	Column 5 is present for certain actions, and contains action- specific data. The action-specific data is described below

kOW3imapActionListMailboxes:

Column	Description
resultList	The list of mailboxes that match the criteria pass to $init(). A 7 column list, with columns as follows (see RFC 3501 for more details - the data in these columns is populated using the LIST or LSUB response):**hasChildren:** Boolean true if the mailbox has child mailboxes.**noInferiors:** Boolean true if it is not possible for any child levels of hierarchy to exist under this name; no child levels exist now and none can be created in the future.**noSelect:** Boolean true if it is not possible to use this name as a selectable mailbox.**marked:** Boolean true if the mailbox has been marked “interesting” by the server; the mailbox probably contains messages that have been added since the last time the mailbox was selected.**unMarked:** Boolean true if the mailbox does not contain any additional messages since the last time the mailbox was selected.**separator:** The mailbox hierarchy delimiter.**mailboxName:** The name of the mailbox
<action-specific>	No action specific column

kOW3imapActionListMessages:

Column	Description
resultList	The list of messages in the mailbox. This is a list with 9 standard columns, followed by a column for each header specified in vParam2 when calling $init() to prepare for this action. The additional header columns are named by removing - characters from the header name, and converting the result to lower case.The 9 standard columns in the message list are:**UID:** The unsigned integer UID of the message**size:** The size of the message in bytes**internalDate:** The internal date of the message.**answered:** The Boolean answered flag for the message.**deleted:** The Boolean deleted flag for the message.**draft:** The Boolean draft flag for the message.**flagged:** The Boolean flagged flag for the message.**recent:** The Boolean recent flag for the message.**seen:** The Boolean seen flag for the message.
<action-specific>	No action specific column

kOW3imapActionFetchMessage:

Column	Description
resultList	If $splitfetchedmessage is kFalse, this column is not populated. Otherwise, this column is a 2 character column list of mail headers:Column 1 is the header name.Column 2 is the header value

Column	Description
<action-specific>	If the action fetches the message content as well as the headers, this column receives the content. It is either:**rawData:** A binary column that receives the un-split fetched message dataor**mimeList:** A MIME list containing the content. See the documentation for the MailSplit command to see how a MIME list is structured; however note that the charset in the worker MIME list is a kUniType… constant rather than a character string

kOW3imapActionSetMessageFlags:

Column	Description
resultList	Not populated.
<action-specific>	No action specific column.

kOW3imapActionAppendMessage:

Column	Description
resultList	Not populated
<action-specific>	If possible, the action extracts the UID of the appended message from the IMAP server response. The UID is returned to this column (named UID) and is non-zero if the UID could be extracted. (Note that not all servers return the UID of an appended message)

kOW3imapActionExecute:

Column	Description
resultList	A single column list of binary values. Each row of the list contains the sequence of responses returned from the server when executing the corresponding command in the list (or single binary value) passed to $init(). You would typically decode this using $utf7tochar, using the option to expect CRLF and replace with CR
<action-specific>	No action specific column

JavaScript Worker Object

The **JavaScript Worker Object** allows you to execute JavaScript methods inside **node.js** by making the request in Omnis code by calling a Worker Object method, and receiving the results via a worker callback. The node.js framework is embedded into Omnis Studio, and contains many open source third-party modules that can be used from inside your Omnis code. All traffic sent between the Omnis and node.js processes is encrypted. For example, the library 'xml2js' is included in Omnis Studio, which converts XML to JSON: in addition, support for ZIP can be added using the node.js jszip module, which is described at the end of this section.

npm is provided alongside Node.js. To launch npm you can run index.js inside the npm folder, e.g.

```
./node npm/index.js
```

Enabling JavaScript Methods

There is a JS file, ow3javascript.js, located in the clientserver/server/remotedebug program folder, that is the entry point for all method calls: on macOS, the remotedebug folder is in the Resources folder in the Omnis.app bundle. Method calls arrive as an HTTP request from Omnis, and respond with their results as HTTP content. A worker executes methods sequentially.

Omnis has a simple structure where you can write a JavaScript module containing one or more methods, and then call methods via their module and method name.

In the Omnis data folder, there is a folder called 'jsworker', where modules required by ow3javascript.js are located. You can install additional node.js modules in this folder using the npm -i command when running in the folder - these might be modules for which you want to provide an interface from Omnis.

There are two key files in this folder, which must always be present:

omnis_calls.js - a module which provides an interface for methods to return their results to Omnis.

omnis_modules.js - a module which provides a table of modules that can be called from Omnis.

There are also two example module files, omnis_test.js and omnis_xml2js.js. These provide Omnis modules named test and xml2js. Each module file must have an entry in omnis_modules.js. Each module file provides a table of methods that can be called from Omnis.

Note that 'jsworker' in the data folder may not be considered suitable for deployment, since the data folder is writeable. The worker provides the ability for you to structure things differently when you deploy your application, but is fine for development.

Auto Loading modules

From Studio 11, the JS Worker will pick up any modules you have added automatically if they are placed in the jsworker folder. Any modules that you have added in their own folder, with a package.json or index.js, are picked up automatically by the JS Worker. (This is in addition to using the hard-coded moduleMap method, as described below.)

Creating the worker

The sub-type of the external object is OW3 Worker Objects\JAVASCRIPTWorker. You can use either an Object variable or an Object Reference variable, either directly if you set $callbackinst to receive results, or by subclassing the external object with an Omnis object.

Properties

The JavaScript worker only has the standard worker properties: $state, $threadcount, $errorcode and $errortext.

Methods

Called Methods

$init()

```
$init([cPath, bDebugNodeJs=kFalse])
```

Initialize the object so it is ready to execute JavaScript method calls. Returns true if successful. You must call $init() before any other methods.

- **cPath**
  allows you to override the default NODE_PATH module search path set by the worker. The default path is <Omnis data folder>/jsworker. If you override this path, the various JavaScript modules that are mandatory for the worker to operate must still be able to be located.
- **bDebugNodeJs**
  is a Boolean that indicates if you want to be able to debug node.js, for example using Chrome. It is possible that you may not be able to start the worker if you set this for more than one active JavaScript worker, as node.js requires a debug port to be available. To debug the JavaScript in Chrome, navigate to chrome://inspect, and then open the dedicated debug tools for node.js via the link.

$start()

```
$start()
```

Runs the JavaScript worker in a background thread. Returns true if the worker was successfully started.

After you call $start(), the background thread starts up a node.js process which will process JavaScript method calls. You can make multiple method calls to the same process, so there is no need to call $start() frequently, which means the overhead of starting the node.js process is minimal.

$cancel()

```
$cancel()
```

Use this to terminate the node.js process. Any in-progress method calls may not complete.

$callmethod()

```
$callmethod(cModule, cMethod, vListOrRow [,bWait=kFalse, &cErrorText, vTag])
```

Call the method passing it a single parameter which is the JavaScript object representation of vListOrRow. Optionally wait for the method to complete. Returns true if successful.

cModule and cMethod (Character) identify a module, and a method within the module, to call, as described above.

vListOrRow (Variant) will be converted to JSON and passed to the method as its parameter. This means that you should be aware of data that will not map to JSON, and avoid trying to pass that to $callmethod.

bWait (Boolean) indicates if the caller wishes to suspend execution until the method completes. If you use bWait, then a completion callback will occur before $callmethod returns.

cErrorText (Character) receives text describing the error if $callmethod fails; code 460 for module not found, and 461 for method not found.

(for Studio 11) vTag (Variant) If supplied, some data can be passed to $methoderror or $methodreturn in the column __tag of the row parameter. This can be used, for example, to identify the caller when the worker object is shared by several instances.

Callback Methods

$cancelled

Override this to receive a notification that the request to cancel the worker has succeeded.

$workererror

$workererror(wError)

Override this method to receive reports of errors from the worker that are not related to calling a method, e.g. failure to start node.js. The worker thread exits after generating this notification.

wError has two columns, an Integer named errorCode and a Character string named errorInfo.

$methoderror

$methoderror(wError)

Override this method to receive reports of the failure of an attempt to call a method with $callmethod.

wError has two columns, an Integer named errorCode and a Character string named errorInfo.

If an unhandled exception occurs causing node.js to exit, Omnis adds the stack traceback of the exception to the errorInfo column.

$methodreturn

$methodreturn(wReturn)

Method called with the results of a call to $callmethod.

wReturn is a list/row parameter with two columns: __module and __method. If the parameter is a list, then __module and __method are only populated for the first line of the list.

If the JavaScript method returns an object, this is the Omnis equivalent of the object, created by converting the JSON to a row. If the JavaScript method returns some other data, e.g. a picture, this is a row with a single column named content, which contains the data returned by the method.

The content column for non-JSON content returned from a worker method is of type character when the content type is text/.

Example: Adding ZIP support

You could add support for ZIP files. To do this, install the node.js jszip module by running the npm command in the jsworker folder:

```
npm i jszip
```

(the npm command is installed with node.js, available on the web)

Edit omnis_modules.js by adding an entry to the 'moduleMap' object for the zip module (or from Studio 11 it should be loaded automatically since it is in its own folder 'omnis_zip'):

```
const moduleMap = {
  zip: require('./omnis_zip.js'),
  ... // Other modules
};
```

You can then make calls from Omnis code as follows:

```
Do lRow.$cols.$add("path",kCharacter,kSimplechar)
Calculate lRow.path as iZipPath
Do iJS.$callmethod("zip","loadZip",lRow,kTrue,lErrorText) Returns lOK
```

POP3 Worker Object

The POP3 worker is similar to other OW3 workers, in that you pass an action to $init and action specific parameters, and use $run or $start to execute the request. There is a sample app in **Samples** section in the **Hub** in the Studio Browser.

Methods

$init()

```
$init(cURI,cUser,cPassword,iAction[,iMessageNumber,cPostCommand])
```

Initialises the object so it is ready to perform the specified action using POP3. Returns true if successful

The $init parameters are:

Parameter	Description
cURI	The URI of the server, optionally including the URI scheme (pop3 or pop3s) e.g. pop3://pop3.myserver.com. If you omit the URI scheme, e.g. pop3.myserver.com, the URI scheme defaults to pop3
cUser	The username to be used to log on to the POP3 server
cPassword	The password to be used to log on to the POP3 server

Parameter	Description
iAction	A kOW3pop3Action... constant that specifies the action to perform; see below
iMessageNumber	The message number to get (applies to actions kOW3pop3ActionGetMessage, kOW3pop3ActionGetHeaders and kOW3pop3ActionDeleteMessage)
cPostCommand	Only applies to action kOW3pop3ActionGetMessage. If not empty, a command to send to the server after getting the message. This would typically be RSET to undelete messages or QUIT to delete messages

The Actions are:

- **kOW3pop3ActionStat** Gets maildrop status. For a successful request, wResults has 2 columns returning the stat information, messageCount and maildropSize
- **kOW3pop3ActionList** Lists messages. For a successful request, wResults has a column named messageList which contains a 2 column list, with columns messageNumber and messageSize
- **kOW3pop3ActionGetMessage** Gets specified message. For a successful request, wResults has either a column rawData or columns headerList and mimeList (depending on the value of $splitfetchedmessage
- **kOW3pop3ActionGetHeaders** Gets headers for a specified message. For a successful request, wResults has either a column rawData or a column headerList (depending on the value of $splitfetchedmessage)
- **kOW3pop3ActionDeleteMessage** Deletes the specified message from the maildrop
- **kOW3pop3ActionQuit** Sends a QUIT command to the server to ensure that any messages marked for deletion are deleted

Properties

The OW3 POP3 worker has the following properties in addition to those supported by all OW3 worker objects:

Property	Description
$splitfetchedmessage	If true, worker splits fetched message into headers and MIME list for any content. Defaults to true. If false, the worker simply returns the raw fetched message data (Applies to kOW3imapActionFetchMessage and kOW3pop3ActionGetMessage)
$defaultcharset	Used by kOW3imapActionFetchMessage and kOW3pop3ActionGetMessage when there is no charset for a MIME text body part. The charset used to convert to character. Default kUniTypeUTF8.A kUniType... constant (not Character/Auto/Binary)
$keepconnectionopen	If true, the worker can leave the connection to the server open when it completes its request. Defaults to false. Note that even when this property is set to true, a protocol error may cause the connection to close.
$requiresecureconnection	If true, and the URI is not a secure URI, the connection starts as a non-secure connection which must be upgraded to a secure connection (using STARTTLS or STLS). If it cannot be upgraded, then the request fails. Defaults to false
$oauth2	An object reference to an OAUTH2Worker object containing the authorization information required to make requests to the server: see OAUTH2 Worker Object

CRYPTO Worker Object

The CRYPTO Worker Object allows you to perform encryption and decryption of data. The encryption types you can use include AES, Camellia, DES, and Blowfish.

The CRYPTO worker is similar to other OW3 workers, in that you pass an action to $init and action specific parameters, and use $run or $start to execute the request. There is an example app in the **HUB** in the Studio Browser to demo the CRYPTO Worker Object.

To use the worker object, create a variable with its subtype set to the CRYPTOWorker object which is contained in the OW3 Worker Objects group in the **Select Object** dialog, or subclass the external object. For encryption of data over 2GB you are advised to write the encrypted data to a file, rather than memory.

The CRYPTO worker has the following methods:

- **$start, $run and $cancel**
  Standard worker methods
- **$completed** and **$cancelled**
  Standard worker completion methods
- **$makerandom**
  $makerandom(iBytes,&xRandomData) generates some random data with the specified length in bytes. This is suitable for use as an encryption key or initialization vector. Returns true if successful (if an error occurs, the standard properties $errorcode and $errortext describe the error).
  iBytes is the number of bytes of random data to generate
  xRandomData is a binary variable that receives the random data

As with the other worker objects you can use the $init method with various input parameters to initialize the object, while the action can be to Encrypt or Decrypt:

- **$init**
  $init(iAction, iEncryptionType, iCipherMode, iPadding, xKey, xIV, vInputData [,cOutputPath]) initialises the object ready to perform the encryption or decryption. Returns true if successful

iAction can be either:

- kOW3cryptoActionEncrypt
  Encrypt the data using the specified encryption scheme and parameters
- kOW3cryptoActionDecrypt
  Decrypt the data using the specified encryption scheme and parameters

iEncryptionType can be one of:

- kOW3cryptoTypeAES
  AES encryption. Key size must be 128, 192 or 256 bits
- kOW3cryptoTypeCamellia
  Camellia encryption. Key size must be 128, 192 or 256 bits
- kOW3cryptoTypeDES
  DES encryption. Key size must be 64 or 192 bits. Uses Triple DES if key size is 192 bits
- kOW3cryptoTypeBlowfish (deprecated in Studio 11; do not use for new apps)
  Blowfish encryption. Key size must be 128 bits

iCipherMode can be one of:

- kOW3cryptoCipherModeCBC
  CBC (Cipher Block Chaining)

- kOW3cryptoCipherModeECB
  EBC (Electronic Code Book)
- kOW3cryptoCipherModeCTR
  CTR (Counter). Not supported for kOW3cryptoTypeDES

iPadding can be one of:

- kOW3cryptoPaddingNone
  No padding (use this for cipher modes other than CBC and EBC)
- kOW3cryptoPaddingPKCS7
  PKCS7 padding
- kOW3cryptoPaddingOneAndZeros
  One and zeros padding (ISO/IEC 7816-4)
- kOW3cryptoPaddingZerosAndLen
  Pad with N-1 zero bytes followed by a byte with value N, where N is the number of padding bytes (ANSI X.923)
- kOW3cryptoPaddingZeros
  Pad with N zero bytes, where N is the number of padding bytes

Note that PKCS7 is the most common and allows binary data to be correctly decrypted, for example, kOW3cryptoPaddingZeros can lead to extra bytes being stripped from decrypted binary data.

xKey is a binary containing the key. See the encryption types for details of supported key lengths.

xIV is a binary containing the Initialization Vector (IV) (random data that can be used to make the same encrypted data different when using the same key). This must be 8 bytes long for DES and Blowfish, and 16 bytes long for AES and Camellia.

vInputData is the data to be encrypted. Either a character value which is the pathname of the file containing the data to encrypt or decrypt, or a binary variable containing the data to encrypt or decrypt.

cOutputPath is:

- Either omitted or empty meaning that the encrypted or decrypted data is supplied to $completed in the data column of the results row parameter (this column has type binary)
- Or the pathname of a file (which must not already exist) to which the worker will write the encrypted or decrypted data

The results row passed to $completed has 3 columns: errorCode, errorInfo and data. The error columns behave in the same way as the other OW3 workers: note that a negative error code is a code returned by the mbedTLS library. The data column is only used when cOutputPath was omitted when calling $init.

The worker has properties as follows:

- $errorcode, $errortext, $state and $threadcount
  Standard worker properties
- $useexplicitiv
  If true, a random IV is added to the start of the data when encrypting or the first IV size bytes is removed from decrypted data. Only applies to CBC cipher mode. This means that a user can decrypt data without knowing the IV (any IV can be used for decryption, which will result in just the first IV size bytes being decrypted incorrectly, because of the way the CBC cipher mode works)

HASH Worker Object

The HASH Worker Object allows you to hash data using SHA1, SHA2, SHA3, MD5, and RIPEMD hash types, which are primarily for signature purposes, while PBKDF2 is available for password hashing. You use the $inithash() method to initialise the object, passing the binary or character data to be hashed, and the hash type represented by a constant, as follows:

Constant	Hash type
kOW3hashSHA1	SHA1
kOW3hashSHA2_256	SHA2 including hash output 256
kOW3hashSHA2_384	SHA2 including hash output 384
kOW3hashSHA2_512	SHA2 including hash output 512
kOW3hashSHA3_256	SHA3 including hash output 256
kOW3hashSHA3_384	SHA3 including hash output 384
kOW3hashSHA3_512	SHA3 including hash output 512
kOW3hashMD5	MD5
kOW3hashRIPEMD_160	RIPEMD_160
kOW3hashPBKDF2	PBKDF2

You can use $initverifyhash to verify or compare some data against previously hashed data generated using $inithash. Having called the $init... methods, you can use $run or $start to execute the request.

To use the worker object, create a variable with its subtype set to the HASHWorker object which is contained in the OW3 Worker Objects group in the Select Object dialog, or subclass the external object. There is a new sample app in the HUB in the Studio Browser to demo the HASH Worker Object.

The HASH worker object has methods as follows:

- **$start, $run** and **$cancel**
  Standard worker methods: see Base Worker Methods
- **$completed** and **$cancelled**
  Standard worker completion methods: see Callback methods
- **$inithash()**
  $inithash(vData,iHashType,vHashParameters) initialises the object so it is ready to hash data using the specified hash type
  vData specifies the data to hash. Either binary or character. A binary value is used directly. Otherwise, for PBKDF2 the worker converts character data to UTF-8 before generating the hash. For other hash types, a character parameter is the pathname of the file containing the data to hash.
  iHashType the hash type, a kOW3hash... constant.
  vHashParameters A row of parameters that control the hash. For all types except PBKDF2, an empty row() to generate a hash, or the binary key to use when generating an HMAC (see below). For PBKDF2, a row with 3 integer columns in the order saltLength, keyLength, iterations.
  saltLength - the length of the random salt (generated by the worker). A good value for this is 16. Must be 8 to 64 inclusive
  keyLength - the length of the hashed key to be generated. A good value is 32. Must be between 16 and 256 inclusive
  iterations - the number of iterations to perform to generate the hash. A good value for iterations is at least 100000 - the higher the value, the more secure the hash, traded off against a longer execution time. Must be between 1 and 256000 inclusive
- **$initverifyhash()**
  $initverifyhash(vData,iHashType,vHash [,xHMACkey]) initialises the object so it is ready to generate the hash for vData using iHashType and compare it against previously generated vHash
  vData specifies the data to hash. Either binary or character. For PBKDF2 the worker converts character data to UTF-8 before generating the hash. For other hash types, a character parameter is the pathname of the file containing the data to hash.
  iHashType the hash type, a kOW3hash... constant.
  vHash A hash previously generated by the worker using $inithash(). Either binary or character. If character, the value must be the BASE64 encoded representation of the hash. Using this method, you could create a hash using $inithash and store that for future use. To verify a password or document you call $initverifyhash, with vData as the password or document to verify, and vHash as the stored hash from your call to $inithash.
  xHMACkey the key to use when verifying an HMAC authentication code (see below)
- **$initsignature()**
  $initsignature(vData,iHashType,vPrivateKeyPEM[,bBlind=kFalse]) initialises the object so it is ready to generate the signature for vData using iHashType and RSA encryption with private key xPrivateKeyPEM.
  vData specifies the data for which a signature is required. Either binary or character. The worker converts character data to UTF-8 before generating the signature.
  iHashType the hash type, a kOW3hash... constant.
  vPrivateKeyPEM The private key in PEM (Privacy Enhanced Mail) format. Either binary or character. Binary is assumed to already be UTF-8. The worker converts character data to UTF-8 before trying to use the private key.
  bBlind (default kFalse) If true, the RSA encryption used to generate the signature uses blinding.

- **$initverifysignature()**
  $initverifysignature(vData,iHashType,vPublicKeyPEM,vSignature) verifies a signature from $initsignature.
  vData the original data
  iHashType the original hash type
  vPublicKeyPEM the public key in PEM format
  vSignature the signature from $initsignature
  When returning to $completed, the row's errorCode will be 0 if the signature has been verified successfully (that is, the data has not been tampered with and it matches the signature), otherwise it will have a mbedtls or an Omnis error code if something has gone wrong (e.g. if the signature doesn't match, it should return -17280 with error info of "RSA - The PKCS#1 verification failed").

HMACs

HMACs can be generated for all hash types, except PBKDF2 and the SHA3 hashes, instead of a hash. The key length for an HMAC is unrestricted (in versions prior to Studio 10.22 key length was restricted to 64 characters).

To generate an HMAC rather than a hash, supply the binary key as the hash parameters parameter of $inithash() – an empty row as this parameter generates a hash rather than HMAC. To verify an HMAC rather than a hash, supply the binary key as a new binary last parameter to $initverifyhash() – this is optional and its presence indicates HMAC.

Results

After calling one of the $init... methods, you call $run or $start. The results row passed to $completed has 3 columns: errorCode, errorInfo and data. The error columns behave in the same way as the other OW3 workers - note that a negative error code is a code returned by the mbedTLS library. The data column is only used for $inithash() and it contains the generated binary hash, provided that no error occurred – you may want to convert this to base64 before storing it, but note that if you do this you will need to convert it back to binary before using it to verify data. For $initverifyhash(), the errorCode is zero if and only if the hash was successfully verified.

The worker has properties as follows:

- $errorcode, $errortext, $state and $threadcount
  Standard worker properties

LDAP Worker Object

Lightweight Directory Access Protocol (LDAP) allows "the sharing of information about users, systems, networks, services, and applications throughout the network" (Wikipedia). The **LDAP Worker** is available in the OW3 group of worker objects, and is named LDAPClientWorker.

To use the LDAP Worker, you should create an Object variable or an Object Reference variable and set its subtype to the LDAPClientWorker object in the Select Object dialog; or you can create an Object class, set its superclass to the LDAPClientWorker object, create an Object variable or an Object Reference variable and set its subtype to the object class.

There is an example library which you can request from your local Support office.

There is an example app called LDAP Client Worker Object in the Samples section in the Hub in the Studio Browser

Properties

The LDAP Worker has all the base worker properties plus $callbackinst and $keepconnectionopen.

Methods

The LDAP Worker has all the base worker methods. You can use the $init method to initialize the worker object to specify what information is to be returned from the LDAP server; $init for the LDAP Worker has the following definition:

- $init(*cURI,cUser,cPassword*) Initialize the object so it is ready to access the specified URI using LDAP. Returns true if successful
  cURI: The URI of the server, optionally including the URI scheme (ldap or ldaps) e.g. ldap://ldap.myserver.com. If you omit the URI scheme e.g. ldap.myserver.com, the URI scheme defaults to ldap
  cUser: The user name to be used to log on to the LDAP server
  cPassword: The password to be used to log on to the LDAP server

To structure the URI of the LDAP server, you will need to find out what parameters are required by the server; you may find the RFC4516 standard useful which defines the syntax of LDAP URLs: https://docs.ldap.com/specs/rfc4516.txt.

As $init() is called, the details of what is to be retrieved from the server are passed in the URL. You can then call $run() or $start() to run the request on the main thread or a background thread.

When the query completes, the worker calls the $completed method in the usual way for OW3 workers. The row passed to $completed has four columns:

errorCode: Zero for success, otherwise an error code

errorInfo: The description of the error

log: If you enabled logging for the OW3 worker, this contains the log

rawData: If successful, the result of the LDAP query. The developer is currently responsible for parsing this.

Python Worker Object

The Python Worker Object works exactly like the JavaScript worker, including support for HTTP/2, with a few exceptions, as follows.

The $init method has only one parameter instead of two, since Omnis does not support the remote debugger capabilities in Python.

The $init method of the Python worker accepts a second optional parameter, cExecutable, which is a character variable containing the path to the Python executable to use, overriding the default location of the Python executable. Without the parameter the Python worker looks in /usr/bin/python3 for the Python executable.

When passing rows to the Python worker, a dictionary object is created in your Python module. When passing lists, a list object is created.

There is an example library which you can request from your local Support office.

There is an example library for the Python Worker under the Samples section in the Hub in the Studio Browser.

Installation

The Python executable is not provided with Omnis Studio, so you will have to install it manually alongside pip (the package manager) on your chosen platform. The Python worker will work with python3 only and ideally you should use at least Python 3.6. Get the latest download and installation notes for different platforms from: https://www.python.org/

On Linux and macOS, Omnis expects the binary to be in /usr/bin/python3 as well as on the PATH. On Windows, Omnis expects the installation to also be in the PATH as it relies on loading the python3.dll to get the directory where Python is installed and use the python.exe within. Currently, you cannot specify a path to a different python executable to use.

In addition, flask, psutils and requests are required; these are listed in a file called requirements.txt which is in the pyworker folder in the Omnis read/write directory. You can either install them manually or by doing pip/pip3 install -r path/to/file/requirements.txt

In order to create a Python module, create a new folder inside the pyworker folder in the Omnis read/write directory, and include a main.py file which Omnis will load at runtime. When calling your module, use the folder name of your module and a function within your main.py. You can import omnis_calls in your main.py and use sendResponse or sendError if required, or you can simply return a message or some data, or raise an Exception if an error occurred (in which case it should automatically return a $methoderror or $methodreturn).

Java Worker Object

The **Java Worker** allows you to invoke methods inside Java modules. There is a **Java Worker** example in the **Samples** section in the **Hub** in the Studio Browser.

Installation

Java is not installed with Omnis Studio, therefore you will have to install it separately. Java version 17 was used to implement the Java Worker, more specifically the Adoptium implementation, but any Java implementation (including Oracle's) should work with the Omnis Java Worker.

Methods

The **$init** method initializes the Java Worker and is the only method that differs from the other existing worker base methods.

```
$init([cPath,cExecutable,bDebugWorker,cJvmOptions])
```

Initializes the Java Worker and returns kTrue if successful, otherwise kFalse is returned.

- **cPath**
  optional character string of items for the Java CLASSPATH. Paths must be separated by ; (semi colon) on Windows, and : (colon) on macOS and Linux. If empty, the worker adds the default CLASSPATH.
- **cExecutable**
  optional character string containing the path to the Java executable that should be used. If empty, Omnis uses /usr/bin/java on macOS and Linux, and java.exe on Windows.
- **bDebugWorker**
  optional boolean to put the Java Worker thread in debug mode, defaults to false. If debug is true, debugging messages are written to stdout.
- **JvmOptions**
  optional character string of options to pass to the JVM.

Properties

The Java Worker has the same base properties as the existing workers.

Main module file structure

A new **javaworker** folder has been added to the clientserver/server folder in the Omnis read/write location (Application Support on macOS or AppData on Windows):

```
<read/write location>/clientserver/server/javaworker
```

which contains:

- **JavaWorker.jar**
  the main Java entry point for the Java Worker. This is a Spring Boot application running the http/2 endpoint that handles Java method calls.
- **lib**
  a folder containing the JAR dependencies for JavaWorker.jar.
- **store.jks**
  a Java KeyStore containing the Omnis self-signed certificate used to form a secure connection between Omnis and the JavaWorker.jar. When Omnis starts the Java process, the JavaWorker.jar, lib subfolder and <read/write location>/clientserver/server/javaworker/ are added to the CLASSPATH.

User module file structure

A new **javaworker** folder has been added in the root of the Omnis read/write location (Application Support on macOS or AppData on Windows) which contains:

- **lib**
  a global folder for JAR dependencies, for example, it contains gson-2.10.1.jar since it is used by some of the JARs in the Java Worker. If your modules have a common dependency, you should use this global lib folder.
- **OmnisCalls** a folder that represents the OmnisCalls module, containing the OmnisCalls.jar.
- **OmnisTest**
  a folder that represents the OmnisTest module, containing the OmnisTest.jar.

Each new module needs to be placed inside its own folder, inside the javaworker folder, at the root of the read/write location. If your module has dependencies that are not shared with other modules, you should also create a lib folder inside your module's folder and place the dependencies there.

You can place common module dependencies in the lib folder at the root of the javaworker folder.

When Omnis starts the Java process, each module folder and its lib subfolder are added to the CLASSPATH, alongside the main lib folder. This means that your module's lib subfolder contents will be visible to other modules, therefore the separation is purely for maintaining a clean file structure.

Building Java Modules

When creating a new Java module to use with the Java Worker, you will need to first create a Java project. Visual Studio has a great extension that assists with this, see more here:

https://code.visualstudio.com/docs/java/java-tutorial

The **Maven** build tool was used to create the Java Worker. More specifically, the Maven wrapper (https://maven.apache.org/wrapper/) which provides a mvnw executable.

Once your project is set up, you will need to add **OmnisCalls** as a dependency. In the Maven project, OmnisCalls was added to the **pom.xml** in the section:

```
...
   <dependencies>
      <dependency>
         <groupId>net.omnis</groupId>
         <artifactId>OmnisCalls</artifactId>
         <version>0.1</version>
         <scope>system</scope>
         <systemPath>path/to/OmnisCalls.jar</systemPath>
      </dependency>
   </dependencies>
...
```

The dependency is needed only at build time since Omnis comes with OmnisCalls.jar in the CLASSPATH at runtime. Whilst editing the pom.xml, if using Maven, you could also setup the Maven plugin to build your dependencies out in a lib folder inside your project, which you could then copy out into your module's lib subfolder:

```
...
   <build>
      <plugins>
         <!-- Get dependencies in lib folder -->
         <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-dependency-plugin</artifactId>
            <version>3.2.0</version>
```

```
            <executions>
               <execution>
                  <id>copy-dependencies</id>
                  <phase>package</phase>
                  <goals>
                     <goal>copy-dependencies</goal>
                  </goals>
                  <configuration>
                     <outputDirectory>
                        ${project.build.directory}/lib
                     </outputDirectory>
                  </configuration>
               </execution>
            </executions>
         </plugin>
      </plugins>
   </build>
...
```

Once the dependency is available to your Java project, you can create the Java class you wish to expose to Omnis by extending OModule, for example:

```
   package net.omnis.OmnisTest; import net.omnis.OmnisCalls.*;
   public class Test extends OModule {
      // Implement your methods here
   }
```

In the above example a package net.omnis.OmnisTest was created with a class Test. When using **$callmethod** from Omnis Studio, the module name parameter would be net.omnis.OmnisTest.Test.

Note that if you do not inherit from the OModule class, the Java Worker will not be able to call your methods; the OModule class does not do anything at present, but is reserved for future use.

To implement a method in your module, you must make it return a Response object (provided by OmnisCalls) and take in a Map:

```
   public Response test(Map<String, Object> pParams) {

      Map<String, Object> data = new HashMap<>(); data.put("unicode",
         "Fingerspitzengef\u00FChl is a German term.\nIt\u2019s pronounced as follows:       [\u02C8f\u026A\u0
\u02D0l]");

      return new SendResponse(data);
   }
```

In the above example a test method was created which returns a new SendResponse object (provided by OmnisCalls) with a Map parameter (i.e. a key-value store) containing one key: 'unicode' with a UTF-8 formatted string value.

The Java Worker automatically converts the Map from SendResponse into JSON, which when received in Omnis Studio will be a row variable in your $methodreturn.

Similarly, if you wish to send back an error, you can return a new SendError (provided by OmnisCalls). If you wish to send a list as part of the data, you can add an ArrayList to the Map, which can be summarized with the OmnisTest example, as follows:

```
   package net.omnis.OmnisTest;
   import net.omnis.OmnisCalls.*;

   import java.util.Map;
   import java.util.HashMap;
   import java.util.ArrayList;
```

```
public class Test extends OModule {

    public Response test(Map<String, Object> pParams) {

        Map<String, Object> data = new HashMap<>(); data.put("unicode","Fingerspitzengef\u00FChl is a German
        pronounced as follows: [\u02C8f\u026A\u014B\u0250\u02CC\u0283p\u026Ats\u0259n\u0261\u0259\u02CCfy\u02

        ArrayList<String> listData = new ArrayList<>();
        listData.add("Python");
        listData.add("JavaScript");
        listData.add("Java");
        listData.add(".NET Core");

        data.put("OW3 Workers", listData);

        return new SendResponse(data);
    }
}
```

To summarize, if you used the above module, you would call it from Omnis Studio through the Java Worker with $callmethod and use net.omnis.OmnisTest.Test as the module parameter and test as the method name.

Web Worker Objects

The following section refers to the Web Worker Objects (OWEB) available in Omnis Studio prior to Studio 8.1 (introduced in Studio 6.1.2): note they require Java to be installed and are therefore no longer supported in Studio 10.x or above: you should use the OW3 Workers for all new development.

SMTP Client Workers

The SMTP worker object allows you to submit email(s) to an SMTP server in a background thread, working in a similar way to the existing "worker objects" for DAMs, HTTP and Timers. In addition, it provides support for authentication methods not supported by the existing SMTPSend command including DIGEST-MD5, NTLM and OAUTH2.

In addition to the SMTP worker object, there is an object called EMAILMessage, which you use to build the message to be sent by the worker.

Software Requirements

The SMTP worker object relies on Java, and therefore relies on some Java files located in the 'java/axis/lib' folder in the main Omnis Studio folder: see the *Java Objects* chapter for details about running Java in Omnis.

EMAILMessage Object

The EMAILMessage object is used to construct the message to be sent to the SMTP Client worker. The EMAILMessage object has methods to manipulate the MIME body parts of the message. Each body part has a non-zero integer identifier that uniquely identifies the body part within the context of the object instance. EMAILmessage needs to remain in scope until the worker $completed or $cancelled message is called.

Note: If you specify a charset of kUniTypeAuto for binary or file body parts, the object will inspect the data, and set its charset according to the presence of a BOM (Byte Order Marker) to kUniTypeUTF8, kUniTypeUTF16BE, kUniTypeUTF16LE, kUniTypeUTF32BE or kUniTypeUTF32LE. If there is no BOM, the charset will be kUniTypeNativeCharacters, resulting in iso-8859-1 for Linux, macintosh for macOS or windows-1252 for Windows.

Methods

$createbodypartfromfile

`$createbodypartfromfile(cPath[,cMIMEType,iCharset,cEncoding,cDisposition,cFilename])`

Creates a MIME file body part. Returns non-zero integer body part id (unique for this EMAILMessage object) or zero if an error occurs.

Parameter	Description
cPath	The pathname of the file containing the body part data.The name of the file will also be used to set the filename for the attachment if you do not also pass the cFilename parameter
cMIMEType	The MIME type of the body part,in the standard syntax of type/subtype. If you omit this, the object will use a mapping table built into oweb.jar to generate the MIME type from the file extension; if this mapping fails, the type defaults to application/octet-stream
iCharset	A kUniType... constant (default kUniTypeAuto) indicating the charset of MIME type text/... (cannot be kUniTypeBinary or kUniTypeCharacter). If this body part needs a charset,it is assumed to be already encoded using this charset
cEncoding	The encoding to be used to transfer the body part data e.g. 'BASE64'. If omitted,the mail client chooses a default
cDisposition	The content disposition value to be used for the body part e.g. 'inline' or 'attachment'. If omitted, the mail client will use the default disposition
cFilename	The name sent as the filename of the body part.Defaults to the file name component of cPath if omitted

$createbodypartfromchar

`$createbodypartfromchar(cData[,cMIMEType,iCharset,cEncoding,cDisposition,cFilename])`

Creates a MIME body part from character data. Returns non-zero integer body part id (unique for this EMAILMessage object) or zero if an error occurs.

Parameter	Description
cData	The character data to be used as the content of the body part
cMIMEType	The MIME type of the body part,in the standard syntax of type/subtype. If you omit this, the type defaults to text/plain
iCharset	A kUniType... constant(default kUniTypeUTF8) indicating the charset to be used for the character data (cannot be kUniTypeAuto, kUniTypeBinary or kUniTypeCharacter)
cEncoding	The encoding to be used to transfer the body part data e.g. 'BASE64'. If omitted,the mail client chooses a default
cDisposition	The content disposition value to be used for the body part e.g. 'inline' or 'attachment'. If omitted, the mail client will use the default disposition
cFilename	The name sent as the filename of the body part. If omitted, no filename will be used

$createbodypartfrombin

`$createbodypartfrombin(xBin[,cMIMEType,iCharset,cEncoding,cDisposition,cFilename])`

Creates a MIME body part from binary data. Returns non-zero integer body part id (unique for this EMAILMessage object) or zero if an error occurs.

Parameter	Description
xBin	The binary data to be used as the content of the body part
cMIMEType	The MIME type of the body part,in the standard syntax of type/subtype. If you omit this, the type defaults to application/octet-stream
iCharset	A kUniType… constant (default kUniTypeAuto) indicating the charset of MIME type text/… (cannot be kUniTypeBinary or kUniTypeCharacter). If this body part needs a charset,it is assumed to be already encoded using this charset
cEncoding	The encoding to be used to transfer the body part data e.g. 'BASE64'. If omitted, the mail client chooses a default
cDisposition	The content disposition value to be used for the body part e.g. 'inline' or 'attachment'. If omitted, the mail client will use the default disposition
cFilename	The name sent as the filename of the body part. If omitted, no filename will be used

$createbodypartfromparts

```
$createbodypartfromparts(cMultiType,vPart[,iPart2,...])
```

Creates a MIME multi-part body part containing specified body parts. Returns non-zero integer body part id (unique for this EMAILMessage object) or zero if an error occurs.

Parameter	Description
cMultiType	The type of multi-part body part being created e.g. mixed
vPart	Either an integer body part id for a previously created body part in the EMAILMessage object or a single column list of integer body part ids for previously created body parts
iPart2	An integer body part id for a previously created body part in the EMAILMessage
…	Further parameters can be integer body part ids for previously created body parts

Note that each body part can only be used once in a multi-part body.

$deleteallbodyparts

```
$deleteallbodyparts()
```

Deletes all body parts that have been created using a $createbodypart… method. Also sets property $contentid to zero.

Properties

The EMAILMessage object has the following properties:

Property	Description
$errorcode	Error code associated with the last action (method call or property assignment) (zero means no error)
$errortext	Error text associated with the last action (method call or property assignment) (empty means no error)
$from	The email address of the sender
$subject	The subject of the message

Property	Description
$to	A space separated list of email addresses of the primary recipients of the message
$cc	A space separated list of email addresses of the carbon copied recipients of the message
$bcc	A space separated list of email addresses of the blind carbon copied recipients of the message
$priority	A kOWEBmsgPriority... constant that specifies the priority of the message. Defaults to kOWEBmsgPriorityNormal
$extraheaders	A list with 2 character columns (name and value). Each row in the list is an additional SMTP header to be sent to the server when submitting the message
$contentid	The id of the content to be sent with this email message. An integer body part id (returned by one of the $createbodypart... methods)

Constants

The EMAILMessage object has the following constants:

Constant	Description
kOWEBmsgPriorityLowest	The message has the lowest priority.
kOWEBmsgPriorityLow	The message has low priority.
kOWEBmsgPriorityNormal	The message has normal priority.
kOWEBmsgPriorityHigh	The message has high priority.
kOWEBmsgPriorityHighest	The message has the highest priority.

SMTPClientWorker Object

The SMTPClientWorker object uses the standard worker mechanism, with methods $init, $start, $run and $cancel, and callbacks $completed and $cancelled. In addition, there are some properties which further control the behavior of the object.

Methods

$init

```
$init(zMessage,cServer[,iSecure=kOWEBsmtpSecureNotSecure,iAuthType=kOWEBsmtpAuthTypeNone,cUser,cPassword,cOAUT
```

Initialize the worker object so it is ready to send message oMessage. Returns true if successful.

Parameter	Description
zMessage	An object reference to the EMAILMessage to send
cServer	The SMTP server that will send the message.Either a domain name or IP address.You can optionally specify the server port by appending ':'
iSecure	A kOWEBsmtpSecure... constant that indicates if and how the connection is to be made secure
iAuthType	A sum of kOWEBsmtpAuthType... constants that specify the allowed authentication types
cUser	The user name that will be used to log on to the SMTP server
cPassword	The password that will be used to log on to the SMTP server

Parameter	Description
OAUTH2	The SMTPClientWorker can manage OAUTH2 authentication for you. If you want it to do this, then this parameter is the name of the authority responsible for OAUTH2 authentication. This name is the name of a folder (which must exist) in secure/oauth2/smtp in the Omnis folder (the authority folder contains configuration and assets for this authority). In this case, cPassword is not used if OAUTH2 is the authentication mechanism chosen by the mail client. If you want to manage OAUTH2 yourself, then OAUTH2 must be empty or omitted, and the password must be the OAUTH2 access token. See the section on OAUTH2 for more details
cRealm	The realm used for DIGEST-MD5 authentication
cNTLMDomain	The domain used for NTLM authentication
lProps	A list with 2 character columns (name and value). A list of properties to be set for the JavaMail SMTP session object (see docs for package com.sun.mail.smtp at https://javamail.java.net/nonav/docs/api/). The SMTPClientWorker sets these properties as a final step, meaning that this can be used to override properties set by the worker, or to set other properties
bMailShot	If bMailShot is true (the default is false), the worker sends a separate copy of the message to each recipient (each recipient cannot see the email address of the other recipients). In this case, only 'to' recipients can be specified

$run

```
$run([cOAUTH2authCode])
```

Runs the worker on current thread. Returns true if the worker executed successfully. The cOAUTH2authCode parameter is discussed in the section on OAUTH2. It is not required in the initial call to $run to send an email.

$start

```
$start([cOAUTH2authCode])
```

Runs the worker on background thread. Returns true if the worker was successfully started. The cOAUTH2authCode parameter is discussed in the section on OAUTH2. It is not required in the initial call to $start to send an email.

$cancel

```
$cancel()
```

If required,cancels execution of worker on background thread. Will not return until the request has been cancelled.

$completed

```
$completed(wResults)
```

Callback method called when the request completes. Typically, you would subclass the SMTPClientWorker, and override $completed in order to receive the results. wResults is a row containing the results of executing the request. It has columns as follows:

Column name	Description
errorCode	An integer indicating the error that has occurred. Zero means no error occurred, and the email was successfully sent
errorInfo	Error text that describes the error. Empty if no error occurred
log	If the property $debuglog was set to kTrue before calling $init, this character column contains debugging information, including a log of the interaction with the SMTP server
oauth2_authcodeurl	Only applies when using OAUTH2 authentication managed by the SMTPClientWorker, and when errorCode is kOWEBsmtpErrorOAUTH2authCodeRequired. The URL to which the user needs to navigate in order to obtain an OAUTH2 authorisation code. See the section on OAUTH2 for more details

$cancelled

```
$cancelled()
```

Callback method called when the request has been cancelled by a call to $cancel(). Typically, you would subclass the SMTPClient-Worker, and override $cancelled.

Properties

The SMTPClientWorker object has the following properties:

Property	Description
$state	A kWorkerState… constant that indicates the current state of the worker object
$errorcode	Error code associated with the last action (zero means no error)
$errortext	Error text associated with the last action (empty means no error)
$threadcount	The number of active background threads for all instances of this type of worker object
$debuglog	If true,when the worker executes it generates a log of the interaction with the SMTP server.Must be set before executing $init for this object.The log is supplied as a column of the row variable parameter passed to $completed
$timeout	The timeout (in seconds) for SMTP requests (default is 30). Zero means infinite. Must be set before executing $init for this object
$clientsecret	The OAUTH2 client secret to be used to obtain tokens for OAUTH2. Must be set before executing $init for this object. Overrides the clientSecret (if any) in the OAUTH2 authority file. See the section on OAUTH2 for more details. This property is only relevant if the SMTPClientWorker is managing OAUTH2 authentication using an OAUTH2 authority

Constants

Authentication Types

These constants can be added together in order to form a bit mask of allowed authentication types:

Constant	Description
kOWEBsmtpAuthTypeNone	This has value zero, as a convenient way to indicate that no SMTP authentication is required.
kOWEBsmtpAuthTypeLOGIN	SMTP LOGIN authentication is allowed if the server supports it.
kOWEBsmtpAuthTypePLAIN	SMTP PLAIN authentication is allowed if the server supports it.
kOWEBsmtpAuthTypeDIGESTMD5	SMTP DIGEST-MD5 authentication is allowed if the server supports it.

Constant	Description
kOWEBsmtpAuthTypeNTLM	SMTP NTLM authentication is allowed if the server supports it.
kOWEBsmtpAuthTypeCRAMMD5	SMTP CRAM-MD5 authentication is allowed if the server supports it.
kOWEBsmtpAuthTypeOAUTH2	SMTP OAUTH2 authentication is allowed if the server supports it

Secure Connection Type

These constants indicate how the connection is to be made secure:

Constant	Description
kOWEBsmtpSecureNotSecure	The connection between client and server is not secure.
kOWEBsmtpSecureSSL	The connection between client and server uses SSL.
kOWEBsmtpSecureSTARTTLS	The connection between client and server is to be made secure by using the STARTTLS comma

OAUTH2

This section provides an overview of OAUTH2, including some key terms.

OAUTH2 provides a way for applications to perform actions on behalf of a user, provided that they have the permission of the user. So in the case of the SMTPClientWorker, when using OAUTH2 authentication, the Omnis Studio client needs to be given permission to send the email.

This all occurs in the context of an OAUTH2 authority, so for example if you are using GMAIL, the OAUTH2 authority is Google, or if you are using Windows mail, the OAUTH2 authority is Microsoft. he client application (Omnis Studio or more typically your application) needs to be registered as an application with the OAUTH2 authority; this gives it two key pieces of information:

- Client ID. A unique identifier for the client application.
- Client secret. A string used in requests to the OAUTH2 authority

  that authenticates the application. This needs to be kept as private as possible.

How you register your application with the OAUTH2 authority depends on the particular authority. For example:

- **For Google**
  go to the Google Developers Console (*https://console.developers.google.com*) and create a new project. On the credentials screen, create a new client ID for an installed application of type other.
- **For Microsoft**
  go to the Microsoft account Developer centre (*https://account.live.com/developers/applications/*) and create an application.
  In the API Settings make sure “Mobile or desktop client app” is set to Yes.

The user interfaces for these developer consoles allow you to obtain the client ID and client secret.

In order to use OAUTH2 authentication, the application needs to supply an OAUTH2 access token as the password. An access token is a short-lived password that is typically valid for an hour. The first time the application needs an access token, there has to be some interaction at the user interface:

- The application opens a browser window at the OAUTH2 authorisation code URL for the authority. Note that this URL includes a scope which indicates what type of permission is being requested. Each authority has a scope value which indicates that the user wants to manage email.
- The browser window may ask the user to log on to their account with the relevant authority. Once logged on, it will ask the user if they give the particular application permission to use their email. If the user agrees, the browser redirects to a URI called the redirect URI, passing an authorisation code to the URI. There are special redirect URIs for OAUTH2 authorities which cause the authorisation code to be available in the browser window after the user agrees - you need to use these special redirect URIs to use the SMTPClientWorker.
- The user copies the authorisation code from the browser window, and pastes it into the application.

- The application uses the authorisation code, client secret, client id and redirect URI to make an HTTP request to the token URL. A successful call to this URL returns an access token, expiry information for the access token (how long it is valid) and a refresh token.
- The application uses the access token as the password.

As part of the process described above, the application stores the access token, expiry information, and refresh token in permanent storage. The next time the application needs to log on, the application reads this information. If the access token is probably still valid, based on the expiry information, the application uses it. If not, the application uses the refresh token to make a slightly different request to the token URL, in order to get a new access token, which it then stores and uses to log on. Note:

- If the log on fails using the saved access token (with an authentication failure), the application will then try to use the refresh token to obtain a new access token.
- The refresh token may be invalidated by the authority at some point. For this, and various other reasons, log on may fail with an authentication failure. In that case, the application needs to return to the initial step of opening the browser window at the OAUTH2 authorisation code URL, so that it can obtain the user's permission, and a new access token and refresh token.

OAUTH2 for SMTPClientWorker

This section describes how the OAUTH2 support for the SMTPClientWorker works.

Authority Configuration

Each authority has a folder with the authority name in the folder secure/oauth2/smtp, in the Omnis data folder. In the installed tree, there are folders for two authorities, and each folder includes a file called config.json; this is a JSON file that configures the authority for use with the SMTPClientWorker. The installed authorities, and their JSON files, are:

```
gmail:
{
  "authURL": "https://accounts.google.com/o/oauth2/auth",
  "tokenURL": "https://www.googleapis.com/oauth2/v3/token",
  "proxyServer": "",
  "scope": "http://mail.google.com/",
  "redirectURI": "urn:ietf:wg:oauth:2.0:oob",
  "clientID": "",
  "clientSecret": ""
}

outlook:
{
  "authURL": "https://login.live.com/oauth20_authorize.srf",
  "tokenURL": "https://login.live.com/oauth20_token.srf",
  "proxyServer": "",
  "scope": "wl.imap,wl.offline_access",
  "redirectURI": "https://login.live.com/oauth20_desktop.srf",
  "clientID": "",
  "clientSecret": ""
}
```

When using these authorities, you need to supply your client ID. You can optionally store your client secret here, or if you want to keep it in another more secure location, you can store it how you like, and then supply it to the SMTPClientWorker using the $clientsecret property.

The proxyServer only requires a value if your client system is using a proxy server; the SMTPClientWorker uses this when making HTTP requests to the token URL.

User Storage

The SMTPClientWorker stores the access token, expiry information, and refresh token for a user in the file <user>.info in the authority folder, where <user> is typically the email address of the authorising user. This file is a JSON file, that is automatically handled by the SMTPClientWorker, so you should not need to edit this file.

Application Logic

After you have configured the authority, to use OAUTH2 in your application with the SMTPClientWorker, there is only one additional step you need to code in your application. Essentially, this comprises:

- Opening a browser window at the OAUTH2 authorisation code URL.
- Accepting the pasted authorisation code.
- Calling $run or $start for a second time, this time also passing the authorisation code.

For example, in $completed:

```
If pResults.errorCode=kOWEBsmtpErrorOAUTH2authCodeRequired

  OK message {A browser window will open so that you can   authorize sending email.//Please follow the instruct

  Launch program ,[pResults.oauth2_authcodeurl]

  While len(lAuthCode)=0
    Prompt for input Authorization code Returns lAuthCode (Cancel button)
    If flag false
      Yes/No message {Are you sure you want to cancel?}
      If flag true
        Quit method
      End If
    End If
  End While
  Do $cinst.$start(lAuthCode)
End If
```

External Commands

Note the Web and Email external commands are obsolete and are no longer supported in Studio 10.x or above: you should use the OW3 Workers for all new development.

The following protocols were supported: HTTP, FTP, SMTP, POP3, and IMAP. The external commands are prefixed with the respective protocol name, e.g. HTTPSend, FTPConnect, etc.

The Web and Email external commands are not displayed in the Code Editor since the 'Exclude Old Commands' filter is selected in the Code Editor (if you select No Filter from the Modify menu they will be available); these commands are described in the *Command Reference* and the Omnis Help (press F1) under the External commands group.

Multi-threading

The Web and Email external commands are multi-threaded, when running on a multi-threaded Omnis Server. The Web commands allow another thread to execute in the multi-threaded server while the current command runs. Note that the same socket cannot safely be used concurrently by more than one thread. See also the 'SMTP Client Workers' section for using email in multi-threaded environment.

MailSplit

If the encoding cannot be determined from the MIME, MailSplit uses $importencoding as the default encoding rather than UTF-8, provided that $importencoding is an 8 bit encoding, that is, anything except kUniTypeUTF16, kUniTypeUTF16BE and kUniTypeUTF16LE.

Email Headers

The SMTPSend and MailSplit commands support international characters in email headers (using RFC2047). The character limit of 76 for RFC2047 encoded words for mail headers has been removed in the MailSplit command.

SSL Security

The Web and Email external commands allow support for secure connections using Secure Sockets Layer (SSL) technology. However, Transport Layer Security (TLS) supersedes SSL and should be used in new development. Applications that require a high level of interoperability should support SSL 3.0 and TLS.

The HTTP, FTP, SMTP, POP3, and IMAP client commands that establish a connection to a server allow you to control if and how a secure connection is used. The commands which allow secure connections are:

FTPConnect	IMAPConnect
HTTPGet	POP3Connect
HTTPOpen	POP3Recv
HTTPPost	POP3Stat
HTTPSetProxyServer	SMTPSend

The parameters *Secure* and *Verify* allow you to enable support for secure connections. The parameters behave as follows:

- *Secure*
  is an optional Boolean* parameter which indicates if a secure connection is required to the server. Pass kFalse for non-secure (the default). Pass kTrue (value 1) for a secure connection; this enables the *Verify* option.
  *In addition, you can pass value 2 to some of the commands to enable specific types of authentication. The SSL package installed on yours or the client's system is used (Windows: Schannel, or macOS: Secure Transport). FTPS resumes the TLS session for data connections. In addition, it automatically sends PBSZ and PROT commands to the server after establishing a secure control connection

- *Verify*
  is an optional Boolean parameter which is only significant when Secure is not kFalse. When Verify is kTrue, the command instructs the SSL package to verify the server's identity using its certificate; if the verification fails, the connection will not be established. You can pass Verify as kFalse, to turn off the verification; in this case, the connection will still be encrypted, but there is a chance the server is an impostor.

For example, the FTPConnect command allows you to establish a secure connection to the specified FTP server; the full syntax of the command is:

```
FTPConnect (serveraddr, username, password [,port, errorprotocoltext, secure {Default zero insecure;1 secure;2
```

where *Secure* is an optional Boolean parameter which indicates if a secure connection is required to the server. Pass kTrue for a secure connection. If you pass secure with the value 2, the connection is initially not secure, but after the initial exchange with the server, FTPConnect issues an AUTH TLS FTP command to make the connection secure if the server supports it (see RFC 4217 for details), followed by further commands necessary to set up the secure connection. Authentication occurs after a successful AUTH TLS command. Note that if you use either of the secure options, all data connections are also secure, and all data transfer uses passive FTP.

When set to true (the default), the "ftpsresumesession" item in the "web" section of the config.json file ensures that the FTP commands attempt to resume the control connection session when establishing a secure data connection.

HTTPPage

The HTTPPage command has an additional parameter to allow you to ignore SSL. The full syntax of the command is:

```
HTTPPage (url[,Service|Port,pVerify]) Returns html-text
```

When passed as false, the pVerify argument prevents SSL verification when using a secure URL, so you can use:

```
HTTPPage (url,,kFalse)
```

SSL Packages

In order to use SSL or TLS in the Web and Email external commands the **Secure Channel** (Schannel) package on Windows, or **Secure Transport** on macOS *must be installed* on your development computer or a client's computer: these are present by default on their respective platforms.

OpenSSL

Existing Users should note: The Web and Email external commands relied on OpenSSL in previous versions to provide secure communications. Support for OpenSSL has been removed for these commands and support for SSL and TLS relies on the built-in security for each platform.

If you have used OpenSSL to provide secure comms for these commands in existing applications, you will need to switch to using either Schannel or Secure Transport depending on what platform your app is running on.

Certificate Authority Certificates

In order to perform the verification (when the Verify parameter is kTrue), the SSL package uses the Certificate Authority Certificates in the cacerts sub-folder of the secure folder in the Omnis folder. If you use your own Certificate Authority to self-sign certificates, you can place its certificate in the cacerts folder, and the SSL package will use it after you restart Omnis.

Web Command Error Codes

Error codes for the Web Commands are listed in the Commands Reference.

Chapter 8—Omnis Graphs

Some parts of this chapter refer to creating Charts and Graphs using the Graph2 window component, which is available for *Window classes only,* and therefore may not be available in your edition of Omnis Studio. You can however use the Graph2 External Object to generate a chart image in your Omnis code, save it as a JPG image and display it in the JavaScript Client in a remote form Picture control (this technique is shown in the **JS Picture** example app in the **Hub** in the Studio Browser), or you can add the chart image to a PDF report. Alternatively, you can use the Bar or Pie Chart JavaScript component to display a chart in your web & mobile apps. As a further alternative for displaying charts in the JavaScript Client, you could embed a third-party chart engine into a remote form, a technique which is described in the tech note: Integrating AmCharts into Omnis Studio TNJC0014.

About Graph2

You can create many different types of chart in Omnis using the Graph2 component and display them in your windows, remote forms, or reports. The data and appearance of a chart is based on the data stored in an Omnis list variable. The different chart types require a different list data structure to represent their data points. The Graph2 component supports four main types of chart: *XY charts, Pie charts, Polar charts* and *Meter charts,* each of which has a number of subtypes (except Pie). The major and minor types of the graph are specified as properties of the graph window object (found under External Components in the Component Store) and the associated list variable is specified in the $dataname property of the object.

Note to existing Omnis developers

The Graph2 component is based on a new graphing engine (called ChartDirector from Advanced Software Engineering Ltd) and was introduced to simplify graphing functionality in Omnis. The Graph2 component can produce many more types of chart and is easier to use than the old Graph component, which for backwards compatibility is still available in Omnis, but is no longer maintained or supported.

High Resolution Charts

From Studio 10.2 Rev 29538 onwards, the Graph2 component draws charts in high resolution suitable for display on high resolution displays. In this case, charts are generated at twice the size and are displayed at the correct physical size on high resolution displays.

You can disable this behavior by setting the property $disablehighresolution to kTrue (the default is kFalse meaning high resolution charts are supported). If the client is running on a display that does not support high resolution, the property will be set to kTrue automatically, and you will not be able to change the value of the property.

Chart Types

XY Charts

XY charts can be one of several different minor types or subtypes.

Bar charts

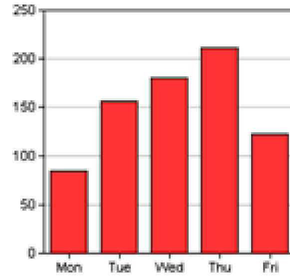


Figure 30:

Bar charts represent individual amounts, or compare individual amounts; you can change the graph orientation to show horizontal bars. There are several subtypes which you can select, including Line, Scatter, Area, and Box whisker.

Line charts

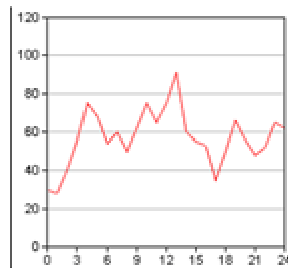


Figure 31:

Line charts emphasize the rate of change rather than individual amounts.

Scatter charts

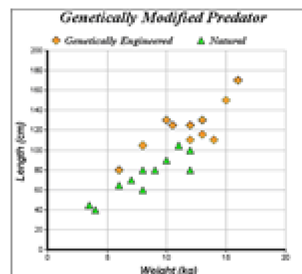


Figure 32:

Scatter charts show the relationship of different groups of numerical data, and plot the data as xy coordinates.

Area charts

Area charts emphasize the amount or magnitude of change rather than the rate of change.

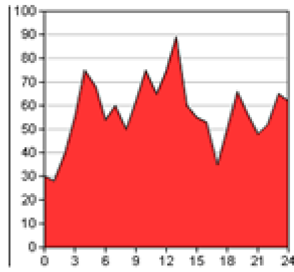


Figure 33:

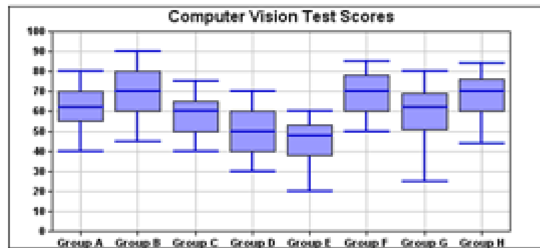


Figure 34:

Box whisker charts

Box whisker charts normally consist of five items of data to represent each element. Gantt charts are also constructed using the Box Whisker type, using three items of data per graph element.

High/Low/Open/Close charts

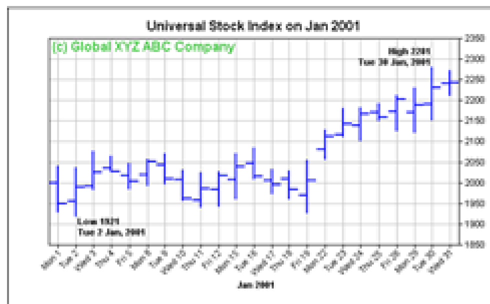


Figure 35:

High/Low/Open/Close (HLOC) charts are used to represent financial data; this type uses four data items to represent each element.

Candlestick charts

Candlestick charts are used to represent financial data; this type uses four data items to represent each element.

Pie Charts

The data in a pie chart is represented as sections, or slices of a pie. There are no minor types for this chart type, but pies have numerous visual effects.

Polar Charts

Polar Area

Polar charts represent data points on a radial axis with the values represented as the distance from the center; four minor types are available.

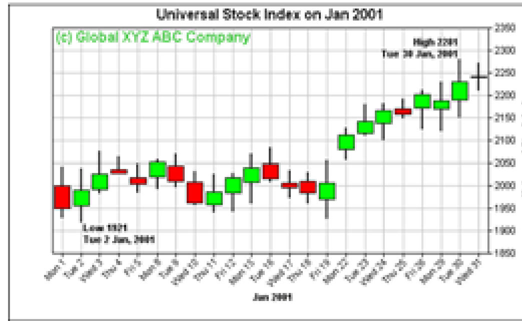


Figure 36:

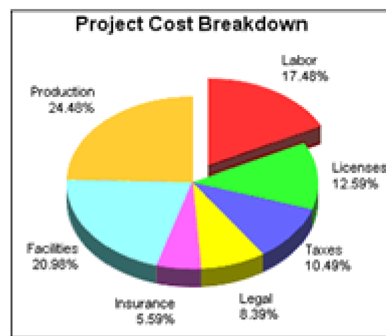


Figure 37:

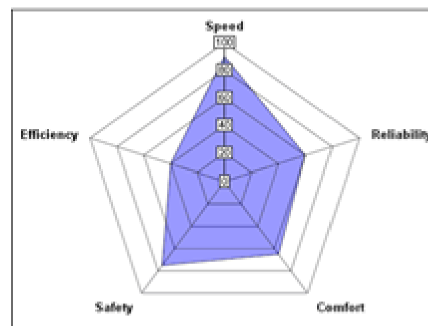


Figure 38:

Polar Line

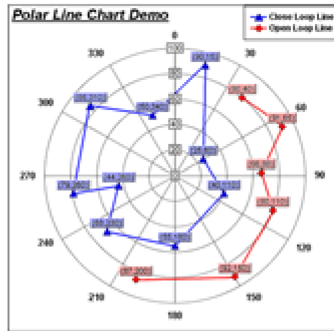


Figure 39:

See later in this section for a description of how to draw the 2 graphs above.

Polar charts represent data points on a radial axis with the values represented as the distance from the center; four minor types are available.

Polar Spline Line and Area

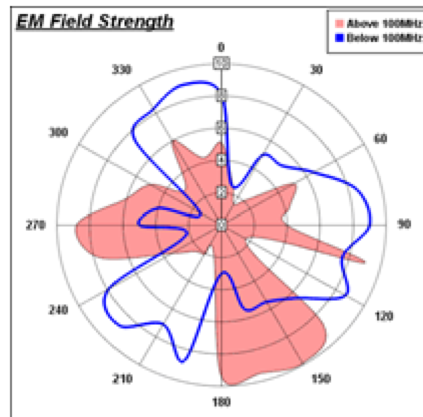


Figure 40:

Meter Charts

Numeric amounts or measurements can be shown as a meter or gauge; angular (or dial) or linear styles are available.

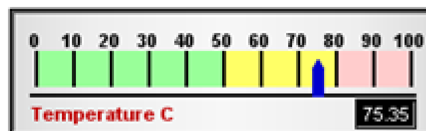


Figure 41:

A meter chart with the Linear subtype. In addition to changing the orientation of meter charts, you can add extra pointers for showing multiple data points, and colored zones to mark the scale or dial, for example.

A Meter chart with Angular subtype providing a dial or gauge effect, shown here with rounded frame

The Graph2 Example Library

There is an example library showing how the Graph2 component works, which is located under the Examples link in the Welcome window when you first launch Omnis – you can open this window by clicking on the New Users button in the main Omnis toolbar. Code from the example library is used later in this manual to show you how to use the component.

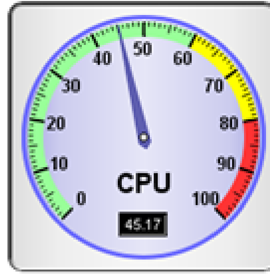


Figure 42:

Common Graph Properties

All the different graph types have the following properties; one of the most important properties is the $dataname which contains the name of the Omnis list variable containing the data for the graph; this is found under the **General tab** in the Property Manager.

Property Name	Data Type	Description
$dataname	List	The name of the Omnis list variable supplying the data to the graph; the structure of the data in the list must match the type of graph you wish to draw

The following common properties are found under the **Prefs tab** in the Property Manager.

Property Name	Data Type	Description
$deviceindependent	Boolean	If True report printing creates a device independent bitmap (DIB), used for printing in the Web client and cross-platform applications
$imagesearchpath	Char	The path and folder name where the Graph component will search for images macOS the property uses a standard HFS colon-separated pathname); if empty (the default), the component searches in the Omnis\Icons folder
$legendbackgroundcolor	RGB Color	The legend background color
$legendbackgroundeffect	Constant	The legend background effect: kG2colorSolid, kG2colorMetal, or kG2colorNotUs
$legendpos	Constant	The legend position: kG2legendNone, kG2legendLeft, kG2legendRight, kG2legendTop, kG2legendBottom, kG2legendManual ($legendx & legendy app
$legendtextcolor	RGB Color	The legend text color
$legendvert	Boolean	If True the legend is drawn vertically
$legendx	Integer	The X Position of the legend (only if $legendpos is kG2legendManual)
$legendy	Integer	The Y Position of the legend (only if $legendpos is kG2legendManual)
$majortype	Constant	The graph type, a constant: kG2xy, kG2pie, kG2polar, or kG2meter
$minorxytype$minorpolartype$minormetertype	Constant	Minor type for XY, Polar, or Meter graphs only; there are no minor types for Pie charts
$roundedframe	Boolean	If True the graph frame has rounded corners

The following common properties are found under the **Custom tab** in the Property Manager for all graph types.

Property Name	Data Type	Description
$3d	Boolean	If true the graph is 3d
$backgroundborder	RGB Color	The background border color
$backgroundcolor	RGB Color	The background color
$backgroundeffect	Constant	kG2colorSolid or kG2colorMetal
$backgroundraised	Integer	The degree to which the background is raised (if positive) or sunken (if negative); 0 is not raised
$columnheadings	List or row	A list or row containing the column headings of the list
$labelfont	Character	Name of the font for the graph label (appears under the main title); must be a font in the Omnis/fonts folder; see the Labels section later in this manual
$maintitle	Character	The main title of the graph
$offsetwidth	Integer	The offset width for the graph

Property Name	Data Type	Description
$offsetx	Integer	Additional x offset for the graph
$offsety	Integer	Additional y offset for the graph
$titlefont	Character	Name of the font for the graph title; must be a font in the Omnis/fonts folder; see the Labels section later in this manual
$titlefontheight	Integer	Height of the font for the graph title
$wallpaper	Character	The path/filename of an image; leave blank for no image
$xaxisfontangle	Integer	The angle of rotation, -1 is the default
$xaxistitle	Character	The title displayed on the X-Axis
$xlabelfontangle	Integer	Angle of rotation for X axis label, -1 is the default
$yaxisfontangle	Integer	angle of rotation for Y axis label, -1 is the default
$y2axisfontangle	Integer	angle of rotation for Y2 axis label, -1 is the default
$yaxistitle	Character	The title displayed on the Y axis

Setting the major and minor type

The different types of graph are specified using the $majortype and $minortype of the graph object. The following types are available, specified under the **Prefs** tab in the Property Manager:

Major type ($majortype)	Minor type	Minor type constants
kG2xy	$minorxytype	kG2xyBar kG2xyLine kG2xyScatter kG2xyArea kG2xyBoxwhisker kG2xyHLOC kG2xyCandlestick kG2xyTrend
kG2polar	$minorpolartype	kG2polarArea kG2polarLine kG2polarSplineArea kG2polarSplineLine
kG2meter	$minormetertype	kG2meterAngular kG2meterLinear
kG2pie	No minor types	

When you have placed a Graph2 component on your window, you can set its $majortype in the Property Manager and select the appropriate minor type. As you change the major and minor type of the graph, several properties will be shown or hidden according to the graph type selected.

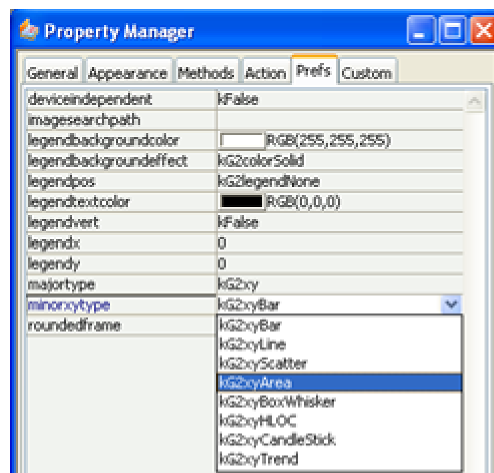


Figure 43:

Common Graph Methods

All the graph types have the following methods under the **Methods tab** in the Property Manager. More details about each method is supplied later in this section. Note these methods apply to the window graph object; methods are also available for the non-visual graph object, described in the second table.

Method Name	Returns	Description
$addmark()	None	$addmark(iAxisID, nValue, iColor [,cText, cFontname, iFontsize]) adds a ma or separator at the specified axis, value and color; you can add text to the mark with the specified font name and size
$addtext()	None	$addtext(cText, iX, iY [,cFontname, iFontsize, iColor, iAlign, iAngle, bVertica adds the specified text positioned according to the X & Y co-ordinates; thi must be done during the evPreLayout event; the co-ordinates are taken from the top left of the graph object and the Y component is inverted so t as Y increases the text string will appear further down the object
$addzone()	None	$addzone(iAxisID, iLowerlimit, iUpperlimit, iFillcolor) adds a zone from the lower and upper limits, in the color and on the axis specified
$convdate()	Integer	$convdate(dDatetime) converts a date/time variable to an integer equival which can then be used in graph data
$dispose()	None	Disposes the graph and rebuilds it. Useful if you need to change somethi other than a property on the graph which requires it to be rebuilt, such as adding more layers
$findobject()	kTrue for success	$findobject(iX, iY, iSetno, iItemno [,cSetname, cItemname]) obtains the se item information for the graph object under the mouse given its position the X & Y co-ordinates: see Drilldown
$formatvalue()	Char	$formatvalue(nValue, cFormatstring) formats a number/date using the formatting syntax described in the Parameter Substitution and Formattin section
$getcolors()	List	$getcolors([PaletteID]) returns a list of RGB colors used for the graph, or palette if specified, a constant: kG2paletteDefault, kG2paletteTransparent kG2paletteWhiteOnBlack
$getmainlayer()	Object	Returns the main layer object
$redraw()	None	Redraws the graph
$setcolors()	kTrue for success	$setcolors(lPaletteList) sets the colors for the graph using the specified lis RGB colors returned using $getcolors()
$setlinearscale()	None	$setlinearscale(iAxisid, nLowerlimit, nUpperlimit, [,nMajortick, nMinortick, lLabellist]) sets the linear scale for the specified axis; only available during Prelayout
$setlogscale()	None	$setlogscale(iAxisid, [cFormatstringORLowerlimit, nUpperlimit, ,cMajortic nMinortick, nLabellist]) sets the log scale for the specified axis; only availa during Prelayout
$snapshot()	Picture	$snapshot([iWidth, iHeight]) captures a snapshot of the graph with the specified width & height in pixels; omitting the parameters will return an image with the same dimensions as the current graph; the $snapshot() method causes the evPrelayout event to occur which allows you to add layers; see Graph Layers section below

The following method is available for the non-visual graph object, that is, object variables based on the Graph2 object.

Method Name	Returns	Description
$prelayout()	None	Available for object variables based on Graph2 component only; called during construction of the graph enabling layers to be added; see Graph Layers section below

XY Charts

The XY chart type has several subtypes, which all share the following properties and methods.

XY chart properties

Property Name	Data Type	Description
$datacombine	Constant	The combine method of the data for Bar, Area, and Line graphs, a constant: kG2dataSide, kG2dataStack, kG2dataOverlay, kG2dataPercentage; these constants can also be used in the $addarealayer(), $addbarlayer(), and $addlinelayer() XY chart methods
$3ddepth	Integer	Specifies the depth of a 3d XY chart; the default is –1 but can be set to a positive integer value to specify a custom depth which is useful for 3d line and area graphs with multiple series
$hgridcolor	RGB Color	The horizontal grid color
$minorxytype	Constant	The minor type of a kG2xy graph; can be one of the following: kG2xyBar, kG2xyLine, kG2xyScatter, kG2xyArea, kG2xyBoxWhisker, kG2xyHLOC, kG2xyCandleStick
$offsetheight	Integer	The additional offset height of the graph
$subtitle	Character	The subtitle of the graph
$swapxy	Boolean	If true, the X & Y axis are swapped
$xaxisontop	Boolean	If true, the X axis is to be displayed on the top
$xaxiswidth	Integer	Width of the X axis in pixels
$y2axistitle	Character	Y2 axis title
$yaxisonright	Boolean	If true, the Y axis is to be displayed on the right
$yaxiswidth	Integer	Width of the Y axis in pixels
$layereffect	Constant	The effect for all layers, a constant: kG2effectNone, kG2effectGlass, kG2effectSoftLight
$layereffectalign	Constant	The alignment or direction for the effect for all layers, as set by $layereffect; a constant: kG2alignTop, kG2alignBottom, kG2alignLeft, kG2alignCenter, kG2alignRight
$plotareacolorend	RGB Color	The end gradient color for the plot area
$plotareacolorstart	RGB Color	The start gradient color for the plot area
$vgridcolor	RGB Color	The vertical grid color

XY chart methods

All the following methods can only be executed during the evPreLayout event; see below for details.

| Method Name | Returns | Description | | ———————— | ——- | ———————————————- | | $addarealayer() | Integer | $addarealayer(pList [,iCombineType]) adds an area layer to the graph; pList is the area data* | | $addbarlayer() | Integer | $addbarlayer(pList [,iCombineType]) adds a bar layer to the graph; pList is the bar data* | | $addboxwhiskerlayer() | Integer | $addboxwhiskerlayer(pList) adds a box whisker layer to the graph; pList is the box whisker data* | | $addcandlesticklayer() | Integer | $addcandlesticklayer(pList) adds candlestick layer to the graph; pList is the candlestick data* | | $addhloclayer() | Integer | $addhloclayer(pList) adds a HLOC layer to the graph; pList is the HLOC data* | | $addlinelayer() | Integer | $addlinelayer(pList [,iCombineType]) adds a line layer to the graph; pList contains the Line data* | | $addscatterlayer() | Integer | $addscatterlayer(pList [,pSymbol, pSymbolSize]) adds a scatter layer to the graph.; pList is the scatter data*; pSymbol is an optional symbol type (of type kG2symbolXXX, defaults is kG2symbolSquare); pSymbolSize is the symbol size, default is 10 | | $addtrendlayer() | Integer | $addtrendlayer(pList) adds a trend layer to the graph using the data in pList* | | $getxaxis() | Object | $getxaxis([bSecondAxis]) returns the x-axis object, or the x2-axis object if bSecondAxis is true; only available during prelayout | | $getyaxis() | Object | $getyaxis([bSecondAxis]) returns the y-axis object, or the y2-axis object if bSecondAxis is true; only available during prelayout | * See appropriate chart section below for the format of the data in pList. In addition, see $datacombine for details of the various data combine types available.

List data structure for XY charts

Bar Charts

The bar chart is one of the most common forms of chart. The list format is one row per series with the first column being the group name followed by the data for each group. You can build the list data for your graph from a database or construct it on the fly: you need to use the $define() method to specify the columns in your list, and you can use $add() (or the SQL database list methods) to build the list line by line.

The following method will construct the list data for a simple bar chart.

```
# define vars listGraph (List), Name (Char), Sales (Long Int), Expenses (Long Int)
Do list.$define(Name,Sales,Expenses)
Do list.$add('Andy',85000,20000)
Do list.$add('Sam',80000,15000)
Do list.$add('Lisa',92000,34000)
Do list.$add('Harry',45000,15000)
```

The list variable listGraph is specified as the $dataname of the graph field. The above method will create a graph of two groups each with four series:

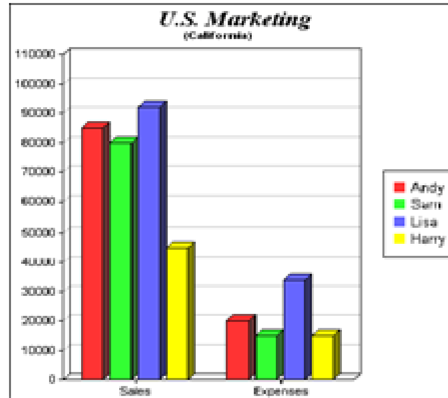


Figure 44:

Note you can use the Omnis 4GL commands to build the list data for the graph, either from an Omnis database, a SQL database, or on-the-fly, such as the following:

```
Set current list list
Define list {(Name,Sales,Expenses)}
Add line to list {('Andy',85000,20000)}
Add line to list {('Sam',80000,15000)}
Add line to list {('Lisa',92000,34000)}
Add line to list {('Harry',45000,15000)}
```

Line/Area Charts

The line and area chart in their data representation are very similar to the bar chart. So given the following data:

```
# define vars listGraph (List), colList (List), Name (Char), and use built-in #vars
Do list.$define(Name,#1,#2,#3,#4,#5,#6,#7,#8, #9,#10,#11,#12,#13,#14,#15,#16,#17,#18,#19,#20)
Do list.$add('Andy',30,28,40,55,75,68, 54,60,50,62,75,65,75,91,60,55,53,35,50,66,56,48,52,65,62)
Do list.$add('Liza',48,52,65,62,30,68,54,60, 50,28,40,55,75,62,75,65,75,35,50,66,56,91,60,55,53)
# now build the column headings list
Do colList.$define(#1,#2,#3,#4,#5,#6,#7,#8,#9,#10,#11,#12, #13,#14,#15,#16,#17,#18,#19,#20)
Do colList.$add(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20)
```

Note the use of the colList list in the portion of code, which can be assigned to the $columnheadings property of the graph field, overriding the normal x-axis variable names.

You should set the $majortype and minor type properties ($minorxytype or $minorpolartype) of the graph to specify its type and subtype. The above method will display the following line or area graph:

3d Line chart 3d Area chart $majortype=kG2xy, $minorxytype=kG2xyLine	**3d Area chart** \$majortype=kG2xy, $minorxytype=kG2xyArea

Scatter Charts

Each individual row in the chart list data represents a new plot (or series). The X & Y plot positions are then represented in two columns. Additional columns represent the other groups, so given the following data:

```
Do list.$define(Name,#1,#2,#3,#4)
Do list.$add('Pt1',10,130,5,100)
Do list.$add('Pt2',15,150,12,95)
Do list.$add('Pt3',6,80,8,105)
Do list.$add('Pt4',12,110,7,82)
Do list.$add('Pt5',10.5,125,10.5,99)
```

There are two groups of five series which will result in the following chart:

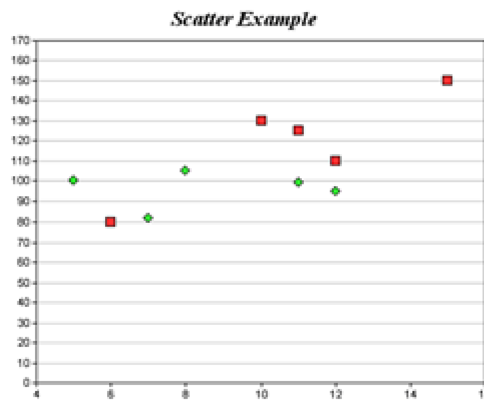


Figure 45:

Note that the symbols used to represent each series are in the order of square, diamond, triangle, right-triangle, left-triangle, inverted triangle, circle, cross, and cross #2.

Box Whisker Charts

Box whisker charts represent data ranges as boxes and/or marks. A common application is to represent the maximum, 3rd quartile, median, 1st quartile and minimum values of some statistics.

Each individual row in the chart list data represents each series.

```
Do list.$define(Name,#1,#2,#3,#4,#5)
Do list.$add('',55,70,80,40,62)
Do list.$add('',60,80,90,45,70)
Do list.$add('',50,65,75,40,60)
```

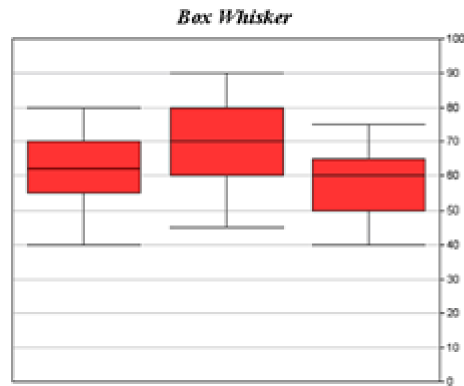


Figure 46:

Gantt Charts

You can create Gantt charts using a Box Whisker chart and setting the $swapxy property to kTrue so the bars are displayed horizontally. For Gantt charts you only need to provide two groups of data in your list. The example Graph2 library contains a simple Gantt chart, as follows:

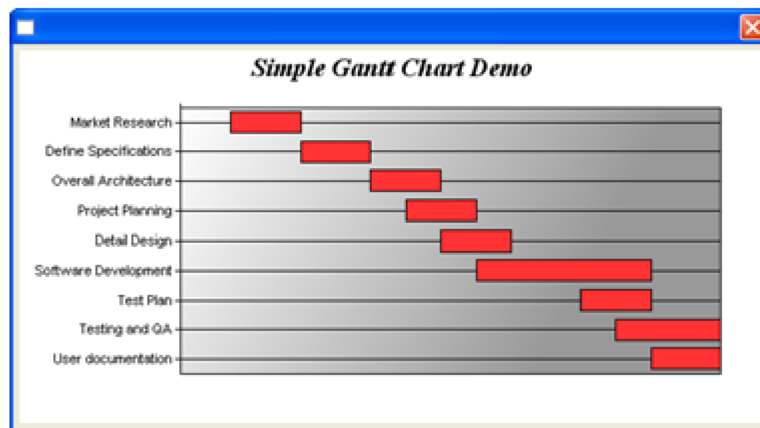


Figure 47:

The following method is executed in the $construct() of the Gantt chart window:

```
# define inst vars: iGraphList (List), Name (Char)
Do iGraphList.$define(Name,#1,#2)
Do method $addline ('Market Research',2004,8,16,14)
Do method $addline ('Define Specifications',2004,8,30,14)
Do method $addline ('Overall Architecture',2004,9,13,14)
Do method $addline ('Project Planning',2004,9,20,14)
Do method $addline ('Detail Design',2004,9,27,14)
Do method $addline ('Software Development',2004,10,4,35)
Do method $addline ('Test Plan',2004,10,25,14)
Do method $addline ('Testing and QA',2004,11,1,21)
Do method $addline ('User documentation',2004,11,8,14)
```

The $addline() class method is:

```
# Params: p1 (Char) p2-p5 (Long Int)
# Local var: chartdate (Date time D m y)
Calculate Name as p1
Calculate chartdate as dat(con(p2,"-",p3,"-",p4),"Y-M-D")
Calculate #1 as $cinst.$objs.ganttChart.$convdate(chartdate)
```

```
Calculate chartdate as dadd(kDay,p5,chartdate)
Calculate #2 as $cinst.$objs.ganttChart.$convdate(chartdate)
Add line to list
```

You can use the $convdate() method to convert a date/time variable to an integer equivalent which can be used in graph data.

High/Low/Open/Close (HLOC) Charts

The HLOC chart is often used to represent financial stock data. Each series consists of four pieces of information, the highest price, the lowest price, the open price, and the close price.

The volatility is then shown as a vertical line, with the opening price as a horizontal line to the left and the closing price as a horizontal line to the right.

```
Do list.$define(Name,#1,#2,#3,#4,#5)
Do list.$add('',2043,1931,2000,1950)
Do list.$add('',2039,1921,1957,1991)
Do list.$add('',2076,1985,1993,2026)
```

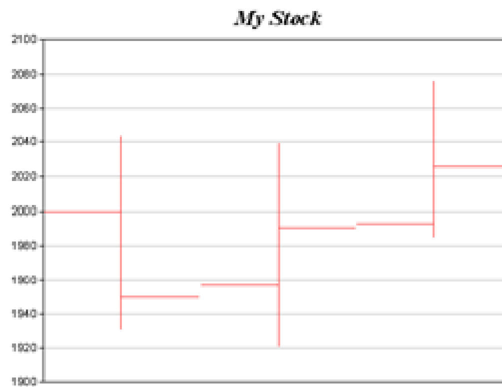


Figure 48:

Candlestick Charts

The candlestick chart is very similar to the HLOC chart and the list data is formatted in the same way. The difference in the graphical representation is that the area between the opening and closing price is shown as a filled rectangle; in the following example a black rectangle indicates that the closing price was lower than the opening price (a bad day). Both examples of the HLOC and Candlestick charts used the same data.

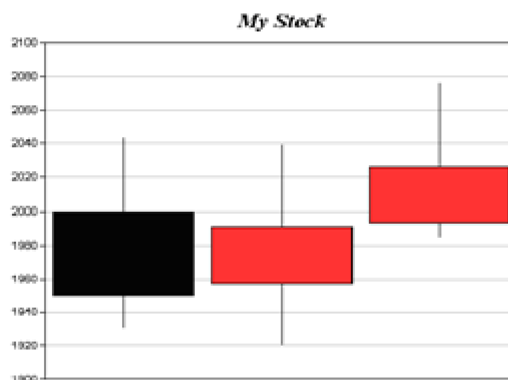


Figure 49:

Trend Chart

A trend chart allows you to view and compare data values over a given period or within a group. The first column in the data list contains the name of item and the second column onwards contains the data values for the item. A trend chart will contain one line for each row of data in the list.

```
# iColList is a List var assigned to $columnheadings of the graph object
Do lReturnList.$define( lName,#1,#2,#3,#4,#5,#6,#7,#8,#9,#10,#11,#12,#13,#14,#15,#16,#17,#18,#19,#20)
Do lReturnList.$add(0,50,55,47,34,42,49,63,62,73,59,56,50,64,60,67,67,58,59,73,77)
Do iColList.$define(#1,#2,#3,#4,#5,#6,#7,#8,#9,#10,#11,#12,#13,#14,#15,#16,#17,#18,#19,#20)
Do iColList.$add(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20)
```

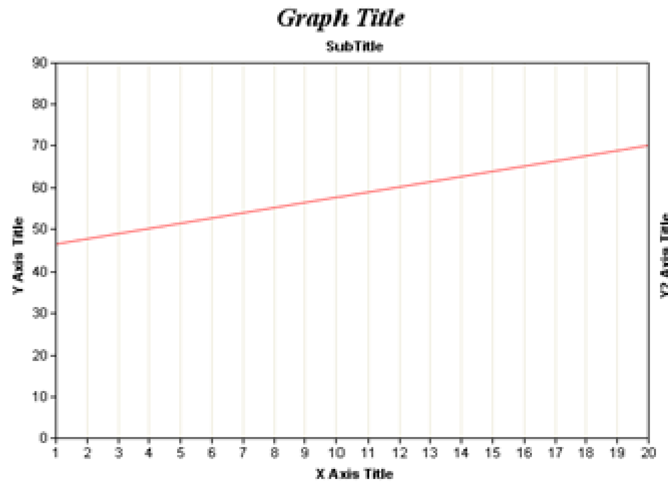
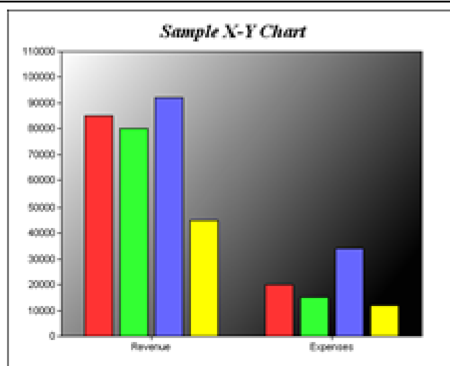


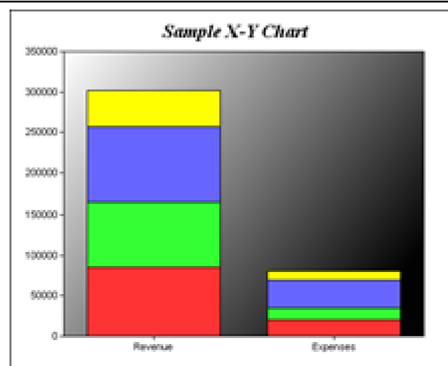
Figure 50:

Data presentation in XY charts

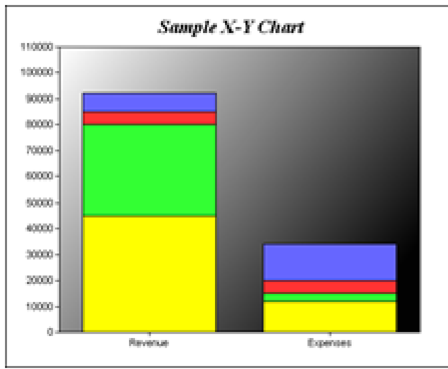
You can change the way data is presented in Bar, Area, and Line graphs using the $datacombine property, which can be set to one of the following constants: kG2dataSide, kG2dataStack, kG2dataOverlay, or kG2dataPercentage. When $datacombine is changed the bars, areas, or lines, and the Y-axis, are redrawn to reflect the selected combine type. The following example graphs shown how the different combine types affect the presentation of the same set of data (these graphs are available in the Graph example library).



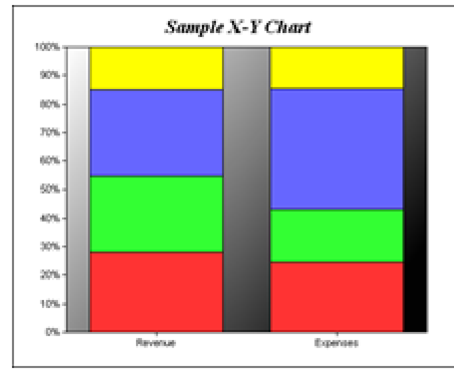
$datacombine = kG2dataSide: The data sets are shown as individual bars side-by-side (the default for XY bar charts)



$datacombine = kG2dataStack: The data sets are combined by stacking up the bar segments, therefore showing individual amounts in relation to the total amount for the data group



$datacombine = kG2dataOverlay: The data sets are overlayed each other providing a clearer comparison of data than the side-by-side type



$datacombine = kG2dataPercentage shows each individual amount as a percentage of the total for the data group

These different combine types can also be used when adding data layers to a graph using the $addarealayer(), $addbarlayer(), or $addlinelayer() XY chart methods; see the Graph Layers section.

Pie Charts

Pie charts have the following properties and methods, in addition to the Common graph properties and methods.

Pie chart properties

Property Name	Data Type	Description
$depth	Integer	The depth of the pie (-1 is default)
$depthcolumn	Integer	The column number (0 if not required) of the depth values in the list. This can be used to enable different slice depths
$donutradius	Integer	The amount of cut out from the center of the pie, specified as the percentage of the total radius of the pie; the range is 0-100, with 0 being no donut (the default)
$drawclockwise	Boolean	if true the slices are drawn clockwise
$feelercolor	RGB	The feeler color
$feelerwidth	Integer	The feeler width
$framecolor	RGB	The color of frame around pie slices, if $frameon is true
$frameon	Boolean	If true a frame is drawn around each pie slice
$labelformat	Character	Overrides the default formatting of labels; see section Parameter Substitution and formatting
$labelpos	Integer	Position of the labels relative to the edge of the pie, only used if $labelposon is true
$labelposon	Boolean	If true $labelpos is enabled
$rotate	Number	The pie rotation, in degrees
$shadow	Boolean	If true a pie shadow is displayed
$showfeeler	Boolean	if true feelers are shown
$sidelayout	Boolean	If true the labels are displayed by the side of pie
$tilt	Integer	If $tilton is enabled this contains the degree of tilt
$tilton	Boolean	if true the tilt is enabled (see $tilt); note $3d has to be enabled as well to display tilt

Pie chart methods

Pie charts have the following method(s), in addition to the common methods.

Method Name	Returns	Description
$slicemove()	None	$slicemove(iDistance, iSlice [,iSliceend]) moves iSlice or range of slices (iSlice to iSliceend) the specified iDistance from the center of the pie; the first slice in the chart is value 0; to move non-adjacent slices, iSlice should be a comma-separated list of slices, e.g. .$slicemove(iDistance,'0,2') to move slices 1 and 3

Note that some 'actions' on a pie chart are controlled by setting a particular property, such as $tilt; see the previous section for pie chart properties. Note also that you cannot add layers to a pie chart using methods, such as those used on XY and Polar charts.

List data structure for Pie charts

The following list data will result in the graph shown.

```
Do listGraph.$define(Name,Sales)
Do listGraph.$add('Andy',85)
Do listGraph.$add('Sam',80)
Do listGraph.$add('Liza',92)
Do listGraph.$add('Harry',45)
```

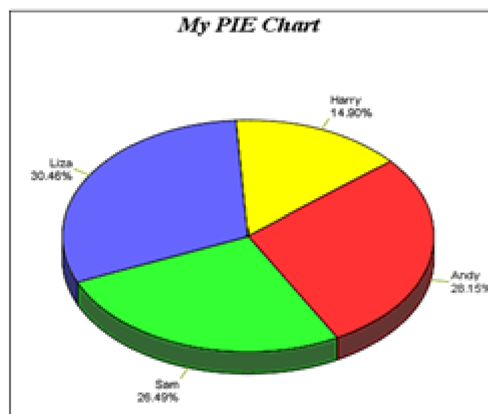


Figure 51:

Pie charts have several properties that allow you to alter the appearance of the chart, such as $tilt and $rotate. These can be used together with a set of sliders to create a dynamic representation of the user's data. See the Graph2 example in the Welcome screen when Omnis starts up (click the New Users button on the main Omnis toolbar) – the Graph2 example application shows many effects for Pie charts.

The code behind the slider can change the appropriate property in the pie chart, such as the following, which allows the user to change the tilt of the pie chart:

```
# $event method for Tilt slider
# $min and $max of slider component are set to 0 and 100
On evNewValue
  Calculate $cinst.$objs.GR.$tilt as pNewVal
```

The following code for a slider component allows the user to change the rotation of the chart.

```
# $event method for Rotate slider
# $min and $max of slider component are set to 0 and 360
On evNewValue
  Calculate $cinst.$objs.GR.$rotate as pNewVal
```

In addition, the $slicemove(iDistance, iSlice) method lets you move or 'slice out' one of the slices in the pie chart. For example, the Slice move slider in the above window has the following method:

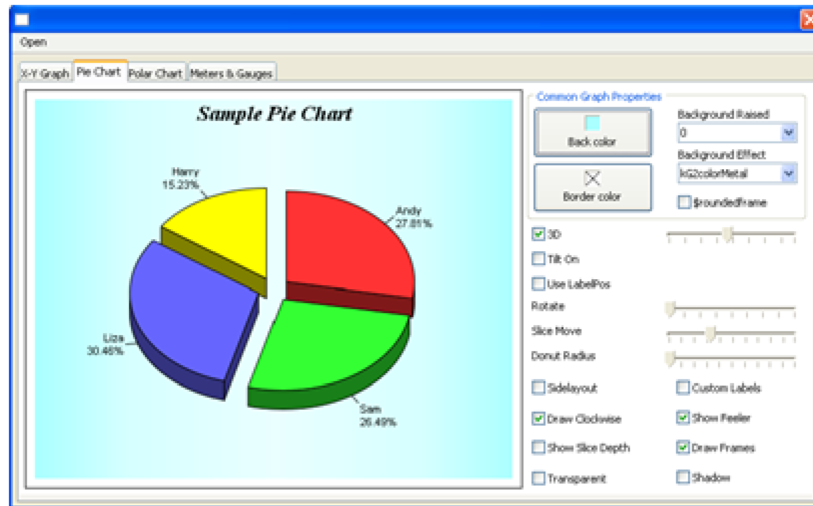


Figure 52:

```
# $event method for Slice-move slider\
# $min and \$max of slider component are set to 0 and 50
On evNewValue
  Do $cinst.$objs.GR.$slicemove(pNewVal,'0,2')   ## items 1 and 3
```

Using a combination of properties you can create some good effects for Pie charts. The following pie chart uses $sidelayout set to kTrue, a custom $labelformat, and $slicemove is set to slice out the first and the third slices in the chart.

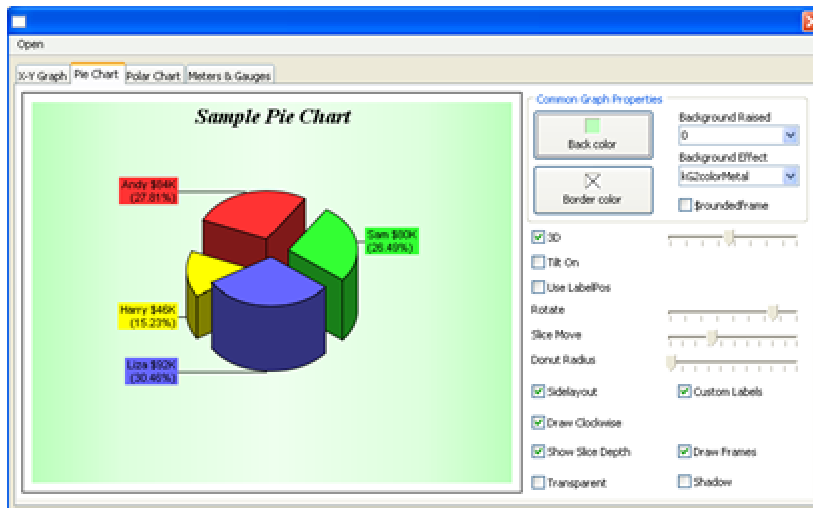


Figure 53:

Polar Charts

The Polar chart has the following properties and methods, in addition to the Common graph properties and methods.

Polar chart properties

Property Name	Data Type	Description
$angularcolor	RGB color	Color of angular grid
$angularlabelson	Boolean	If true labels are shown on the angular axis
$angularwidth	Integer	Width of the angular grid

Property Name	Data Type	Description
$circulargrid	Boolean	If true the grid is circular, otherwise the default is polygonal
$minorpolartype	Constant	The minor type for kG2polar type graphs; can be one of the following: kG2polarArea, kG2polarLine, kG2polarSplineArea, kG2polarSplineLine
$radialcolor	RGB color	Color of radial grid
$radiallabelson	Boolean	If true labels are shown on the radial axis
$radialwidth	Integer	Width of the radial grid

Polar chart methods

The following methods must only be executed during the evPreLayout event; see below for details.

Method Name	Returns	Description
$addarealayer()	None	$addarealayer(pList [,iCombineType]) adds an area layer to the graph; pList is the area data*
$addlinelayer()	None	$addlinelayer(pList [,iCombineType]) adds a line layer to the graph; pList contains the Line data*
$addsplinearealayer()	None	$addsplinearealayer(pList [,pSymbol, pSymbolSize]) adds a spline area layer to the graph; pList is the spline area data list (same format as all polar charts)
$addsplinelinelayer()	None	$addsplinelinelayer(pList [,pSymbol, pSymbolSize]) adds a spline line layer to the graph; pList is the spline line data list (same format as all polar charts)
$getradialaxis()	Object	$getradialaxis() returns the radial axis object

* See appropriate chart section for the format of the data in pList. In addition, see $datacombine for details of the various data combine types available.

List data structure for Polar charts

The data points in a polar chart are plotted using polar co-ordinates (*radius,angle*) whereby *radius* is the distance or amount from the center of the chart and *angle* is the relative angle with reference to the "12 o'clock position" in the chart. The first three columns in the Omnis list for a polar chart should therefore be Name (Label), Amount, Angle. If the angle is omitted from the list data, then the data points are distributed evenly around the chart. For example, consider the following data:

```
# define listGraph (List), Name (Char), Score (Long int)
Do listGraph.$define(Name,Score)
Do listGraph.$add("Speed",90)
Do listGraph.$add("Reliability",60)
Do listGraph.$add("Comfort",65)
Do listGraph.$add("Safety",75)
Do listGraph.$add("Efficiency",40)
```

Will give the following polar chart:

Note the following properties are set for the chart:
$majortype = kG2polar, $minorpolartype = kG2polarArea,
$legendpos = kG2legendManual, $legendx & $legendy = 10,
$angularlabelson = kTrue, $offsetwidth = 30,
$angularcolor = 187,187,255, $radialcolor = 153,153,255,
$maintitle = 'Speed Reliability Test'

Adding layers to a polar chart

To draw a second set of data in a polar chart, from a second list perhaps, you can add the additional data in a layer over the first set of data. You can add layers to charts during the evPrelayout event which is triggered as the graph component is instantiated.

Consider the following example. The first set of data is constructed in the $construct() method of the graph window: the second set of data is added as a layer.

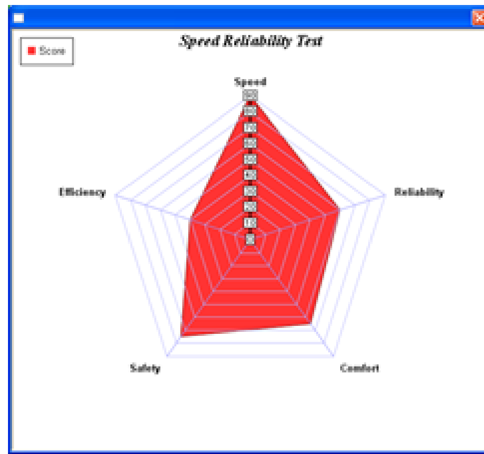


Figure 54:

```
# $construct() method for graph window
# define listGraph (List), listNames (List), Name (Char)
Do listGraph.$define(Name,#1,#3)
Do listGraph.$add("",90,15)
Do listGraph.$add("",25,60)
Do listGraph.$add("",40,110)
Do listGraph.$add("",55,180)
Do listGraph.$add("",68,230)
Do listGraph.$add("",44,260)
Do listGraph.$add("",79,260)
Do listGraph.$add("",85,310)
Do listGraph.$add("",50,340)

# $columnheadings of graph is set to listNames
Do listNames.$define(#S1)
Do listNames.$add("Closed loop")
```

The second data set is constructed during the evPrelayout event for the graph object and added as a line layer using the $addlinelayer() method.

```
# $event() method for graph window object
# define listLayer2 (List), local vars lLayer and OpenLoop
On evPreLayout
  Do list.$define(Name,OpenLoop,#1)
  Do list.$add("",80,40)
  Do list.$add("",91,65)
  Do list.$add("",66,88)
  Do list.$add("",80,110)
  Do list.$add("",92,150)
  Do list.$add("",87,200)
  Do $cinst.$objs.GROBJ.$getmainlayer() Returns lLayer
  Do lLayer.$setlinewidth(2)
  Do lLayer.$setsymbol(kG2symbolTriangle,11)
  Do lLayer.$setdatalabelformat("({value},{angle})")
  Do list.$define(#S1)
  Do list.$add("Open loop")
  Do $cinst.$objs.GROBJ.$addlinelayer(list) Returns lLayer
  Do $cinst.$objs.GROBJ.$setlinearscale(kG2axisAngular,0,360,30)
  Do lLayer.$setlinewidth(2)
  Do lLayer.$setcloseloop(kFalse)
  Do lLayer.$setsymbol(kG2symbolDiamond,11)
  Do lLayer.$setDataLabelFormat("({value},{angle})")
```

The above code produces the following graph:

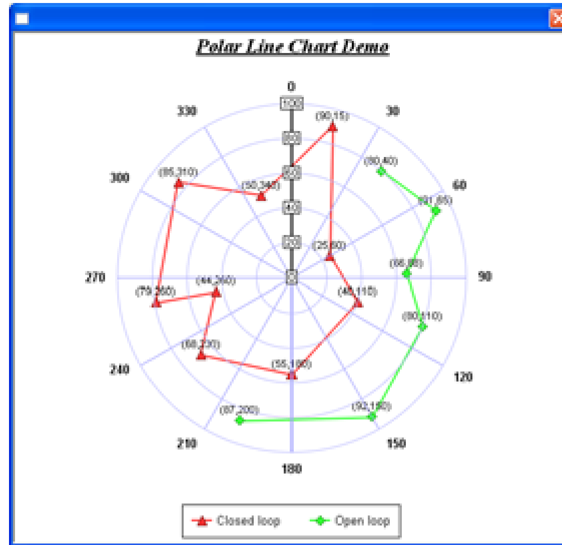


Figure 55:

Meter Charts

You can create Linear or Angular (radial) meter charts, and change their appearance using a range of properties and methods. Meter charts have the following properties and methods, in addition to the common graph properties and methods.

Meter chart properties

Property Name	Data Type	Description
$minormetertype	Constant	The minor type for meter charts, a constant: kG2meterLinear, kG2meterAngular
$linearalignment	Constant	The alignment for linear meter charts, a constant: kG2alignTop, kG2alignBottom, kG2alignLeft, kG2alignRight, kG2alignCenter
$angulardegrees	Integer	The start and end degrees for an angular chart in the form n,n, the default is 0,360
$angularpositionh	Integer	The horizontal position of the center (start) of an angular meter chart; the default is 50 which centers the chart horizontally
$angularpositionv	Integer	The vertical position of the center (start) of an angular meter chart; the default is 50 which centers the chart vertically
$majortickwidth	Integer	The width of the major tick in pixels; the default is 1
$microtickwidth	Integer	The width of the micro tick in pixels; the default is 1
$minortickwidth	Integer	The width of the minor tick in pixels; the default is 1
$pointercolor	RGB color	The color of the pointer for a meter chart

Changing the minor type for Meter charts

You can change the minor type of a meter graph using the following method:

```
Do $cinst.$objs.meterGraph.$minormetertype.$assign(kG2meterAngular)
```

or

```
Do $cinst.$objs.meterGraph.$minormetertype.$assign(kG2meterLinear)
```

Meter chart methods

Meter charts have the following methods in addition to the common methods.

Method Name	Returns	Description
$addpointer()	None	$addpointer(nValue, iColor [,nPointertype]) adds a pointer to a meter chart with the specified value and color; this must be done during the evPreLayout event; the pointer type is a number in the range 0-5; see the Adding Pointers section below
$addring()	None	$addring(iStartradius, iEndradius, iFillcolor) adds a colored ring to an angular meter with the specified start and end radius in pixels; this must be done during the evPreLayout event; you can use this to add a colored band or a number of bands around a standard meter dial; you can color the entire face by setting the start radius to zero and the end radius to beyond the meter ticks

List data structure for Meter charts

The Meter chart type (kG2meter) lets you create Angular and Linear meters to represent single data points. The properties of the meter type let you specify analog and digital read outs, colored backgrounds, multiple pointers per chart, pointers of different shapes, and so on. The example library demonstrates angular/radial and linear charts.

The data in the list associated with the graph object is used to specify the data point and various properties of the graph, as follows:

```
Do iGraphList.$add(45.17) ## the data point
Do iGraphList.$add(0)     ## scale start
Do iGraphList.$add(100)   ## scale end
Do iGraphList.$add(10)    ## major tick interval
Do iGraphList.$add(5)     ## minor tick interval
Do iGraphList.$add(1)     ## micro tick interval
```

The example library uses the above data to create the following chart:

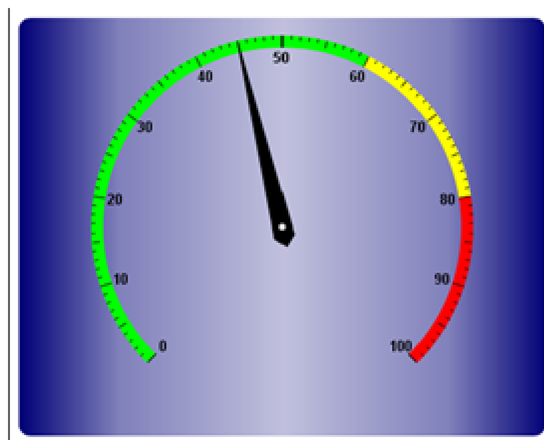


Figure 56:

Note that this and the next meter chart has colored zones defined on the scale; these are described on the *Adding Colored Zones* section.

When the minor type is set to kG2meterLinear, the following method:

```
Do iGraphList.$add(75.35)
Do iGraphList.$add(0)
Do iGraphList.$add(100)
Do iGraphList.$add(10)
```

Can be used to build the following linear chart:

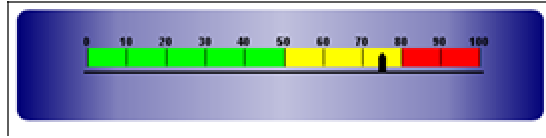


Figure 57:

Adding Pointers

The standard meter chart represents a single data point, which is shown using the default pointer, but you can add one or more additional pointers to a chart to show other data. You can add extra pointers to a meter chart (angular or linear) using the $addpointer() method, which must be executed during the evPreLayout event for the graph object; see the Graph Layers section for information about evPreLayout.

1. $addpointer(nValue, iColor [,nPointertype])
   adds a pointer to an angular or linear meter chart with the specified value and color; this must be done during the evPreLayout event; the pointer type is a number in the range 0-5, as follows:

0	Pencil pointer (the default pointer for linear charts)
1	Diamond pointer (the default pointer for angular charts)
2	Triangular pointer
3	Arrow with square ends
4	Arrow with sharp/pointed ends
5	Line pointer

The different pointer styles are shown in the following linear chart:

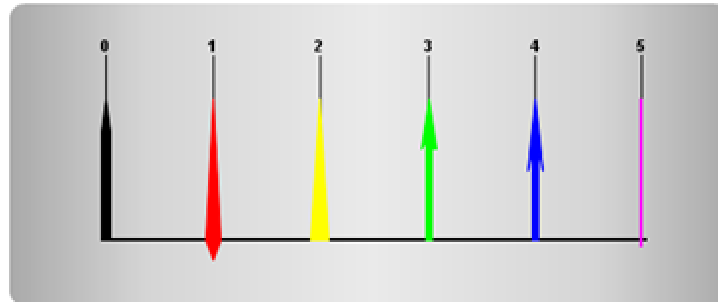


Figure 58:

The following method adds two pointers to an angular chart:

```
On evPreLayout
  If iAddpointer                 ## var behind checkbox on window
  Do $cinst.$objs.meterGraph.$addpointer(30,kGreen,2)    ## triangle
  Do $cinst.$objs.meterGraph.$addpointer(10,kMagenta,5) ## line
  End If
  ## etc
```

The method produces the following chart:

Adding Rings

A ring is the region in an Angular chart between two concentric circles. You can add rings to an Angular chart using the $addring() method, which is useful for adding circular borders and backgrounds to a meter; this must be done during the evPreLayout event.

- $addring(iStartradius, iEndradius, iFillcolor)
  adds a colored ring to an angular meter with the specified start and end radius in pixels

The difference between the start and end radius is, in effect, the width or thickness of the ring in pixels. If you use a start radius of zero, the ring will be drawn starting from the center of the chart face and, depending on the value of the end radius, will draw a circle on the chart with the given radius.

The example library demonstrates adding a ring to a radial meter chart, and uses the following method, which is added to the $event()

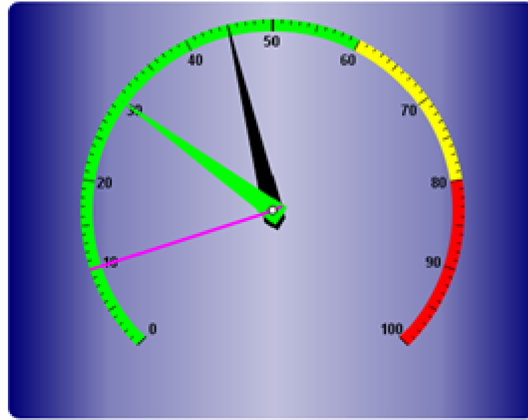


Figure 59:

```
On evPreLayout
  Do $cinst.$objs.meterGraph.$addring(
  173-abs(iBackgroundRaised),
  174-abs(iBackgroundRaised),
  $cinst.$objs.ringColor.$contents)
  # etc...
```

Which produces the following chart; note the ring has been changed to red using the color picker:

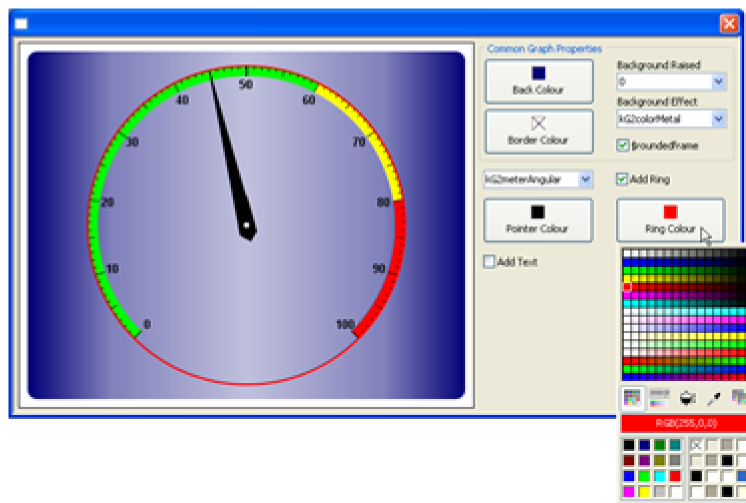


Figure 60:

In the example library, the start and end radius settings for the $addring() method take account of the raised background, since when the background is raised or inset by any amount, the radius of the meter dial is reduced by the same amount. The color for the ring is taken from the color picker, which is returned in the value of the $contents property of the color picker button.

A ring must be added to a meter chart during the evPreLayout event, which is triggered following a $dispose() method for the graph object. For example, the method for the color picker in the example library is as follows:

```
On evClick
  Do $cinst.$objs.meterGraph.$dispose()
```

This causes the $event() method for the graph window object to be called and the evPreLayout event is triggered, which in this case runs the method to add the ring.

Graph Layers and the Prelayout Event

You can add multiple layers to certain types of XY and Polar graph using the $add<layertype> methods during the Prelayout event (evPreLayout) event, which is triggered just before the graph instance is created in a window. Note you cannot add layers to Pie charts.

You can add layers to any type of XY or Polar chart using one of the $add<layertype> methods. In theory, any combination of chart type and layer is possible within the major XY and Polar types, but not all combinations are that meaningful. It largely depends on what type of data you are trying to represent or what data sets you might want to compare by showing the data sets as different layers.

Adding layers to charts

You can specify how the layer data is combined with or added to the graph using one of the data combine type constants as follows:

kG2dataOverlay	The additional data layer is added on top of the existing graph data
kG2dataPercentage	The existing and additional data sets are combined and scaled so each adds up to 100
kG2dataSide	The existing and additional data sets are shown side by side
kG2dataStack	The additional data set is stacked on top of the existing graph data
kG2dataOverlay	The additional data layer is added on top of the existing graph data

For example, in the case of the High/Low/Open/Close charts, it is quite common in financial applications to draw a line between the open and close points. This can be achieved by adding a line layer during the evPreLayout event using the $addlinelayer() method, as follows:

```
# $event() method for graph window object
# define var: lineLayer (List)
On evPreLayout
  Do lineLayer.$define(Name,#1,#2,#3)
  Do lineLayer.$add('',1950,1991,2026) ## Closing prices
  Do $cinst.$objs.GraphObj.$addlinelayer(lineLayer, kG2dataOverlay)
```

Refer to the appropriate layers section for the exact format of the list. The above method will result in the following graph:

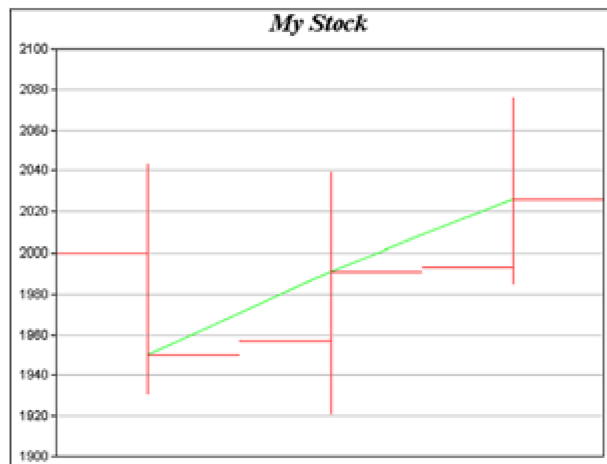


Figure 61:

When using an object variable based on the Graph2 component you can use the $prelayout() method in the object to add layers in the same way as described above.

Caution: A word of warning, evPreLayout events occur during the construction of a graph image which typically happens during the drawing of the graph control. This can make debugging the graph object's $event() method potentially troublesome.

Graph Clicks and Drilldown

The Graph2 window component reports an evGraphClick event when the user clicks on the object. The event returns the following event parameters:

pItem	The number of the item or data point clicked on within the current data set or group; the first item or data point is 1, the second is 2, and so on.
pItemname	The name of the data point clicked on.
pSet	The number of the data set or group clicked on; the first set or group is 1, the second is 2, and so on.
pSetname	The name of the data set or group clicked on.

For example, using the following method behind the $event() method of the graph object:

```
# the $event() for the graph object
On evGraphClick
  OK message {Graph Click =
      // Item Number [pItem] (Name = [pItemname])
      // Set Number [pSet] (Name = [pSetname])}
```

And clicking on the *first item* in the *second set* or *group* of the following graph

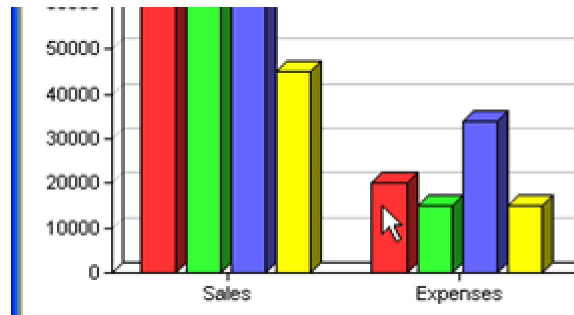


Figure 62:

Will give the following message:

```
Graph Click =
Item Number 1 (Name = Andy)
Set Number 2 (Name = Expenses)
```

You can use the item and set number/name information to drilldown into the data by reformatting the list data and redrawing the graph.

Note that pSet and pSetname are not valid for pie charts since they represent a single set of data only.

Changing the Color of Graph elements

The methods $getcolors() and $setcolors() allow you to get and set the color of elements within a graph at runtime. $getcolors() returns a list containing the RGB color of each element in the current graph.

The first eight color values in the color list have special significance. The first three palette colors are the background color, default line color, and the default text color of the chart. The fourth to seventh palette colors are reserved for future use. The eighth color is a special dynamic color indicating the “current data set”. The ninth color (index = 8) and subsequent colors in the list represent the elements in the graph, such as lines and text objects.

In a pie chart, Omnis will automatically use the ninth color for the first slice, the tenth color for the second slice, and so on. Similarly, for a multi-line chart, Omnis will use the ninth color for the first line, the tenth color for the second line, and so on.

The $setcolors() method allows you to set the color of any element in the graph. For example, the following method will set the color of the first data element in the chart to black (value 0):

```
# define local var: lColorList (List)
Do $cinst.$objs.GR.$getcolors() Returns lColorList
Calculate lColorList.9.1 as 0 ## 0 = black
Do $cinst.$objs.GR.$setcolors(lColorList)
```

You can return the color palette for a graph by specifying one of the palette constants in the $getcolors([PaletteID]) method. The constants are: kG2paletteDefault, kG2paletteTransparent, or kG2paletteWhiteOnBlack (the latter inverts the black and white colors in the graph). The following method can be used to switch to a transparent color palette for a graph:

```
# define local var: lColorList (List)
Do $cinst.$objs.polarGraph.$getcolors(kG2paletteTransparent) Returns lColorList
Do $cinst.$objs.polarGraph.$setcolors(lColorList)
```

The following method can be used to switch the graph colors back to the default palette, perhaps after having colored individual elements or switching the whole graph palette to transparent or White-on-black:

```
Do $cinst.$objs.polarGraph.$getcolors(kG2paletteDefault) Returns lColorList
Do $cinst.$objs.polarGraph.$setcolors(lColorList)
```

Adding Colored Zones

A zone is a color band on the back of the plot area. Like marks, zones can be horizontal or vertical. They are particularly useful for marking different parts of the scale on a meter chart, or adding bands of color behind a bar, area or line chart.

You can add zones to the background of a chart using the $addzone() method; this must be done in the evPreLayout event. For example, the following method can be used to set the color for different parts of the scale on a linear meter.

```
On evPreLayout
  Do $cinst.$objs.meterGraph.$addzone(0,50,kGreen)
  Do $cinst.$objs.meterGraph.$addzone(50,80,kYellow)
  Do $cinst.$objs.meterGraph.$addzone(80,100,kRed)
  # etc…
```

This method produces the following chart.

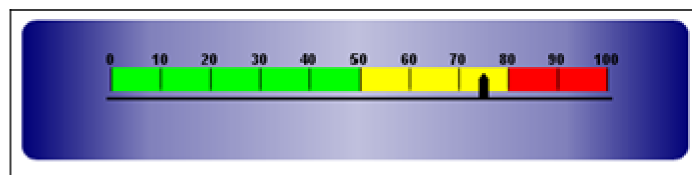


Figure 63:

Parameter Substitution and Formatting

Charts often contain a lot of text strings, such as sector labels in pie charts, axis labels for XY charts, data labels for the data points, graph titles, and so on. You can substitute many of these parameters to allow you to configure precisely the information contained in the text and their format. For example, when drawing a pie chart with side label layout, the default sector label format is:

```
"{label} ({percent}%)"
```

In drawing the sector labels, the graph component will replace "{label}" with the sector name, and "{percent}" with the sector percentage. So the label will be something like:

```
"ABC (34.56%)"
```

You can change the sector label format by changing the format string. For example, you can change it to:

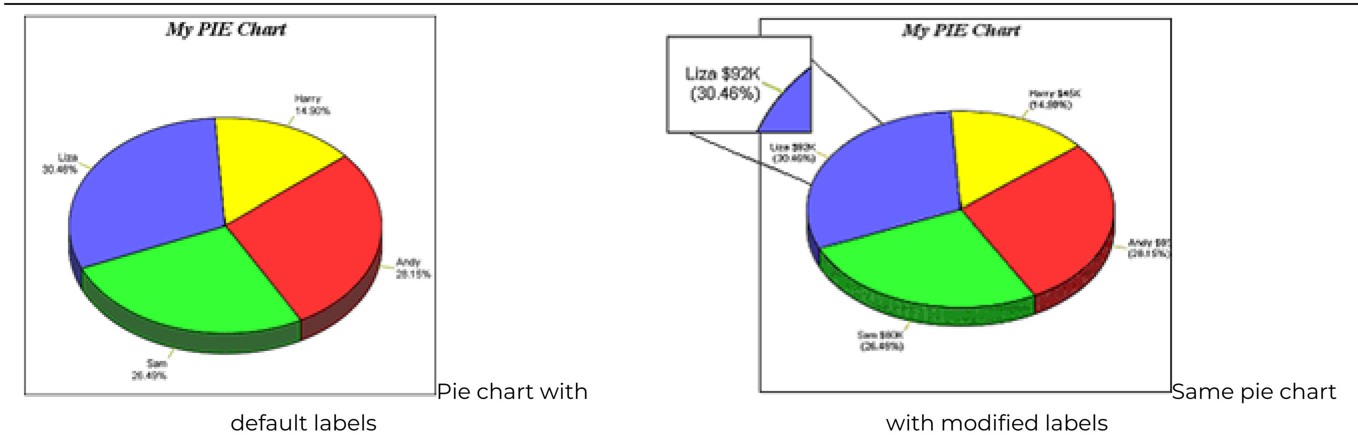
```
"{label}: US\${value}K ({percent}%)"
```

The sector value will then be something like:

```
"ABC: US\$123 (34.56%)"
```

Here is an example method that changes the $labelformat property in a pie chart.

```
Calculate $cinst.$objs.GROBJ.$labelformat as con("{label} ${value}K",chr(10),"({percent}%)")
```



Pie chart with default labels

Same pie chart with modified labels

For fields that are numbers, dates, or times, the Graph component supports a special syntax in parameter substitution to allow formatting of these values. Please refer to the *Number Formatting* and *Date/Time Formatting* sections below for details.

The following tables describe the fields available for various chart objects.

Parameters for Pie charts

Parameter	Description
sector	The sector number. The first sector is 1, the second is 2, and so on.
dataSet	Same as {sector}. See above.
label	The text label of the sector.
dataSetName	Same as {label}. See above.
value	The data value of the sector.
percent	The percentage value of the sector.

Parameters for all XY Chart Layers

The following are parameters that apply to all XY Chart layers in general. Some layer types may have additional parameters (see below).

Note that some parameters do not apply in certain cases. For example, when specifying the aggregate label of a stacked bar chart, the {dataSetName} parameter does not apply, because a stacked bar is composed of multiple data sets. It does not belong to any particular data set and hence does not have a data set name.

Parameter	Description
x	The x value of the data point.
xLabel	The bottom x-axis label of the data point.
x2Label	The top x-axis label of the data point.
value	The value of the data point.
accValue	The accumulative value of the data point. This is useful for stacked charts, such as stacked bar chart and stacked area chart.
totalValue	The total value of all data points. This is useful for stacked charts, such as stacked bar chart and stacked area chart.
percent	The percentage of the data point based on the total value of all data points.

Parameter	Description
accPercent	The accumulated percentage of the data point based on the total value of all data points. This is useful for stacked charts, such as stacked bar chart and stacked area chart.
dataSet	The data set number to which the data point belongs. The first data set is 1, the second is 2, and so on.
dataSetName	The name of the data set to which the data point belongs.
dataItem	The data point number within the data set. The first data point is 1, the second is 2, and so on.
dataGroup	The data group number to which the data point belongs. The first data group is 1, the second is 2, and so on.
dataGroupName	The name of the data group to which the data point belongs.
layerId	The layer number to which the data point belongs. The first layer is 1, the second is 2, and so on.

Additional Parameters for HLOC and CandleStick Layers

The following are in addition to the parameters for all XY Chart layers.

Parameter	Description
high	The high value of the data representation.
low	The low value of the data representation.
open	The open value of the data representation.
close	The close value of the data representation.

Additional Parameters for Box Whisker Layers

The following are in addition to the parameters for all XY Chart layers.

Parameter	Description
top	The value of the top edge of the box-whisker symbol.
bottom	The value of the bottom edge of the box-whisker symbol.
max	The value of the maximum mark of the box-whisker symbol.
min	The value of the minimum mark of the box-whisker symbol.
med	The value of the median mark of the box-whisker symbol.

Additional Parameters for Trend Layers

The following are in addition to the parameters for all XY Chart layers.

Parameter	Description
slope	The slope of the trend line.
intercept	The y-intercept of the trend line.
corr	The correlation coefficient in linear regression analysis.
stderr	The standard error in linear regression analysis.

Parameters for Polar Charts

Parameter	Description
radius	The radial value of the data point.
value	Same as {radius}. See above.
angle	The angular value of the data point.
x	Same as {angle}. See above.
label	The angular label of the data point.
xLabel	Same as {label}. See above.
name	The name of the layer to which the data point belongs.
dataSetName	Same as {name}. See above.

Parameter	Description
i	The data point number. The first data point is 1, the second is 2, and so on.
dataItem	Same as {i}. See above.

The following method sets the label for the data points in a polar chart.

```
Do $cinst.$objs.GROBJ.$getmainlayer() Returns lLayer
Do lLayer.$setdatalabelformat("({value},{angle})")
```

Parameters for Axis

Parameter	Description
value	The axis value at the tick position.

Number Formatting

For parameters that are numbers, the Graph component supports a number of formatting options in parameter substitution.

For example, if you want a numeric field *(value)* to have a precision of two digits to the right of the decimal point, you can use *(value|2.,)* where 2 and '.' (dot) is used to specify the decimal point precision, and ',' (comma) as the thousand separator. The number *123456.789* will therefore be displayed as *123,456.79*.

For numbers, the formatting options are specified using the following syntax:

`([param]|[a][b][c][d])`

where:

Parameter	Description
[param]	The name of the parameter
[a]	An integer specifying the number of digits to the right of the decimal point. The default is automatic. To use the default, simply skip this parameter.
[b]	The thousand separator. Should be a non-alphanumeric character (not 0-9, A-Z, a-z). Use '~' for no thousand separater.
[c]	The decimal point character.
[d]	The negative sign character. Use '~' for no negative sign character.

You may skip the trailing formatting options if they are needed. For example, *(value|2)* means formatting the value with two digits to the right, where the thousand separator, decimal point character, and negative sign character are all using the default settings of the chart.

Date/Time Formatting

For parameters that are dates & times, the formatting options can be specified using the following syntax:

`([param]|[datetime_format_string])`

where [datetime_format_string] must start with an alphabetic character (A-Z or a-z), and may contain any characters except ')'. Certain characters are substituted according to the following table:

Parameter	Description
yyyy	Year in 4 digits (e.g. 2005)
yyy	Year showing only the least significant 3 digits (e.g. 007 for the year 2007)
yy	Year showing only the least significant 2 digits (e.g. 07 for the year 2007)
y	Year showing only the least significant 1 digit (e.g. 7 for the year 2007)

Parameter	Description
mmm	Month formatted as its name. The default is to use the first 3 characters of the English month name (Jan, Feb, Mar ...).
mm	Month formatted as 2 digits from 01 - 12, with leading zero if necessary.
m	Month formatted using the minimum number of digits from 1 - 12.
dd	Day of month formatted as 2 digits from 01 - 31, with leading zero if necessary.
d	Day of month formatted using the minimum number of digits from 1 - 31.
w	The name of the day of week. The default is to use the first 3 characters of the English day of week name (Sun, Mon, Tue ...).
hh	The hour of day formatted as 2 digits, adding leading zero if necessary. The 2 digits will be 00 - 23 if the ‘a’ option (see below) is not specified, otherwise it will be 00 - 12.
h	The hour of day formatted using the minimum number of digits. The digits will be 0 - 23 if the ‘a’ option (see below) is not specified, otherwise it will be 0 - 12.
nn	The minute formatted as 2 digits from 00 - 59, adding leading zero if necessary.
n	The minute formatted using the minimum number of digits from 00 - 59.
ss	The second formatted as 2 digits from 00 - 59, adding leading zero if necessary.
s	The second formatted using the minimum number of digits from 00 - 59.
a	Display either ‘am’ or ‘pm’, depending on whether the time is in the morning or afternoon.

For example, a parameter substitution format of *(value|mm-dd-yyyy)* will display a date as something similar to *09-15-2002*. A format of *(value|dd/mm/yy hh:nn:ss a)* will display a date as something similar to *15/09/02 03:04:05 pm*.

Further formatting options

You can use a type of Mark Up Language (supported in the underlying graph engine and called ChartDirector Mark Up Language or CDML) to include formatting information in text strings by marking up the text with tags. This Mark Up Language allows a single text string to be rendered using multiple fonts, with different colors, and even embed images in the text. CDML is supported in all text objects including chart titles, legend keys, axis labels, data labels, and so on.

Font Styles

You can change the style of the text by using special tags. For example, the following code substitutes some of the parameters and formats the fonts using tags:

```
Calculate $cinst.$objs.GR.$labelformat as
con("<*font=timesi.ttf,size=16,color=FF0000*>{label}
 ${value}K",chr(10),
 "<*font=arial.ttf,size=12,color=8000*>({percent}%)")
```

In general, all tags in CDML are enclosed by **<*** and ***>**. Attributes within the tags determine the styles of the text following the tags within the same block. If you want to include **<*** in text without being interpreted as CDML tags, use **<<*** as the escape sequence.

The following font style attributes can be used:

- font The font file name.
- size The font size.
- width The font width. This attribute is used to set the font width and height to different values. If the width and height are the same, use the *size* attribute.
- height The font height. This attribute is used to set the font width and height to different values. If the width and height are the same, use the *size* attribute.
- color The text color in hex format.
- underline The line width of the line used to underline the following characters. Set to 0 to disable underline.

Embedding Images

You can embed images in text using the <*img*> tag and the following syntax:

```
<*img=my_image_file.png*>
```

You can use the $imagesearchpath property (on the Preference tab) to specify the folder where the Graph component will search for images (on macOS the property uses a standard HFS colon-separated pathname). If you leave the property blank (the default), the component sets the search path to the icons folder in the Studio tree, which means your image file must be located in this folder. Having set the image search path you can use the image file name only in the <*img*> tag. For example, the line:

```
# set $imagesearchpath to C:\Program Files\RainingData\OS43\images
Calculate $cinst.$objs.GR.$labelformat as con(
 "{label} ${value}K",chr(10),
 "({percent}%)",chr(10),"<*img= sun_bullet.gif*>")
```

embeds the image file sun_bullet.gif which is located in the Omnis\images folder.

Labels

You can override the default fonts for the main title and labels in a graph by setting the $titlefont and $labelfont properties (found under the Custom tab in the Property Manager). You can specify any true type font (TTF) or True type collection (TTC) that is located in the Fonts folder in the main Omnis folder. If a folder called 'Fonts' does not exist you can create one with this name. If you add fonts to this folder, you have to restart Omnis before the fonts become available in Omnis.

The default value is set to "(DEFAULT)" which means the default font for your system will be used (e.g. Times Roman under Windows). You can adjust the height or size of the main graph title by setting the $titlefontheight property.

Adding Text to a Chart

You can add text to a chart using the $addtext() method, but this must be done during the evPreLayout event.

- $addtext(cText, iX, iY [,cFontname, iFontsize, iColor, iAlign, iAngle, bVertical])
  adds the specified text positioned according to the X & Y co-ordinates

The X & Y co-ordinates are taken from the top left of the graph object, but the Y component is inverted, so that as Y increases the text string will appear further down the chart. The following method adds some text to a Linear meter chart. Note the $formatvalue() method is used to format a number before it is added.

```
On evPreLayout
  Calculate lTextToAdd as $cinst.$objs.meterGraph.$formatvalue(75.35,"2")
  Do $cinst.$objs.meterGraph.$addtext(
    lTextToAdd,345,70,
    "arialbd.ttf",8,kBlack,kG2alignBottomRight)
  Do $cinst.$objs.meterGraph.$addtext(
    "Temperature °C",10,68,
    "arialbd.ttf",8,kBlack,kG2alignBottomLeft)
```

This method produces the following chart:

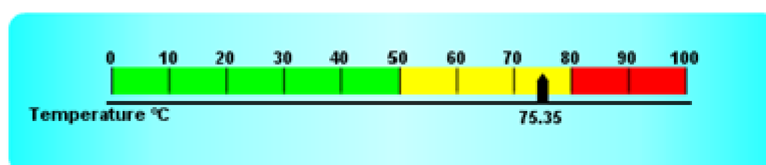


Figure 64:

Using Graphs in Reports

You can use the Graph2 component on Omnis reports in much the same way as you can for windows for displaying data contained in an Omnis list. That is, you can place a graph component on a report class, assign a list variable to it, print the report and the data will be displayed in a chart as expected, depending on the properties you have set in the graph object.

However, if you wish to manipulate a graph using any of the graph methods, this must be done within the report instance itself and before the report is printed. Therefore to do this, you must instantiate an object variable based on the Graph2 component, execute any methods against the graph object and transfer the image of the graph to an Omnis picture variable using the $snapshot() method. The graph image contained in the picture variable can be displayed in the report using a standard Omnis Picture field.

For example, the following report class has a single picture field with its $dataname set to the variable iPicture. The $height and $width properties are set to 7.9cm and 10.5cm respectively, which corresponds the height and width of the graph image created in the report instance.

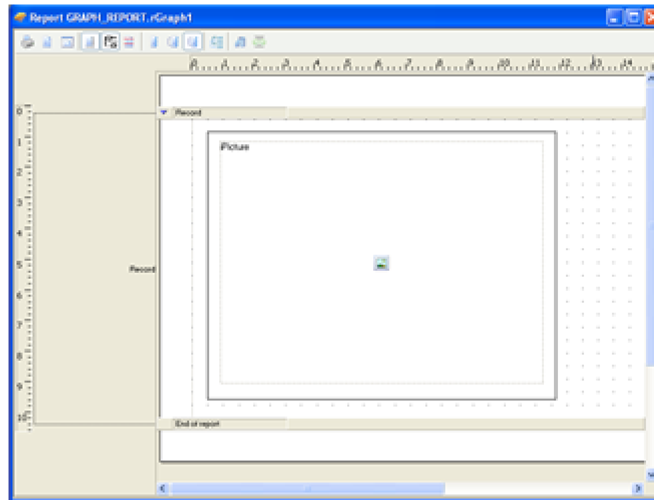


Figure 65:

The following method is contained in the $construct() method of the report class, therefore the method is executed when the report is instantiated, and the graph object is created and transferred to the picture variable before the report is printed. The $snapshot() method is used to capture the graph image and returns it to the picture variable.

```
# Define class var: cGraphObj (Object), subtype Graph2
# Define instance var: iList1 (List), iPicture (Picture)
# Define instance var: iCol1, iCol2 (Char)
Do iList1.$define(iCol1,iCol2)
Do iList1.$add('Col1',10)
Do iList1.$add('Col2',60)
Do iList1.$add('Col3',30)
Do cGraphObj.$majortype.$assign(kG2xy)
Do cGraphObj.$dataname.$assign(iList1)
Do cGraphObj.$snapshot(400,300) Returns iPicture
Do $cinst.$printrecord()
Do $cinst.$endprint()
```

Note that the graph object will use all the default properties, so if you want to change the appearance, type or subtype of graph, you need to change the appropriate properties of the graph object at runtime using the notation. The above method sets the graph major type and assigns the list variable to the object.

The report will look something like the following:

Using Graphs in the Web client

If you wish to display a chart in a remote form you have to use a similar technique to displaying graphs in a report – this is because there is no visual graph component for use with remote forms. Instead, you can use the graph component as an object variable and

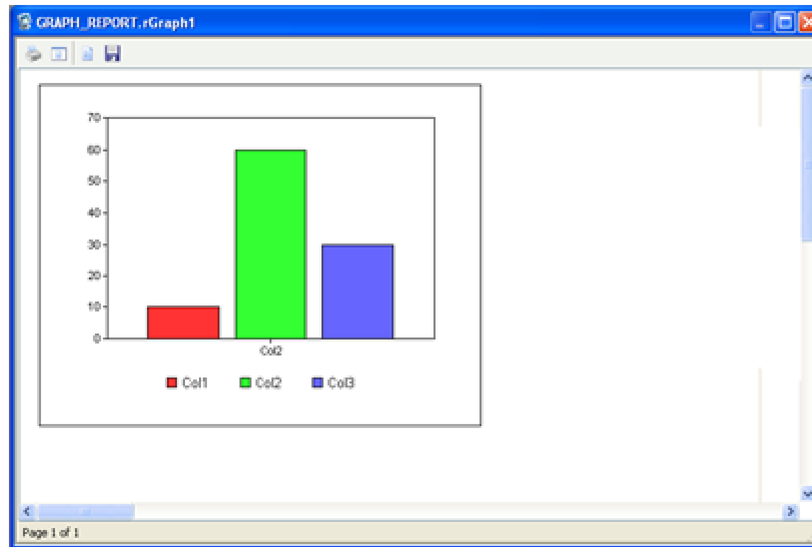


Figure 66:

construct the graph image in memory on the Omnis Server and 'save' and display it as a picture in the remote form. When the graph is constructed in memory, the $snapshot() method lets you transfer the graph image to an image variable suitable for displaying in an Omnis picture field in a remote form.

You have to create an Object variable in your remote form with the Graph object specified as the object subtype. The following method will do this:

```
# cGraphObj is an Object variable with Subtype of 'Graph2'
Do list.$define(Name,Sales)
Do list.$add('Andy',85)
Do list.$add('Sam',80)
Do list.$add('Liza',92)
Do list.$add('Harry',45)
Do cGraphObj.$majortype.$assign(kG2pie)
Do cGraphObj.$dataname.$assign(list)
Do cGraphObj.$maintitle.$assign("WebClient Pie Example")
Do cGraphObj.$snapshot(640,400) Returns iWebImage
```

This will result in this image:

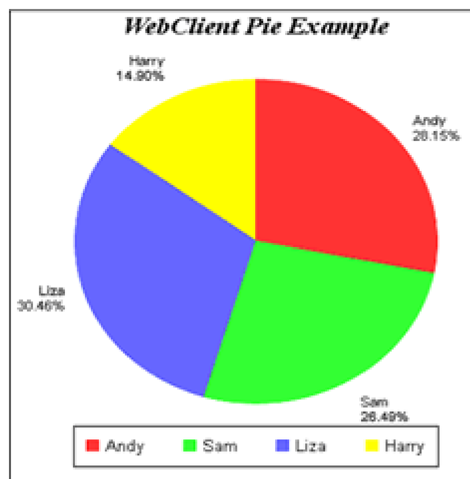


Figure 67:

The Graph2 example library contains a Remote form example, showing all the main chart types. In this case, the remote form uses an object class based on the Graph2 component and uses $snapshot() to transfer the graph image to a picture field in the remote form.

Drilldown in Web Client graphs

The remote form Picture field type reports the mouse X and Y co-ordinates when an evClick event is triggered. This allows you to pinpoint where in the image the user has clicked. You can use this information to find out which active graph element has been clicked, such as a bar, line or pie slice. Having captured the graph image using the $snapshot() method you can use the $findobject() method to return the item and set information for the selected object. For example:

```
# do code to construct the graph object, such as
Do list.$define(Name,Sales,Expenses)
Do list.$add('Andy',850,400)
Do list.$add('Sam',800,600)
Do list.$add('Liza',920,560)
Do list.$add('Harry',450,230)
Do cGraphObj.$majortype.$assign(kG2xy)
Do cGraphObj.$minorxytype.$assign(kG2xyBar)
Do cGraphObj.$dataname.$assign(list)
Do cGraphObj.$maintitle.$assign("Web Client Bar Chart")
Do cGraphObj.$snapshot(640,400) Returns iWebImage
Do $cinst.$objs.pic.$redraw
# graph image is transferred to remote form picture field and redrawn
```

The bar chart would look something like the following:

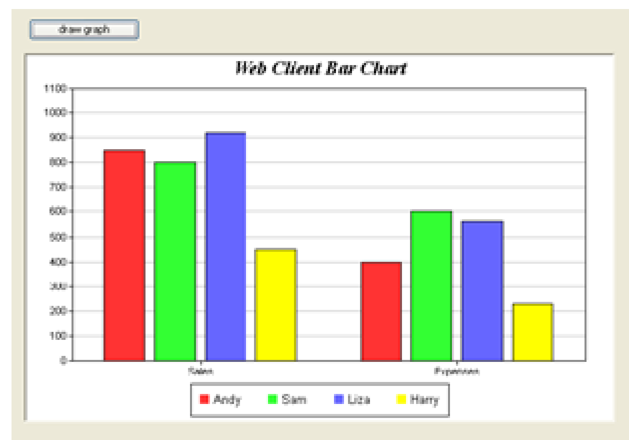


Figure 68:

```
# $event() method for remote form Picture field
On evClick
  Do cGraphObj.$findobject(pMouseX,pMouseY,setno,itemno,setname,itemname)
   Returns ok         ## true if successful
```

The mouse co-ordinates are returned in pMouseX and pMouseY in the evClick event. The $findobject() method uses the mouse co-ordinates to get the setno, itemno, setname, and itemname which contain information about the graph object clicked on. Clicking on the first bar in the second group of the above chart produces this data:

```
setno = 2, itemno = 1, setname = Expenses, itemname = Andy
```

You can use the item and set number/name information to drilldown into the data by reformatting the list data and redrawing the graph in the remote form.

Note that the Set number and name are not valid for pie charts.

Chapter 9—Remote Studio Applet

Remote Studio is an applet, which provides an equivalent of remote procedure calls to an Omnis server in order to provide functionality. The applet may reside either on the client or the server and can be utilized from many languages. For example, you could invoke it on your server from PERL or PHP, or invoke it on your client from VBScript, Delphi or .NET(C#, C or Visual Basic).

Remote Studio is available as COM or PHP objects so the possibilities are endless (Java support available in older versions has been removed). For example, you could write a Web Service using .NET, which uses the Remote Studio applet to provide functionality via SOAP requests; or you could simply use it from VBScript embedded in an HTML page.

How does it work?

The Remote Studio applet consists of two objects, both of which provide various methods and properties. The main object, the server object, lets you set up connection information; this information is very similar to the information required for an Omnis web client connection.

The second object, the variable object, provides functionality to get and set the variable information using variable types, which are native to the language. This object ensures that when the applet is translated to another language, such as Java, your code is virtually identical to all the other languages.

To use the applet, follow these steps:

- Create the server object
- Configure the connection information parameters (URL address, task name, etc.)
- Call the server object
- Destroy the server object

During the calling of the server object, the specified Omnis task will be constructed and the $event() method will be called with the method name and the parameters. Note that if the parameters are altered during the call, they will be updated in the calling code so in that sense they can be referred to as being “passed by reference”.

Object Interfaces

Server Object methods & properties

Method	Description	Parameters
Open	Opens the connection.	None
Close	Closes the connection	None
ErrorId	Obtain the last error id (0 = Ok)	None
ErrorText	Obtain the last error text string	None
Execute	Execute the specified remote method	Method Name, Parameter Array, Parame
ExecuteNoParams	Execute the specified remote method	Method Name
NewChar	Helper method to create a variable with a string	String
NewLong	Helper method to create a variable with a integer	Integer
NewBool	Helper method to create a variable with a Boolean	Boolean
NewVar	Helper method to create a variable	None

Note: All the above methods affect the error code and error text. Methods ErrorId() and ErrorText() should be used to verify success.

Property	Description	Type	Assignable
ServerURL	The URL of the server	String	Yes
OmnisClass	The name of the Omnis class	String	Yes
OmnisLibrary	The name of the Omnis library	String	Yes
OmnisServer	The name of the Omnis server	String	Yes

Property	Description	Type	Assignable
ServerScript	The server script text	String	Yes

Variable Object methods

Method Name	Description	Parameters	Return V
GetType	Get the variable type. The following types exist: varNone(0), varChar(1), varBinary(2), varPicture(3), varBool(4), varNumber(5), varList(6), varRow(7), varLong(8), varDateTime(9).	None	Integer
Clear	Clears/empties the variable content resulting in its type being varNone.	None	None
GetChar	Get the character data	None	String
SetChar	Set the character data	String	None
GetBool	Get the Boolean data	None	Boolean
SetBool	Set the Boolean data	Boolean	None
GetLong	Get the long integer data (32 bits)	None	Integer
SetLong	Set the long integer data (32 bits)	Integer	None
GetNumber	Get the number data	None	Double
SetNumber	Set the number data	Double	None
GetDate	Get the date / time data	None	Date
SetDate	Set the date / time data	Date	None
GetBinary	Get the binary data	Byte buffer, start from (0), buffer size.	Integer:
SetBinary	Set the binary data	Byte buffer, start from (0), buffer size	None
GetBinaryLen	Get the length of the binary data	None	Integer
SetRow	Creates a row	Integer column count	None
SetList	Creates a list	Integer row count, integer column count	None
GetRowCount	Get the row count	Integer	None
GetColumnCount	Get the column count	Integer	None
GetElement	Get the variable for the specified column & row (zero based). Note that this variable is a duplicate and you must use setElement if you wish to update the list/row contents.	Integer row, integer column	Variable
SetElement	Set the variable for the specified column & row (zero based)	Integer row, integer column, Variable Object	None

Note: Variable conversion between types will automatically occur, however if a variable cannot be converted then a data mismatch error will be called.

Studio Remote Tasks

You need to create an Omnis remote task class on the server to process the method requests. You should consider whether or not you need an instance of the remote task class to be constructed and destructed for each method call (most of the time you will not), therefore you will need to add a $canclose() method (see examples section) to the task.

The Remote Studio applet will add the following variables to the row parameter which is passed to the remote task instance during $construct() and $event() methods.

Parameter vars	Description
RSrval	RSrval is the row column, which contains the return value. Populate this value with the data that you wish to return to the caller.
RSname	RSname is the name of the method that should be invoked by the task instance.
RSparmNN	Where NN is the number of the parameter from 1 to N. RSparmNN are the parameters for the method. If these are modified then the variables in the server applet will also be modified.

Parameter vars	Description
MessageType	MessageType contains an integer to indicate the reason for the call. Currently the only valid values are:0 = Object is being closed by the caller1 = Method call

At the end of the $construct() and $event() methods you need to return the row parameter to ensure that the values are returned to the server applet.

Remote Studio Examples

Studio server remote task format

An example of a remote task class:

```
# Instance Var: iCanClose (Boolean)
# $construct method
# Parameter 1: pParams (Row)
Quit method $processmsg(pParams)
# $canclose method
Quit Method iCanClose
# $event method
# Parameter 1: pParams (Row)
On evPost
  Quit method $processmsg(pParams)
# $processmsg method
# Parameter 1: pParams (Row)
Switch pParams.MessageType
  Case 0 ## Object being closed by server
    Calculate iCanClose as kTrue
  Case 1 ## Method call
    # Use values in pParams to call method.
    # Note that this is a simple example and more work
    # is required for the parameters.
    Calculate iCanClose as kFalse
    Do method [pParams.RSname] Returns pParams.RSrval
End Switch
Quit method pParams
# TestRoutine method
# Sample method which can be called by server applet
# You should add more methods as required
Quit method "Test Return Value"
```

Use of the Server applet in Visual Basic

The following is an example of use in Visual Basic. To use this code in other languages such as .NET, VBA, Delphi you will need to change the syntax, but the structure should remain similar or unchanged.

```
Private Sub Command1_Click()
  Dim rsapp As RemoteStudio.Application
  Dim args(1) As RSvar
  Dim result As RSvar
  Rem Create applet & connect
    Set rsapp = New RemoteStudio.Application
    rsapp.omnisLibrary = "MyLibrary"
    rsapp.omnisClass = "rtMyTask"
    rsapp.omnisServer = "5000"
    rsapp.serverURL = "http://000.000.000.000" ## IP of your server
    rsapp.serverScript = "/cgi-bin/nph-omniscgi.exe"
```

```
    If (rsapp.open()) then
      MsgBox ("Error "+rsapp.errorText())
    End If
  Rem Setup parameters and call routine
    Set args(0) = New RSvar
    args(0).setChar ("Parameter 1")
    Set result = rsapp.execute("TestRoutine",args,1)
    If ( rsapp.errorId()) Then
      MsgBox ("Error "+Str(rsapp.errorId()) + ":" + rsapp.errorText() )
    Else
      MsgBox ("Method returned "+result.getChar())
    End If
  Rem Close applet
    rsapp.Close
End Sub
```

Use of the complex variables and Server applet in Omnis Studio

The following is an example of returning a complex variable type, in this case a list, in Omnis Studio. To use this code in other languages such as Visual Basic, .NET, VBA, Delphi you will need to change the syntax, but the structure should remain similar or unchanged.

```
# Server rtnVarList Method
Set current list myList
Define list {myCol1,myCol2}
Add line to list {("Row1 Col1","Row1 Col2")}
Add line to list {("Row2 Col1","Row2 Col2")}
Add line to list {("Row3 Col1","Row3 Col2")}
Do pParams.$cols.RSrval.$coltype.$assign(kList)
Calculate pParams.RSrval as myList
# Client Method
Do rsapp.$createobject()
Calculate rsapp.$omnislibrary as "RSTUDIO"
Calculate rsapp.$omnisclass as "rtRStudio"
Calculate rsapp.$omnisserver as "5112"
Calculate rsapp.$serverurl as http://127.0.0.1
Calculate rsapp.$serverscript as "/cgi-bin/nph-omniscgi.exe"
Do rsapp.$open()
Do rsapp.$executeNoParams("rtnVarList") Retuns result
Calculate rcnt as result.$getrowcount()
Calculate ccnt as result.$getcolumncount()
For row from 1 to rcnt step 1
  For col from 1 to ccnt step 1
    # To confirm with Visual Basic, offsets are zero based
    Calculate parm1 as result.$getelement(row-1,col-1)
    Send to trace log {[parm1.$getchar()]}
  End For
End For
```

Use of the Server applet in PHP

The following is an example of use in PHP: See previous example for the server method code.

Server rtnVarList Client Method

```
<?php
  include_once("rstudio.php");
  $rsapp = new RemoteStudio;
  $rsapp->mOmnisLibrary = "RSTUDIO";
  $rsapp->mOmnisClass = "rtRStudio";
```

```
  $rsapp->mOmnisServer = "5112";
  $rsapp->mServerURL = "127.0.0.1";
  $rsapp->mServerScript = "/cgi-bin/nph-omniscgi.exe";
  if ( $rsapp->open() )
  {
    echo "Port opened\r\n";
    $result = $rsapp->executeNoParams("rtnVarList");
    $lst = $result->getList();
    for ( $r=1; $r<=$lst->rowCount(); $r++ )
    {
      echo " Row $r = ";
      for ( $c=1; $c<=$lst->colCount(); $c++ )
      {
        $var = $lst->getElement($r,$c);
        echo $var->getCString() . " ";
      }
      echo "\r\n";
    }
    echo "Port closed\r\n";
    $rsapp->close();
  }
  if ( $rsapp->getErrorID() )
  {
    echo " Error ID :" . $rsapp->getErrorID() . "\r\n";
    echo " Error Message:" . $rsapp->getErrorText() . "\r\n";
  }
?>
```

Chapter 10—Automation

Automation, formerly called OLE Automation, is a technology that allows you to utilize an existing program's content and functionality, and to incorporate it into your own applications. Automation is based on the Component Object Model (COM). COM is a standard software architecture, based on interfaces, that is designed to separate code into self-contained objects, or components. Each component exposes a set of interfaces through which all communication to the component is handled.

For example, with Automation you can use a Word processor's mail merge feature to generate form letters from data in an Omnis database without the user being aware that Word processor is involved.

Automation consists of a client and a server. The automation client (Omnis) attaches to the automation server so that it can use the content and functionality that the automation server provides. Omnis can only be an automation client, it cannot be a server.

Automation servers consist of an object or a number of objects. For example, in the object hierarchy for Excel, the uppermost object in the Excel object model is the Application object. The Excel Application object has many children, two of which are Workbooks and CommandBars. Workbooks and CommandBars are collection objects that contain other objects. A Workbooks collection object contains Workbook objects and a CommandBars collection objects contain CommandBar objects. And, the list goes on, but understanding the object relationships in the automation server is fundamental to its use in Omnis and the third-party manufacturer can only provide the appropriate level of documentation for these relationships. See the Excel documentation for an illustration of the object hierarchy.

OLE2 Menu Options

Note that from Studio 11, the OLE2 menu options **Links, Object** and **Insert Object** have been removed from the **Edit** menu under Windows (these were used by OOLE2 which was removed in Studio 6.1).

Instantiating an Automation Server

In Omnis, before you can utilize automation servers, you need to get an object (in Excels' case you need to obtain an Application object).

You can obtain an automation object by:

- Creating an object variable with subtype set to the automation server that you require. Once created, you will have to invoke a standard method, such as $createobject(), to instantiate the server.
- Calling the OLE picture control $getobject() method and using the returned object variable.
- Querying an ActiveX controls' property, or calling a method, which returns an object variable.

Automation Server Functionality

An object by itself does nothing unless you can do something with that object. To programmatically examine or control an object, you can use the properties and methods that the object supports. A property is a function that sets or retrieves an attribute for an object. A method is a function that performs some action on an object.

For example, in Omnis, you can navigate to an object by starting at the uppermost object and working your way down to your target. Consider Excel and its Workbooks collection object, which represents all the open workbooks. You could use its Count property to acquire the count of workbooks open in Excel:

```
lNum = objectApplication.$Workbooks().$xcount
```

You may notice that the above example references a property called xcount whereas the Microsoft Excel automation documentation refers to the property as count. This is because, Omnis already has $count as a core internal notation attribute and has therefore appended **"x"** as a prefix. Unfortunately, other automation properties and methods also clash with Omnis internal notation but have not been renamed. To utilise the automation attributes simply add two colons before the name. So $xcount becomes $::xcount. Adding two colons even when they aren't required will not have an effect, so it is a good habit to get into.

Another example would be to enquire on a particular value in a worksheet:

```
CValue = objectApplication.$Workbooks("Book1.xls").$Worksheets("Sheet1").Range("A1").Value
```

If you require a particular object often, it is beneficial to assign the automation object to a Omnis object variable. To do this simply, create an object variable, with an empty subtype, and then invoke the automation method/property and assign the results. For example:

```
Do objectApplication.$Workbooks("Book1.xls").$Worksheets("Sheet1") returns objWorkSheet
Do objWorkSheet.$Range("A1").$value.$assign("Cell Value")
```

So, how exactly, do you know what automation objects have what properties and methods?

- Obviously the best source of documentation is the third-party, which developed the automation server. For example, for Microsoft Office software, you can refer to the vbaxl8 (Excel), vbagrp8 (Graph), vbaoff8 (Office), vbaoutl (Outlook), vbappt (Powerpoint), vbawrd8 (Word), help files.
- Use an automation object browser such as the OLE/COM Object viewer, which is included with Microsoft Studio.
- Use the Omnis Interface manager and Values list.
- And lastly, many Microsoft Applications come with a Macro recorder, which enables you to record user actions as a Visual Basic for Applications, or VBA, script.

To illustrate this, follow these simple steps: -

- Start Microsoft Word.
- On the Tools menu, click Macro, and then select Record New Macro. In the Store Macro In drop-down box, select the name of the active document. Make note of the new macro's name, and then click OK to start recording.
- Start a new document.
- Type one and press ENTER.
- Type two and press ENTER.
- Type three.
- On the File menu, click Save, and save the document as C:\doc1.doc

- Click the Stop Recording button (or, on the Tools menu, click Macro and then Stop Recording).

To view the VBA code that the macro recorder generated from your actions, on the **Tools** menu, click **Macro**, and then click **Macros**. Select the name of the new macro in the list and click **Edit**. The Visual Basic Editor displays the recorded macro.

```
Documents.Add
Selection.TypeText Text:="one"
Selection.TypeParagraph
Selection.TypeText Text:="two"
Selection.TypeParagraph
Selection.TypeText Text:="three"
ActiveDocument.SaveAs FileName:="Doc1.doc",
  FileFormat:=wdFormatDocument, _
  LockComments:=False, Password:="", AddToRecentFiles:=True,
  WritePassword:="", ReadOnlyRecommended:=False,
  EmbedTrueTypeFonts:=False, SaveNativePictureFormat:=False,
  SaveFormsData:=False, SaveAsAOCELetter:= False
```

Whilst, this isn't valid Omnis script code, it does illustrate the automation properties and methods, and usually it is simply a case of making a few minor modifications such as adding the **$** prefix to the names and appending the parenteses to methods.

Built-in Methods

Along with the automation server's methods, Omnis provides additional methods to enable you to manage the server.

These methods are: -

- **$createobject()** creates an instance of the object and may be called before the automation server can be used.
- **$getactiveobject()** obtains an instance to a current automation object.
- **$getobject(Filename,[class])** creates an instance of an object from the specified filename and class, if supplied. For example $getobject("C:\CAD\SCHEMA.CAD")
- **$isavailable()** returns kTrue if the object variable has a server instance, or kFalse otherwise.

Whether these methods exist or not, will depend on the source of the object variable. For example, an object, which originated from an ActiveX, OLE2, or an automation collection, will not have these methods.

Lifetime of an Automation Server Instance

The lifetime of an automation instance typically follows these steps:

- Create an instance of the object using $createobject, $getactiveobject or $getobject. And validate that an instance was created by calling $isavailable.
- Communicate with the object via properties and methods.
- Finally, terminate the object. Many automation servers provide a method called **$quit** which should be used to ensure that the instance is terminated.

Terminating Processes

The $quit() method of an Automation Object has an optional parameter 'hwnd' to allow you to terminate the object's process. For example:

```
Do ExcelObj.$quit(ExcelObj.$hwnd)
```

will terminate the automation object's process once the QUIT method finishes.

Automation Event Handling

Some automation servers fire events, for example, an email application may fire an email alert event to signify new emails. In Omnis, it is possible to intercept COM automation events (ActiveX and OLE2 events are handled differently). To receive automation server events, you need to add the "enableEvents" item to the "ole2auto" section of the config.json file and set it to true:

```
"ole2auto": {
  "enableEvents": true
},
```

To intercept events, you simply sub-class the automation server object and override the desired events. The event method name will have the suffix '**event**' added.

For example, the following steps illustrate the trapping of events for an automation server called **UrlReader**, which is available from Microsoft.

- Before you start Omnis, ensure that you have used regsvr32 (a DOS program which is distributed by Microsoft and used to register automation servers) to register the UrlReader server.
- Create a new object, which is subclassed from the automation object '**UrlReader.UrlReader.1**'. To do this either, use the Class Wizard 'New Sub-class object' from the Studio Browser and select "Automation>>UrlReader.UrlReader.1" from the list or create an object and change the superclass property to "Automation>>UrlReader.UrlReader.1" using the superclass dialog.
- Now open the method editor for the new object. You will notice numerous methods for the UrlReader object. Two of which will have the suffix 'event'. Automation events have the extension "event" appended by Omnis. Now override both event methods and add the code 'Send to tracelog Event triggered', then close the method editor.
- Create a new window and add a button.
- Open the method editor for the window and add an object variable called **myObj** with the subtype that you named your object (in the first stage).
- Add the following code:

```
$construct:      Do myObj.$createobject()
$destruct:       Do myObj.$quit()
Button evClick:  Do myObj.$readurl(http://www.microsoft.com/,"c:\mscom.txt")
```

- Open the window and the tracelog. Then press the button. Notice that the tracelog will contain "Event triggered" text.

Automation to Omnis Variable Conversion

An advanced topic is the differences between variable types in automation and in Omnis. This topic becomes less and less relevant as the automation servers become more and more flexible; unfortunately, some of the older automation servers are rather inflexible when it comes to the type of variables used.

Automation has a parameter type called VARIANT; this type can hold any type of data. Unfortunately with this flexibility comes a price. Namely that some objects, Excel for example, state that they handle any type of data (ie VARIANT) in theory, but in fact they may be expecting data passed by reference or of a limit subset of the VARIANT types.

The Omnis automation component takes the same approach as Visual Basic, in that everything is passed by VARIANT, and depending on the Omnis data type used, certain assumptions are made.

All these assumptions can be over-ridden; the table below shows the default conversion.

Omnis DataType	Automation DataType
Boolean	VT_BOOL
Integer (0 to 255)	VT_I1
Integer (Long)	VT_I4
Number	VT_R8

Omnis DataType	Automation DataType
Character	VT_BSTR
List	VT_ARRAY
Row	VT_ARRAY
Binary	VT_ARRAY | VT_UI1

Typically a server will return the error code of 80070057 (see Automation errors section) if the parameter wasn't of the correct type. You can coerce variables to another datatype, by preceding the parameter with a constant listed in the table below:

Constant Name	Automation DataType	Datatype description
KAutoBOOL	VT_BOOL	Boolean Value (True or false)
kAutoI1 / kAutoUI1	VT_I1 / VT_UI1	Signed / Unsigned Byte
kAutoI2 / kAutoUI2	VT_I2 / VT_UI2	Signed / Unsigned short
kAutoI4 / kAutoUI4	VT_I4 / VT_UI4	Signed / Unsigned Long
kAutoR4	VT_R4	4 byte real
kAutoR8	VT_R8	8 byte real
kAutoBSTR	VT_BSTR	Binary String
kAutoDISPATCH	VT_DISPATCH	IDispatch *
kAutoCY	VT_CY	Currency
kAutoEMPTY	VT_EMPTY	Empty
The above constants+REF	VT_xxxREF	By Reference
kAutoNULL	VT_NULL	Null

For example:

```
Do object.$setvalue(kAutoI4,"45")
```

Automation Errors and Limitations

Should an automation error occur, then the contents of **#ERRCODE** and **#ERRTEXT** can be used to isolate the problem.

On an error, #ERRCODE will be set to the automation error code type, HRESULT, which is a 32bit unsigned integer. A value of –1 indicates that the error occurred in the automation component rather than in the automation server, for example, "Automation method not found".

#ERRTEXT will contain a string representation of the error.

HRESULT codes are difficult to document as they can be defined by both the server application and by the operating system. However, here are a few of the more common codes (which are in hexidecimal):

8000FFFF	Unexpected error
80004001	Not implemented
8007000E	Out of memory
80070057	Invalid argument
80004002	No such interface supported
80004004	Operation aborted
80004005	Unspecified error
800401F3	Invalid class string

An automation limitation is Constants, or automation enums, which are not supported for COM objects and OLE2 objects.

Automation Examples

Some of the best examples of automation in Omnis are contained within the automation sample library, but the following illustrate the use of XML, DAO and Outlook in Omnis.

XML

This example requires the file **books.xml**, which is available from Microsoft. Ensure that you obtain the correct version, which have "AUTHOR" tags, in the correct case; otherwise you will have to subsitute the "AUTHOR" tag with another tag in the example code.

The variables xml (type Object and subtype 'Microsoft.XMLDOM.1.0') and element (type Object, no subtype) need to be added to your code.

```
Do xml.$createobject()
Do xml.$async.$assign(kFalse)
Do xml.$load("c:\books.xml")
Do xml.$getelementsbytagname("AUTHOR") Returns element
For #1 from 0 to element.$length-1 step 1
  Calculate #S1 as element.$item(#1).$xml
  OK message {[#S1]}
End For
Do xml.$quit()
```

This example loads the xml file and enumerates each AUTHOR tag.

DAO

This example requires the Microsoft Office sample database **northwind.mdb**. The variables obj (type Object and subtype 'DAO.DBEngine.36') and database, recordset, both of type Object (no subtype).

```
Do obj.$createobject()
Do obj.$Workspaces(0).$opendatabase("c:\office\northwind.mdb") Returns dat
Do dat.$openrecordset("Select * from Products",4) Returns recordset
Calculate #1 as recordset.$fields().$xcount
For #2 from 1 to #1 step 1
  Calculate #S1 as recordset.$fields(#2-1).$::value
  Calculate #S2 as recordset.$fields(#2-1).$::name
  Send to trace log {[#S2]=[#S1]}
End For
Do recordset.$close()
Do dat.$close()
Do obj.$quit()
```

This example opens the database file and enumerates each product in the record set.

Outlook 2000

This examples requires variables ol (type Object and subtype 'Outlook.Application.9') and variables olns, objFolder, ocontacts, and ocontact (all of type Object and no subtype).

```
Do ol.$createobject()
Do ol.$getnamespace("MAPI") Returns olns
Do olns.$getdefaultfolder(10) Returns objFolder
Do objFolder.$items() Returns ocontacts
OK message {There are [ocontacts.$xcount] contacts}
Do ocontacts.$getfirst() Returns ocontact
For #1 from 1 to ocontacts.$xcount step 1
  Calculate #S1 as ocontact.$xfullname
  OK message {[#S1]}
  Do ocontacts.$getnext Returns ocontact
End For
```

This example enumerates the contacts in your Outlook application.

Chapter 11—Apple Events

All the Apple Event commands, including *Send core event* and *Send Finder event,* are obsolete and have been removed from the Method Editor in Omnis Studio 10 and above. You can use the oFinderEvent object class to call Apple Finder events, as described below.

Note that in Omnis Studio 8.0.3; Apple Events do not work on macOS Sierra and are therefore no longer supported in that release of Omnis Studio. The commands have been moved from the Apple events... group and placed into the *Obsolete* commands group in the Method Editor and are no longer supported.

Apple Events Object

To replace the functionality of the old “Send Finder Event” commands, this release includes a new Object class called **oFinderEvent** which contains a number of methods which run AppleScript to execute the equivalent Apple Finder events, such as a *Get File Info* event or a *Duplicate Files* event. The AppleScript is run using the $runapplescript() Omnis method from inside each method in the object class.

To use the object class and these methods, click on the Class Wizard option in the Studio Browser, then click on Object, select the oFinderEvent option (available on macOS only), name the object class (or keep the name oFinderEvent) and press Return: a copy of the object class template is added to your library. Open the Method Editor for the class in which you want to use the Finder events (such as a window, menu or toolbar class), and then create an **Object variable** in the class, setting its subtype to the oFinderEvent object you created.

Apple Event Methods

You can call the methods in your code, and run the AppleScript as required, using the Omnis command *Do ObjVar.$methodname()* using the appropriate method name, as below.

Some of the methods can take a file path as the first parameter, or if this is omitted or empty a file selection dialog will open. The title of the dialog can be customized by editing the cOpenFilesTitle class variable.

- **$getfileinfo**([cFilePath])
  Sends a Get File Info event: equivalent *Send finder event {Get File Info}* command
- **$duplicatefiles**([cFilePath])
  Sends a Duplicate Files event: equivalent *Send finder event {Duplicate Files}* command
- **$makealiasforfiles**([cFilePath])
  Sends a Make Alias For Files event: equivalent *Send finder event {Make Alias For Files}* command
- **$openfiles**([cFilePath])
  Sends a Open Files event: equivalent *Send finder event {Open Files}* command
- **$printfiles**([cFilePath])
  Sends a Print Files event: equivalent *Send finder event {Print Files}* command
- **$revealfiles**([cFilePath])
  Sends a Reveal Files event: equivalent *Send finder event {Reveal Files}* command
- **$emptytrash**()
  Sends a Empty Trash event: equivalent *Send finder event {Empty Trash}* command
- $**restart**()
  Sends a Restart Macintosh event: equivalent *Send finder event {Restart Macintosh}* command
- **$shutdown**()
  Sends a Shutdown Macintosh event: equivalent *Send finder event {Shutdown Macintosh}* command
- **$sleep**()
  Sends a Sleep Macintosh event: equivalent *Send finder event {Sleep Macintosh}* command

The object has three instance variables which you can use in your code to handle errors:

- **iErrCode**
  The error code generated by the last command. 0 for no error.
- **iErrText**
  The error text generated by the last command.
- **iScript**
  The AppleScript sent by the last command.

The following legacy commands are not supported in the latest version on macOS: Send finder event {Show About}, Send finder event {Share Files}, Send finder event {Show Clipboard}.

You can examine the Omnis code and AppleScript in each method inside the object class. For example, various simple operations are handled in a generic method $simpleop and the operation is passed in as a parameter:

```
# $simpleop method
# pOperation param receives 'Empty', 'Restart', 'Shut down', or 'Sleep' msg
Begin text block
  Text: tell application "Finder" (Carriage return)
  Text: [pOperation] (Carriage return)
  Text: end tell (Carriage return)
End text block
Get text block iScript
Do $root.$runapplescript(iScript,iErrCode,iErrText)
Quit method iErrCode
```

Each of the new methods in the object class includes the equivalent old command as a comment to help you map your code to the new methods.

```
# Send finder event {Empty Trash} ## old command
Quit method $cinst.$simpleop("Empty") ## new method
```

Chapter 12—Omnis ODBC Driver

The Omnis ODBC Driver is available on Windows and macOS platforms and enables read-only access to Omnis data files. You can use it to import data into ODBC-compliant applications such as Microsoft Access, Excel and Word using the Microsoft Query tool.

There are also separate versions of the ODBC Driver available for Unicode and non-Unicode/legacy data files (Omnis 7 & pre-Studio 4.3.2).

Enable ODBC Access

Before you can access an Omnis data file via ODBC, you need to add one or more *ODBC users* to the data file. These user names and passwords are used to authenticate users attempting to connect to the data file. Each user also has an associated ODBC access mask value which is used to enable or disable access to one or more of the tables inside the data file. Therefore, each user can have customized access to certain tables whilst being denied access to others.

Select Tables

You can find the ODBC Admin tool in Omnis Studio by selecting the
Tools -> Add-Ons->ODBCAdmin… menu option.

Press **Open Data File** to proceed, then select the *Tables* tab.

This is where you select which tables will be visible via ODBC. Each table has an associated 32-bit access mask; shown as two banks of 16 checkboxes.

The choice and assignment of groups is entirely user-defined, but a table must be a member of one or more groups to permit access.

Select (single-click) each table in turn and select one or more of the checkboxes, e.g.

Figure 69:

Figure 70:

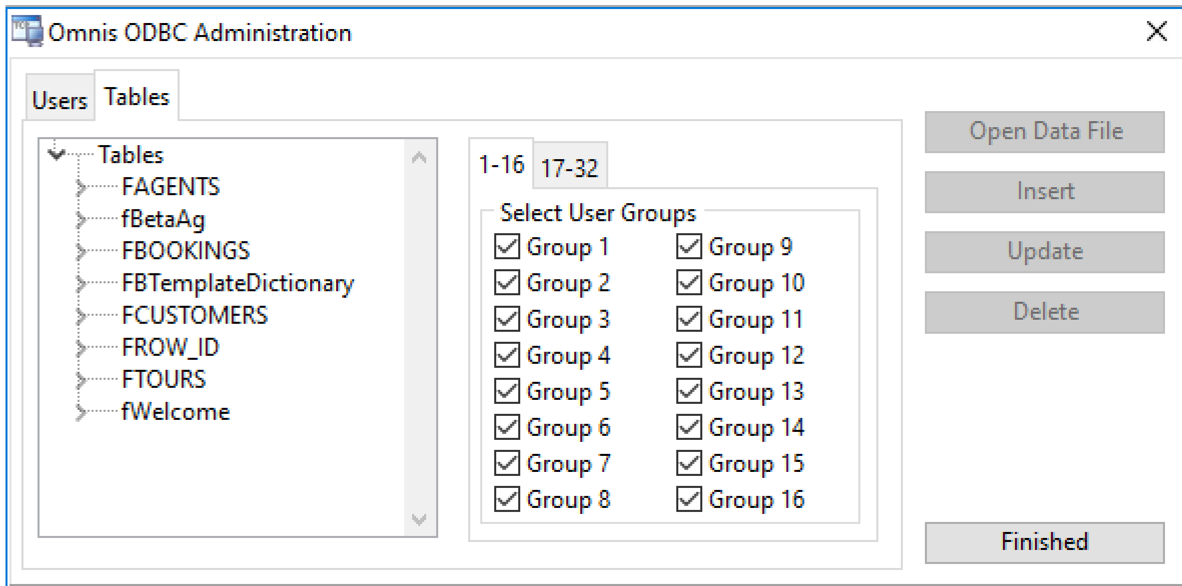


Figure 71:

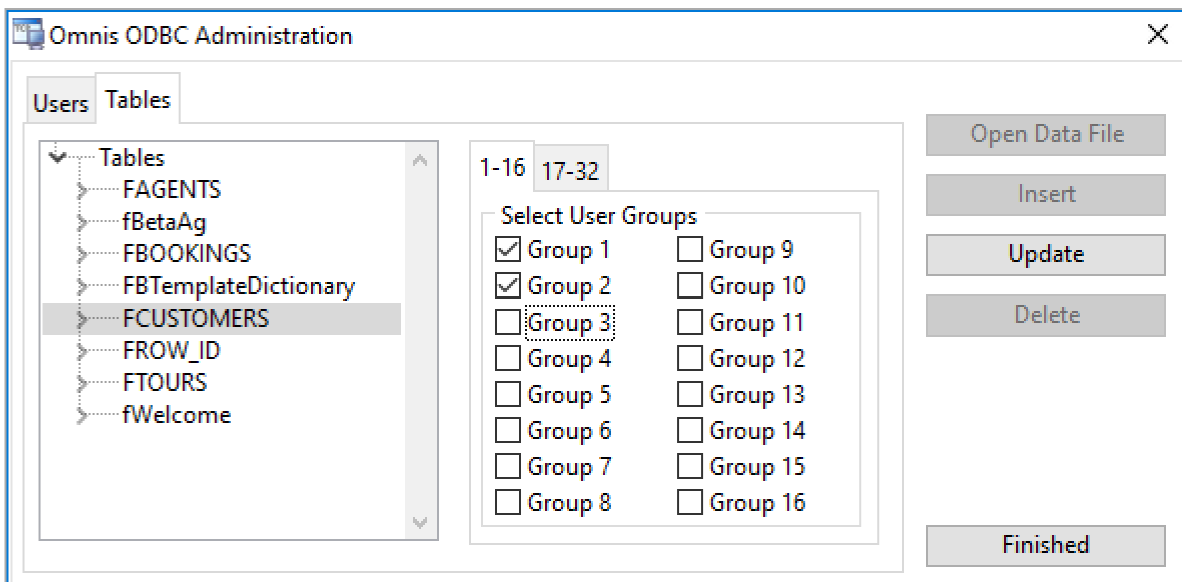


Figure 72:

When you enable a table for ODBC access, all of its columns are enabled by default. You can further restrict access to a table's individual columns by expanding the tree list on the left, or by double-clicking the table name. Select and uncheck the group box(es) to hide individual columns.

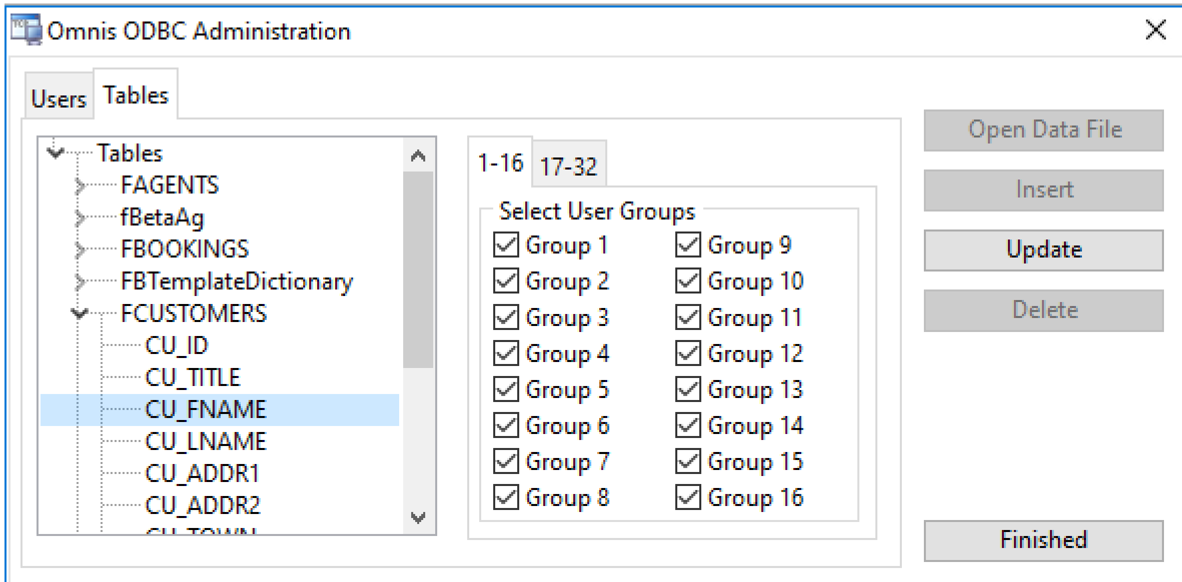


Figure 73:

At each stage, and before leaving the *Tables* tab, press **Update** to save your changes.

Select Users

Now that one or more tables have been enabled for ODBC access, click on the *Users* tab. If one or more users already exist for this data file, they will be shown on the left. Otherwise, enter a new user name in the *Current User* field to create a new one, add a password then press **Insert** to save changes.

For existing users, select the user name on the left, modify the fields as required, then press **Update** to save changes.

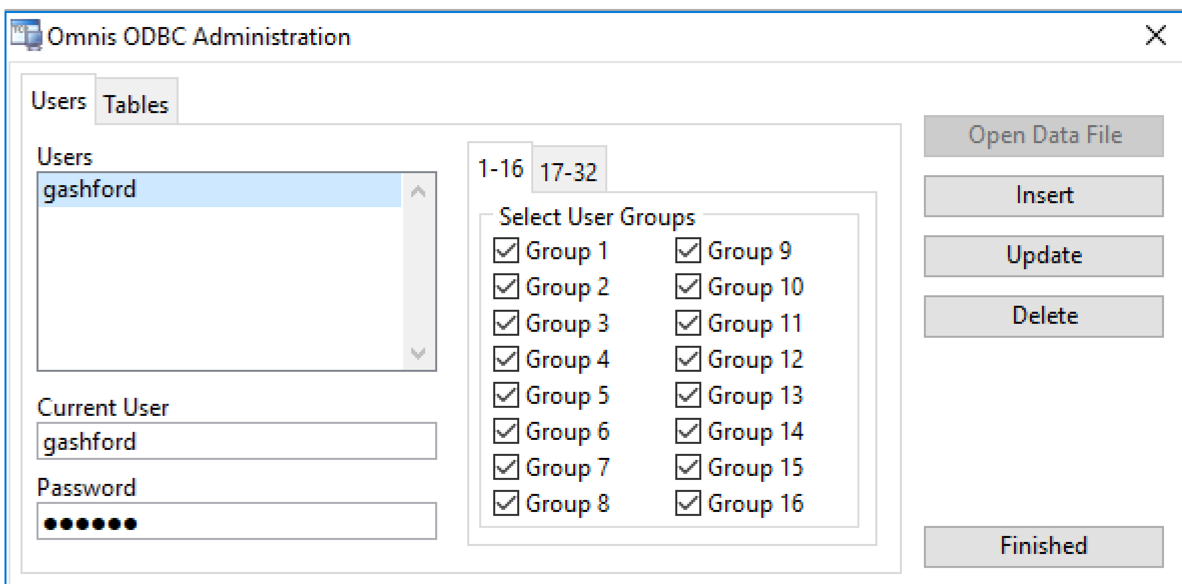


Figure 74:

For each user, select one or more group checkboxes. These correspond to the groups assigned to your tables in the previous step. Therefore, a given user will only have access to tables in the selected groups.

If the FCUSTOMERS table in the above example belongs to Group 1 only, the user needs to be a member of Group 1 in order grant access to that table.

If the FBOOKINGS table belongs to its own group, e.g. Group 3, a user must have group 3 selected in order to see it. Similarly, a user that has both groups 1 and 3 selected will be able to access both tables.

The categorization of tables into different access groups and users that have access to those groups is entirely at your discretion. As a minimum, or for basic access to all tables by all users, you can assign all tables to Group 1 (for example), then assign all users to Group1.

Once you have completed your table groups and user group assignments, press **Finished** to close the ODBC Admin tool. ODBC user information is stored in a (hidden) table named ODBC_USERS. (This table is normally accessible only if you retain access to the data file using Omnis Studio and the OmnisSQL DAM.)

Download and Install the Driver

Having added one or more ODBC Users to your Omnis data file, you can now download and setup the Omnis ODBC Driver.

You can download the latest version of the ODBC driver from the Omnis website at: https://www.omnis.net/developers/resources/download/tools/odbcdriver.jsp

Once downloaded, run the installer and follow the steps in order to add the driver to the system.

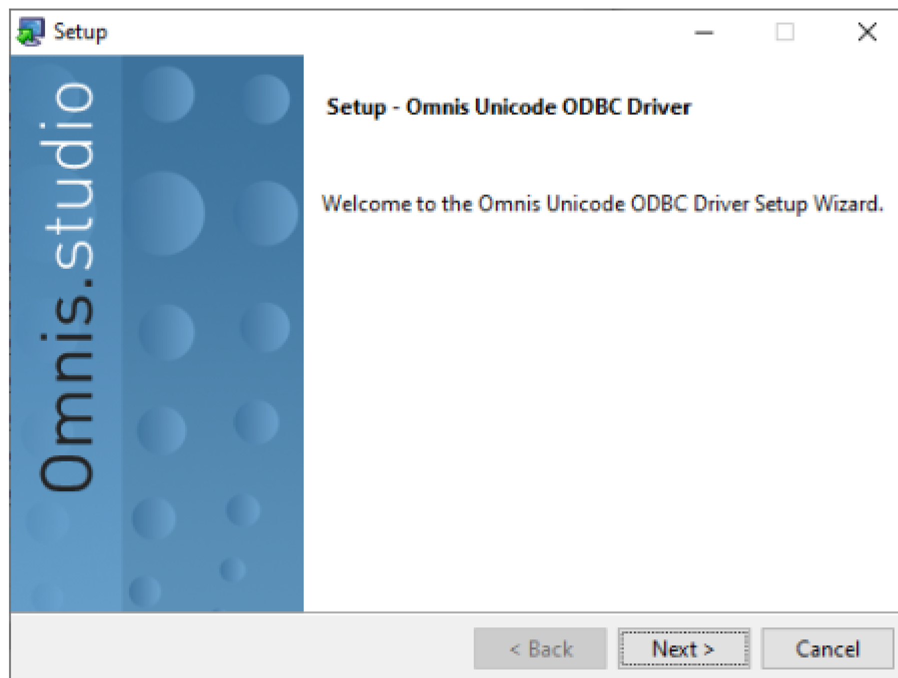


Figure 75:

Configure ODBC DSNs

Once installed, you can add one or more ODBC Data Source Names (DSNs); one DSN is needed for each Omnis data file that you need access to.

On Windows, you can use the 64-bit ODBC Administrator to configure a DSN that uses the 64-bit ODBC Driver (for use with 64-bit apps) or the 32-bit ODBC Driver (for use with 32-bit apps). Both drivers are provided during installation.

Press the Windows key (□) then type "ODBC" in order to locate and run the ODBC Administrator.

Press **Add** to create a new *User* or *System* ODBC DSN dependent on which tab is currently selected. Select the Omnis ODBC driver that is compatible with your Omnis data file. For data files used with recent versions of Omnis Studio, this will normally be the **Omnis Unicode ODBC Driver (64-bit)**. The choice of 32 or 64-bit depends on the third-party application that you will be using. 32-bit apps may not be able to see ODBC DSNs defined using the 64-bit ODBC Administrator and vice-versa.

Now select the Omnis data file to be accessed or enter the path name directly into the entry field.

The user name and password you add here must correspond with a name that you added to the data file previously using ODBC Admin tool.

Press **OK** to complete setup of this ODBC DSN.

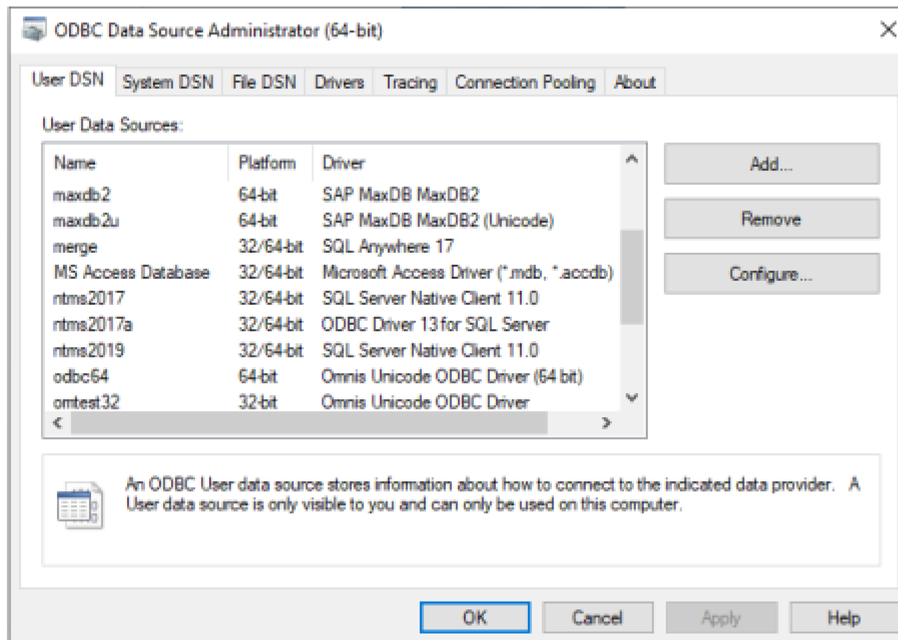


Figure 76:

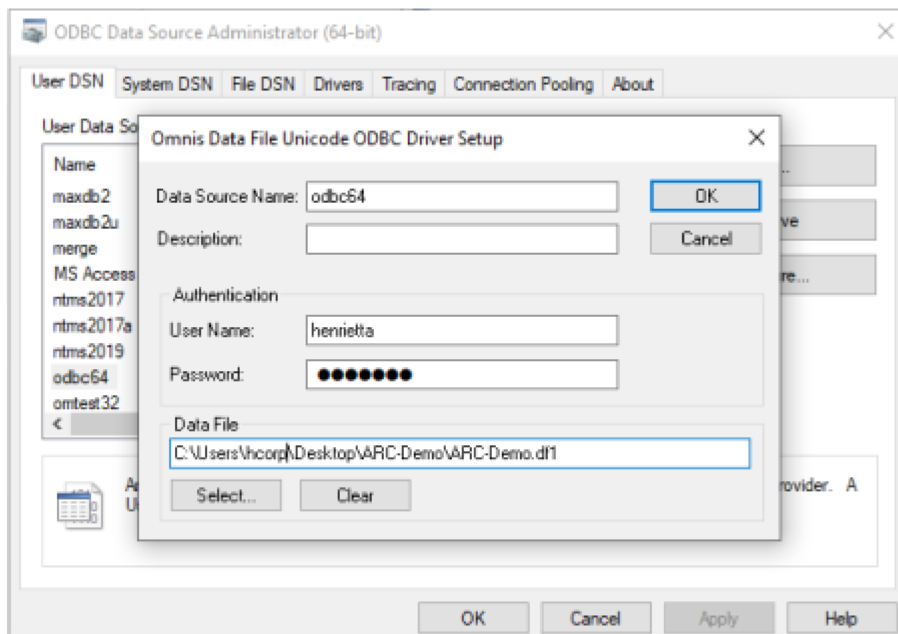


Figure 77:

Testing the DSN

You can either use Omnis Studio or your third-party application (such as Microsoft Query/ Excel) to test your ODBC DSN.

Using Omnis Studio

From Studio, you can use the SQL Browser by setting-up an ODBC session to the ODBC data source.

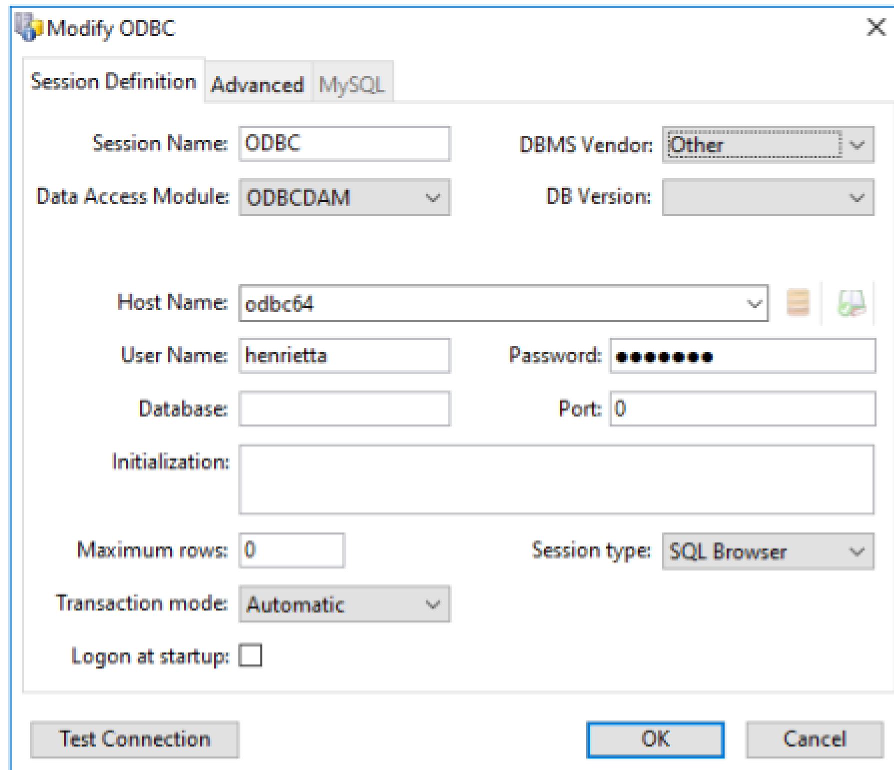


Figure 78:

Press **Test Connection** to verify the connection details.

Using Excel

Using Microsoft Excel, go to the **Data -> From Other Sources -> From Microsoft Query**.

Select the ODBC data source name created in the previous step, then select the table and (optionally) columns that you want to import.

Selecting the table, then pressing **>** will select all visible columns. Press **Next** to select additional options and to choose how/where to return the columns. You can return the results to Microsoft Query or import the results directly into your spreadsheet.

Other Apps

Other ODBC-compliant applications provide different ways to import data via ODBC. Please refer to the documentation accompanying the application for specific details.

The only caveat is that the application must be compatible with *read-only* operation.

Using SQL

The Omnis ODBC driver recnognizes a limited subset of the SQL programming language. Tools like Microsoft Query handle the SQL syntax for you, but if you want to modify the syntax of the SQL SELECT query, please refer to the Omnis Programming Manual where the OmnisSQL language definition is explained in more detail.

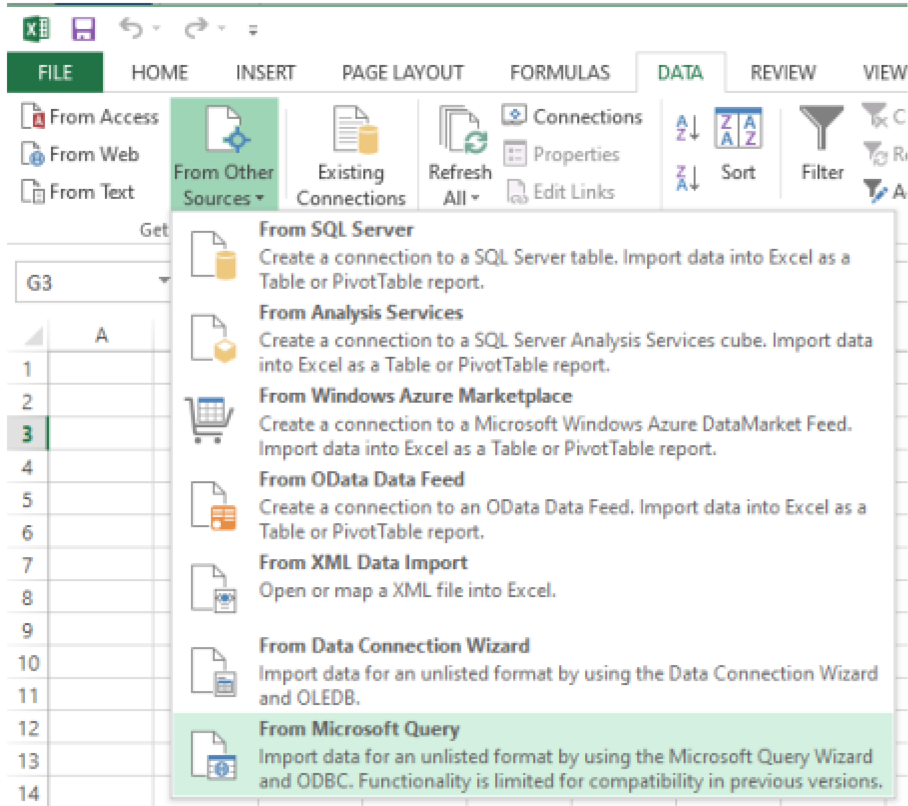


Figure 79:

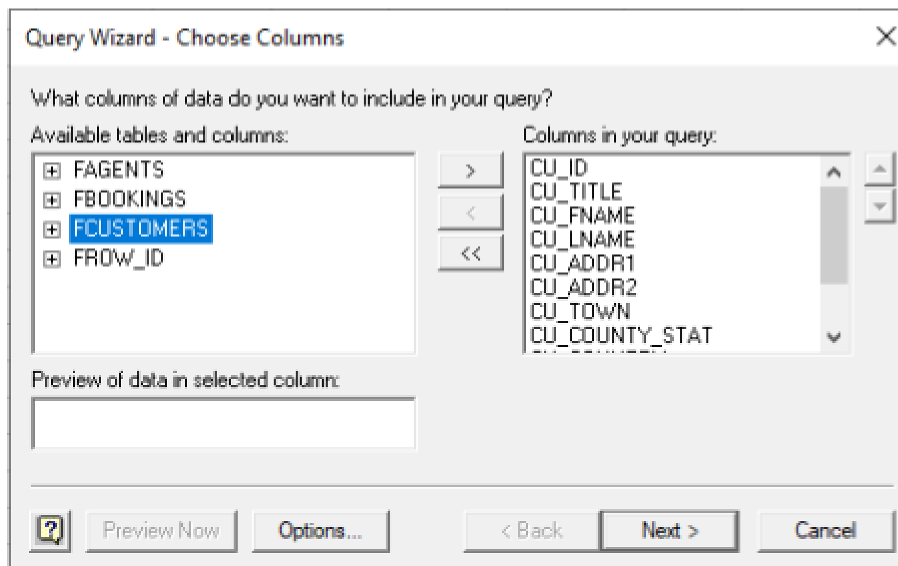


Figure 80:

Chapter 13—Blowfish Encryption

You can use the **Blowfish** external component to integrate encryption into your Omnis Studio applications. Alternatively, you can use the CRYPTO Worker Object to perform encryption and decryption of data using the AES, Camellia, and DES encryption types.

About Blowfish

Omnis Studio supports the *Blowfish* encryption algorithm via a non-visual **External Object** which you can use in code to provide a layer of security. Blowfish is a fast and freely available encryption algorithm created by Bruce Schneier.

The Blowfish external object contains methods to encrypt and decrypt, as well as initialize the encryption object using an initial key. To create an encryption object, you should create an object variable and specify the Blowfish object as the subtype of the variable. You must initialize the blowfish object with a variable length key using the $initkey() method. For example:

```
# create Object var 'blowfish' of Blowfish subtype
Do blowfish.$initkey("MyKey")
```

To encrypt binary data use the following method:

```
Do blowfish.$encrypt("MyData"[,bAddLenHeader=kFalse])
```

Optionally adds a length header if bAddLenHeader is kTrue and then returns the binary encrypted data. The object can use the length header to restore the data length when decrypting the data. The header is 8 bytes.

To decrypt data use the following method:

```
Do blowfish.$decrypt("MyData"[,bHasLenHeader=kFalse])
```

Decrypts the data and returns the binary result. If bHasLenHeader is kTrue, the object strips the length header and uses it to set the length of the returned data.

You can also use the following methods to encrypt or decrypt Character data:

```
Do blowfish.$encryptchar("cData"[,bAddLenHeader=kFalse])
# or to decrypt
Do blowfish.$decryptchar("cData"[,bHasLenHeader=kFalse])
```

Padding

The $padding property allows you to specify the type of padding to use when encrypting data.

- **$padding**
  A kBlowFishPadding… constant that indicates the type of padding to use when encrypting or expect when decrypting (default kBlowFishPaddingNone). A value other than kBlowFishPaddingNone is ignored if you specify a length header.

Valid values of the padding constant are kBlowFishPaddingNone (use or expect no padding) and kBlowFishPaddingPKCS5 (use or expect PKCS5 padding).

The presence of PKCS5 padding allows the code decrypting the data to correctly restore its length, without requiring the non-standard length header. This allows the BlowFish object to be used to encrypt data to be passed to applications other than Omnis – these applications (assuming they have the key) can decrypt the data and set its length correctly.

Binary Encryption

You can use the encxtea() and decxtea() functions to encrypt and decrypt binary data. The functions are found in the Binary Field group and use the eXtended Tiny Encryption Algorithm (XTEA) to encrypt the data.

- **encxtea**(*binary*,*key*)
  Returns the *binary* result of encrypting *binary* using the eXtended Tiny Encryption Algorithm (XTEA) with the binary *key*; the key must be 128 bits long.
- **decxtea**(*binary*,*key*)
  Returns the *binary* result of decrypting *binary* (previously encoded using encxtea()) with the binary *key*; the key must be 128 bits long.

String Encoding

You can encode and decode a string using the encstr() and decstr() functions. You can supply a code or Omnis supplies a default key. The functions are found in the String group of functions.

- **encstr**(*string*[,*key*])
  Encodes the *string* using the *key*. If omitted, Omnis uses its default value for the *key*. The return value of **encstr()** is a string that is difficult to decode without knowing the key. To decode the string, and return the original value, use the *decstr()* function.
- **decstr**(*string*[,*key*])
  Decodes the *string* (previously encoded using encstr()) using the *key*. If omitted, Omnis uses its default value for the *key*. Note that decstr(encstr(string,key),key) = string.

```
Calculate lEncoded as encstr('Testing',10)
Calculate lDecoded as decstr(lEncoded,10)
# returns the original string 'Testing'
```