# Chapter 3—Omnis Programming

Omnis Studio has a powerful programming environment that lets you create almost any type of enterprise or web application. The Omnis programming environment contains hundreds of 4GL commands and functions, as well as a low-level scripting language, called *Omnis Notation*, that allows you to manipulate objects dynamically in the runtime environment. To program in Omnis, you must consider the following things:

1. **Variables**  
   *variables* are the principal data container in Omnis: most objects in Omnis can contain variables, but their *scope* and the kind of *data* they can contain depends on the *type* of variable: you can use the method editor to add variables to an object: the definition and dynamic manipulation of variables is at the heart of programming
2. **Methods**  
   *methods* are pieces of Omnis code contained within the objects in your application, each performing a particular operation specific to the object or the application as a whole: creating and modifying methods in your library is key to creating an application.
3. **Events**  
   almost all user actions in Omnis generate an *event*: when an event occurs a *message* is sent to the object in which the event occurred: you need to write methods behind the objects in your library to handle the events.
4. **Tasks**  
   when your application runs in Omnis many object instances are created, such as windows, reports, and web forms: these instances are opened and handled within a *task*. Omnis creates a default task, but you can create your own tasks that allow you to handle the objects in your application.

All the above topics are covered in this chapter. You create and modify methods in Omnis using the *method editor*.

## Variables

Variables can hold different types of data and are visible in different parts of your application depending on their data type and scope. For example, if you create a variable of list data type in a window class, the list variable and hence its data is visible within the window class and all its instances, but is not accessible elsewhere in your library. The data types available in Omnis are described in detail in the *Libraries and Classes* chapter.

### Declaration and Scope

A variable may be global, accessible from all parts of your application, or it may have its scope restricted to certain areas so that it cannot be referred to from elsewhere. By declaring variables in the proper scope, you limit the potential for arbitrary connections across your application and thus reduce the potential for error and the complexity of your application.

The following table lists the different kinds of variables and their scope. It also shows when they are initialized and destroyed.

|  |  |  |  |
| --- | --- | --- | --- |
| Variable | When Initialized | When Destroyed | Scope |
| Parameter | on calling the method | returning to the calling method | the recipient method |
| Local | on running the method | on terminating a method | the method |
| Instance | on opening an instance | on closing the instance | a single instance of a class |
| Class | on opening a library | on clearing class variables or closing a library | the class, and all instances of the class |
| Task | on opening an instance of the task | on closing the task instance | the task, and all its classes and instances |
| Hash | on starting Omnis | on quitting Omnis | global |

Apart from hash variables which are permanently built into Omnis, you must create all variables with the appropriate type and scope in the objects in your library using the method editor. After you have declared them, variables that are in scope are listed in the Catalog. You can remove a variable using the Delete Variable option in the variable pane context menu. Declared variables are removed from memory when they are destroyed.

### Parameter Variables

You can use a parameter variable to receive a value in a method, for example, a value that has been passed to the method using the *Do method*. You would normally do something with the value in the method and possibly return a new value. Parameter variables are visible within the called or recipient method only. They are initialized when the method is called, and cleared when the method returns to its caller.

### Local Variables

Local variables are local to the method. You can refer to the variable within that method only. Local variables are initialized when the method begins execution, and are cleared automatically when the method terminates.

### Instance Variables

Instance variables are visible to the instance only, that is, all methods and objects in the instance. You can define an instance variable only for classes that can be opened or instantiated, that is, tasks, tables, windows, remote forms, remote tasks, reports, menus, toolbars and object classes, as well as External Objects. Note that you cannot declare instance variables in code classes. There is a set of the declared instance variables for each instance of a class: these are initialized when the instance is constructed and cleared when the instance is destructed.

### Class Variables

Class variables are visible within the class and all its instances. You can declare class variables for tasks, tables, windows, reports, menus, toolbars, and code classes. Any object or method in the class can refer to a class variable, and all instances of the class also have access to the class variable.

Class variables are not automatically cleared from memory. You can remove them from memory by closing the library containing the class, or using the *Clear class variables* command.

### Task Variables

Task variables are visible within the task, all its design classes and instances. In practice, you can refer to a task variable from any method within any class or instance that belongs to the task. Omnis initializes task variables when you open the task: for the Startup\_Task this is when the library opens. Note that you cannot declare a task variable for a class until you have set the **$designtaskname** property for the class.

### Hash Variables

Omnis has a built-in set of global variables, called *hash variables* since they start with the symbol "#". You can view them in the Catalog (F9/Cmnd-9). Hash variables are global, unlike any other variables, so all libraries have access to them. The advantage of having global variables is that you can use these variables to pass data between libraries in an application. The disadvantage is that any data you place in hash variables remains there when you switch between libraries or combine libraries, with potentially unpredictable results.

### Adding a Variable

You add variables to a class or object in the Variable Pane of the Method Editor. If the variable pane is not visible you can show it using the View>>Show Variable Panes menu option on the method editor menu bar.

The tabs in the Variable Pane let you define Task, Class, Instance, Local and Parameter variables: note that the local and parameter tabs only appear after you have added or selected a method in the method editor. You can add up to 400 variables of each type to the current object, including 400 local and parameter variables for each method in the current object. The name, type, subtype, and initial value of each variable is listed in the variable pane. You can size the columns in the variable pane by sizing the column headers.

You cannot declare a task variable within a class until you have set the $designtaskname property for the class: see the section below on *Adding Task Variables*.

**To add a new variable**

1. Open the class in which you want to add the variable
2. Right-click on the background of the class to open the class context menu
3. Select the Methods option to open the method editor
4. Choose the tab for the type of variable you require
5. Click in the blank field under the Variable column header
6. Enter the name of the variable

or

1. Right-click in the variable pane to open the variable context menu
2. Choose Insert New Variable and click in the variable name to edit it, or type over the new variable name if it is selected (see Variable naming below)
3. Tab to the Type box and choose the type from the droplist using the mouse or arrow keys

or when the focus is in the Type box

1. Type the first letter(s) of a data type to select it, for example, you can type “nu” to select the Number data type, or “b” for Boolean and “bi” for Binary type

For Number and Date Time variables

1. Tab to the Subtype box and choose a subtype: again, you can type the first letter(s) of a subtype, for example, for Numbers you can type “L” to select the Long Integer subtype

You can enter an initial value or calculation for all types of variable. The initial value allowed for a variable depends on its type. See *Variable Values* below.

#### Variable Type

As well as scope, the data type you choose for a variable is critical to how it functions in your code and any calculations, The types available include Character, Number, Boolean, Picture, List, Row and Object, which are described in full under the Omnis Data Types section in the *Libraries and Classes* chapter.

#### Variable Naming

Variable names can be up to 255 characters long, although in practice you should keep them as short but descriptive as possible. When you name a variable you should prefix its name with one or more letters to indicate its scope. For example, parameters variables can begin with the letter “p”, local variables “lv” or just the letter “l”, instance variable “iv” or just the letter “i”, and so on.

When two or more types of variable use the same variable name, a reference to that variable could be ambiguous. In a situation where more than one variable of the same name exists, Omnis automatically uses the variable with the smallest scope. Therefore, it is possible, though not good practice or recommended, to have local, class, and task variables called "MYNAME". As Omnis resolves ambiguity, a reference to MYNAME will refer to the local variable if it exists for the current method. To avoid all ambiguity you should use a naming convention similar to the one described above

#### Adding Local and Parameter Variables

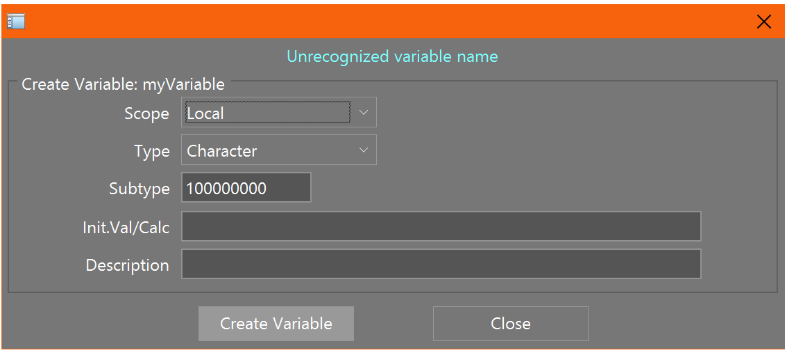
Local and parameter variables are inserted into the currently selected method. Therefore to insert these variables for a particular method, you need to select the method before inserting local and parameter variables.

Parameter variables receive values from a calling method in the order that they appear in the variable pane. You can reorder parameter variables by dragging them into position in the variable pane. Click and drag the fixed-left column or row number for a parameter variable in the list.

Normally you must declare all types of variable, including local variables, in the variable pane before you can use them in your code. However you can declare a special type of local variable in your code without first declaring it in the method editor. To declare such a variable, prefix the variable name with %% to create a string variable, or prefix the variable name with % for a numeric variable of Floating point type. You can type such variable names directly into your code in the method editor, at which time their names are added to the Local variables pane.

#### Creating Unrecognized Variables

When Omnis encounters a variable name that does not exist, say if you type its name in the Code Editor, a dialog will pop up automatically that allows you to create the unknown variable, including options for the scope, data type, subtype, initial value and description. This may occur when you type a new or unknown variable name into the current method in the Code Editor where Omnis expects a variable according to the syntax: the dialog only opens if the invalid text is a possible variable name.



The unrecognized variable dialog can open when assigning a new or unknown variable name to a property in the property manager. In this case, for properties such as $dataname, the initial type of the variable creation dialog is set to the most likely data type for the control, e.g. List data type for a list form control. The dialog restricts the scope of the new variable to what makes sense based on class type, and so on.

#### Create Variable Prefixes

When you encounter the "Unrecognized variable name” error when entering code in the editor, you can press the Fix button to open the Create Variable dialog to create the variable. You can now specify the initial scope for a new variable using a predefined *prefix.* For example, you can begin the variable name with “i” to create an instance variable, or “p” to create a parameter. The default variable prefixes are:

|  |  |
| --- | --- |
| **Prefix** | **Variable scope** |
| i | Instance |
| c | Class |
| p | Parameter |
| l | Local |
| t | Task |

The prefixes allowed in the Create Variable dialog can be configured in the Omnis configuration file (config.json) using a new entry called “createVariableScopePrefixes” and located in the ‘codeAssistant’ section in config.json:

"createVariableScopePrefixes": [

"i:Instance",

"c:Class:",

"p:Parameter",

"l:Local",

"t:Task"

],

The Create Variable dialog processes these entries in array order, and as soon as it finds a scope that is allowed for the method being edited (e.g. instance variables are only allowed for class types that have instances), where the first part of the entry value case insensitively matches the start of the variable name, it uses the configured scope (the second part of the entry value after the colon) to set the initial scope suggested by the dialog. If no prefix match occurs, the scope suggested is local.

#### Create Variable Suffixes

As well as setting the scope of a variable, using a prefix, you can specify the data type of a variable using one of a set of predefined *suffixes.* For example, you could enter the name “iData**Row**” which would create a variable of type Row, typing “iDataList” would create a list, and typing “iVarRef” would create an item reference. The default variable suffixes are:

|  |  |
| --- | --- |
| **Suffix** | **Variable type** |
| Row | Row variable (kRow) |
| List | List variable (kList) |
| Ref | Item reference variable (kItemref) |
| Date | Date variable (kDate) |
| Obj | Object variable (kObject) |
| Bin | Binary variable (kBinary) |

The suffixes allowed in the Create Variable dialog can be configured in the Omnis configuration file (config.json) using a new entry called “createVariableTypeSuffixes” and located in the ‘codeAssistant’ section in config.json:

"createVariableTypeSuffixes": [

"Row:kRow",

"List:kList",

"Ref:kItemref",

"Date:kDate",

"Obj:kObject",

"Bin:kBinary"

],

Omnis strips any consecutive digits from the end of the desired variable name, and then compares (case independently) the end of the resulting name string against the suffixes in the config.json array (strings before the colon in each array entry). If there is a match, and if the variable type is suitable (e.g. it is not a non-client executed type when creating a variable for a client-executed method), then the initial type is set using the type constant after the colon.

#### Deleting Unused Variables

The context menu of the Variable pane has the option **Delete Unused Variables…**, available by Right-clicking on the variable pane away from a variable line. When selected, it opens a dialog from which you can select variables to delete. The dialog displays the variables of the current type displayed in the variable pane, which are potentially unused. This means the variables could still be in use, for example, they could still be used in subclasses or notation.

#### Adding Task Variables

To add a task variable for a class you *have* to set its $**designtaskname** property. In most cases, the design task for a class is specified as the Startup\_Task by default. You can change it using the Property Manager or the notation. The design task for a class is ignored at runtime.

To set up the design task for a class

1. Click on the class in the Browser
2. Display the Property Manager or bring it to the top (F6/cmnd-6)
3. Click on the droplist in the $**designtaskname** property to view the current tasks

The list of tasks will contain a Startup\_Task, and any tasks you may have created.

1. Select the design task by clicking on it

You will now be able to define task variables for this class.

Changing the Scope of Variables

You can change the scope of a variable at any time by dragging the variable from one variable pane to another. For example, you can change a class variable into an instance variable by dragging and dropping it onto the instance variable tab. Note you cannot change the scope of task variables.

### Variable Initial Values

When you declare a variable in the variable pane of the method editor you can assign it an *initial value*. The first time a variable is referenced in a method, Omnis assigns the specified initial value to the variable. You can set the initial value to be a number, string, calculation, some notation, an Omnis constant, or another variable name. In the latter case, when you first use the variable it gets the value in the other variable, regardless of the order of declaration.

For class variables only, the *Clear class variables* command clears the current values in all class variables and resets them to their initial values.

You can set the initial value of parameter variables, which in effect gives them a default value, but when and if a value is received in the method the initial value is overridden. For example, you may want to assign an initial value of zero to a parameter value to avoid it being null if a value is not received.

For instance variables, the initial value is assigned when the instance is created, e.g. when a form is opened.

Variable Context Menu

You can lookup and edit the value of any variable or constant in Omnis at any time using its context menu. You can Right-click on a variable name wherever it appears in Omnis to open its context menu and view its current value. The Variable context menu displays the variable name, its current value, which group of variables or class it belongs to, and its type and length. You can also perform various debugging functions from this menu as well.

If you select the first option in the Variable context menu, Omnis opens a variable window containing the current contents of the variable which you can edit. Note that you cannot edit binary variables.

Variable Tooltips

You can pass the mouse over a variable or constant and a *variable tooltip* will pop up displaying the variable’s current value and description. Variable tooltips are available wherever variable names appear in Omnis including the method editor and Catalog. However, they are not available if Help tips are enabled for the tool containing the variable. For some variable types, such as list or binary variables, the tip may say “not empty” which tells you the variable has a value, but it is too long to display.

Variable panel

The Variable panel is displayed in the lower-right section of the Method Editor and allows you to view and modify variables while debugging or stepping through your code. This is described in more detail in the *Debugging Methods* chapter.

### Viewing Variables in the Catalog

You can view the variables in your library and the current class using the Catalog (press F9/Cmnd-9 to open it). The Variables pane shows all the Task, Class, and Instance variables for the current class, plus all Local and Parameter variables for the currently selected method. Following the Event Parameters group, the Catalog also lists any file classes in your library. You can enter the name of any variable that appears in the Catalog into your code either by double-clicking on the name in the Catalog (assuming the cursor is at a position that can accept input), or by dragging the variable name out of the Catalog into the method editor.

When you drag a variable from the Catalog, Omnis shows you what type of variable it is and its name. Note that you can also drag variables from the Catalog and drop them onto window and report classes to create a data field for the variable.

You can also drag a variable from the variable pane in the method editor to any calculation or entry field in the command palette. To drag a variable name you need to click and drag the fixed-left column or row number in the variable list.

### Auto Fill Variable Option

When you want to enter a variable in the method editor command palette and you can’t remember its full name, you can type the first few characters of the variable, wait a short while for a list to appear, and choose the variable from the list that pops up. The list contains all the variables beginning with the characters you typed. The time it takes for the autofill option to work is set in the $notationhelptimer Omnis preference (the default is 1000 milliseconds).

### Custom Variable Types

You can define your own custom variable types. To do this you have to create a custom method called $<customatttribute name>.$assign, and then the $type, $subtype and $sublen properties of the custom variable return their value according to the type of parameter 1 of $<customatttribute name>.$assign.

### Comparing Variables

You can do comparisons in the Omnis language between *binary* variables, *object* variables and *object reference* variables, when both sides of the operator are the same type.

Binary comparisons compare the data byte by byte until there is a non-matching byte, in which case the first variable is greater than the second variable if the non-matching byte in the first variable is greater than that in the second variable. The comparison extends to the length of the shortest variable: if all bytes match, then the first variable is greater than the second if it is longer than the second, and vice versa.

Object comparisons compare the object instance – if the instance is the same, the variables are equal.

## Methods

Omnis provides a complete 4GL programming language comprising over 400 commands, each command performing a specific function or operation. In addition, Omnis provides a means to manipulate the objects in your library called the *notation*: this accesses the standard properties and methods contained in the objects in your library.

In previous versions of Omnis, the number of method lines was limited 1024, but this limit has been removed. However, although the number of method lines is theoretically unlimited, the maximum number of method lines is capped at 256,000 to maintain efficiency in your code.

Each method line can contain an Omnis command, or some notation, or often a combination of these: you can also add comments to method lines. For example, to open a window from a menu line you only need one command in your method, that is the *Open window instance* command, which as the name suggests opens an instance of a window. A method that connects you to a server database requires several commands executed in a particular order. You can perform most operations using the notation and the *Do* command. For example, you can open a window using the *Do* command and the $open() method.

For further details about specific commands and notation used throughout this chapter, see the Omnis Help (press F1 to open it), or the Omnis Reference manuals. When you start to program methods you will need to use the debugger which is described in the Debugging Methods chapter.

### Commands

The following sections outline the more important commands or groups of commands in Omnis. The commands that you can use in your methods are listed in the *command list* at the bottom of the method editor. If the command list is not showing in the method editor you can show it using View>>Show Command Palette, or by pressing Shift-F6 under Windows or Shift-Cmnd-6 under macOS.

Double-click on each group in the command list to get an idea of the full range of commands available in Omnis.

Each group in the command list contains a number of commands that manipulate a particular type of object or perform related operations. For example, the *Calculations*... group contains the *Calculate* command that lets you do calculations and assign a value to a variable, and the *Do* command that lets you execute and modify objects using the notation. The *Constructs*... group contains programming constructs such as *If...Else If, Repeat...Until,* and *For* loops.

### The Flag (#F)

Many of the Omnis commands set a Boolean Omnis variable called the *flag,* or #F, to true or false depending on the success of an operation. Other commands test the current value of the flag and branch accordingly. The *Omnis Studio* Help documents whether or not a command affects the flag. You can return the current status of the flag (#F) in client executed methods in the JavaScript Client using the *flag()* function.

### Notation

Omnis structures its objects in an object tree, or hierarchical arrangement of objects and groups that contain other objects. The complete tree contains all the objects in Omnis itself, together with your design libraries, classes, and other objects created at runtime. You can view the complete object tree in the Notation Inspector.

The object at the base of the tree is called $root. The $libs group contains all the current open libraries and lets you access each library and its classes at design time. The classes and objects in each library are stored in their own separate groups: for example the $windows group contains all the window classes in a library. Most of the other groups directly under $root contain the objects created at runtime when you run your application: for example the $iwindows group contains all the window instances currently open, or $iremotetasks contains all the remote task instances currently open.

When you want to reference a particular object, a class or instance perhaps, you must access the right branch of the object tree. For example, you must access the $windows group to reference a window class. Whereas, to access a window instance, say an instance of the same window class, you must reference the window instance via the $iwindows group, which is contained directly under the $root object.

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

In the notation all property and standard method names begin with a dollar sign “$”, and methods are further distinguished from properties by having parentheses after their name. Standard objects and group names also begin with a dollar sign. To write the full notation for an object you need to include each object and group in the path to the object, separating each object using “.” a dot. For example, to refer to a window class in a library you would use the following notation

###### $root.$libs.LIBRARYNAME.$windows.Windowname

This notation includes $root as the base object, the $libs group containing all the open libraries, the name of your library, the $windows group containing all the window classes in your library, and lastly the name of the window itself. If you want to refer to a particular object on your window you need to add the $objs group and the name of the object

###### $root.$libs.Libraryname.$windows.Windowname.$objs.Objectname

You can omit certain object names from a notation string to make it shorter, and when you have only one library open usually you can omit the library name as well. You can omit the following objects: $root, $constants, $clib, $hashvars, $libs, $extobjects, $tvars, $datas, $cvars, $files, $lvars, $vals. In most cases therefore, you can refer to an object on a window as

###### $windows.Windowname.$objs.Objectname

In addition, there are a number of shortcuts that let you reference objects, without always referring right back to the $root object, and certain global objects that you can use to make your code more generic. These are described below.

#### Item References

To save you time and effort, and to make your code more efficient, you can create an alias or reference to an object which you can use in place of the full notation for the object. To do this, you create a variable of type *item reference* and use the *Set reference* command to assign the notation to the variable. The item reference variable can be of any scope, and the notation can be any valid Omnis notation for an object, a group, or even an object property. For example

###### # Declare variable WinRef of type Item reference Set reference WinRef to Libraryname.$windows.Windowname # creates a reference to the window which you can use in your code Do WinRef.$forecolor.$assign(kBlue) ## changes the window forecolor

You can enter the notation for an object in the initial value field for the item reference variable. You can also find the full notation for an object in the Notation Inspector and drag it to the notation field when you enter the *Set reference* command.

You can also use an item reference variable to return a reference to a new object, when using methods to create a new class, instance, or object. Furthermore Omnis contains a special property called $ref which you can use to return an item reference to an object. Both these features are used in the section describing the *Do* command below.

Note that WinRef.$parentfolder, where WinRef is an item reference to a class, will return an item reference to the parent folder of the class and not the $ident value of the folder class containing the class which is usually the case.

#### Current Objects

Under $root, Omnis contains a number of global state variables that tell you about how Omnis is currently executing, or what objects, instances, and methods are currently being used. These objects provide a shortcut to the *current object* or instance that is currently executing. Mostly their names begin with “$c”, and they include

1. $cclass  
   the current class
2. $cdata  
   the current open data file
3. $cinst  
   the current instance: usually the instance containing the currently executing method
4. $cfield  
   the field where the current method is executing
5. $clib  
   the current library
6. $cmethod  
   the current executing method
7. $cobj  
   the current object within a class or instance
8. $cparmcount  
   the number of parameters that the caller has passed to the current method
9. $crecipient  
   the current recipient of an event: if a custom method is being processed, $crecipient is the recipient of that method
10. $ctarget  
    a reference to the target field, that is, the field which currently has the focus (shows the caret and is sent keyboard events)
11. $ctask  
    the current task: is usually the startup or default task until you open another task
12. $cwind  
    the current window instance
13. $topwind  
    the topmost open window instance

You can use the current objects in place of the full notation for a specific object to make the object and its code reusable and portable between libraries. For example, you can use $cinst in a method within a window instance to refer to itself, rather than referring to it by name

$cinst

# rather than

$root.$iwindows.WindowInstanceName

You can refer to the current library using $clib. For example, to make the current library private use

###### Do $clib.$isprivate.$assign(kTrue) # is more generic than Do $libs.MyLibrary.$isprivate.$assign(kTrue)

### Do Command and Executing Methods

While you can use *Calculate* to change an object property or evaluate an expression, you can use the *Do* command for all expressions that execute some notation. In this respect, the *Do* command is the single-most powerful command in Omnis. You can use the *Do* command to set the value of a property, or to run any standard or custom method. The *Do* command has several variants which include

1. *Do*   
   sends a message to an object in your library, or assigns a value to an object property. Normally you should execute the *Do* command in the context of the *current object* to execute one of its methods or assign to one of its properties. There are a number of common methods that you can use with the *Do* command including $open() to open an instance of a class, $assign() to change an object property, and so on
2. *Do inherited*   
   executes the inherited method for the current method
3. *Do default*   
   runs the default processing for a custom method
4. *Do redirect*   
   redirects method execution to a custom method with the same name as the current method contained elsewhere in your library
5. *Do method*   
   calls a method in the current class and returns a value
6. *Do code method*   
   runs a method in a code class and returns a value

Note that you can display a list of built-in methods for an object or object group by clicking on the object in the Notation Inspector and opening the Property Manager. The methods for an object are listed under the Methods tab in the Property Manager: to view all the methods of a class or object, ensure that the ‘Show All’ option in the Property Manager is enabled. See *Omnis Studio* Help for a complete list of methods for all the objects in Omnis. The Show Runtime Properties option in the Property Manager context menu lets you view properties that are normally available in runtime only, that is, properties of an instance rather than a design class. When runtime properties are visible in the Property Manager the methods for the instance are also shown. You cannot set runtime properties or use methods shown in the Property Manager, they are there as a convenient reference when you are writing code.

#### Do command

You can use the *Do* command in Omnis to do almost anything: execute some notation, evaluate an expression, and so on. Specifically, you can use it to execute a method for an object or assign a value to one of its properties. The *Do* command returns a value to indicate whether the operation was successful or not, or for some methods a reference to the object operated upon. This section shows you how you can use the *Do* command and introduces some of the most useful methods.

#### Calling Private Methods

The *callprivate()* function allows you to call a private method within the current class or instance and return a value. The syntax is:

###### callprivate(method[,parameters...] ## calls the private method

The function can be called in client methods in the JavaScript Client.

#### $open() method

Using the *Do* command with the notation you can perform many operations that are otherwise performed with a command. For example, the class types that you can open contain an $open() method which you can execute using the *Do* command. For example, you can open a window using

###### Do $windows.WindowName.$open(‘InstanceName’,kWindowCenter) # opens a window in the center of the screen

The $open() method returns a reference to the instance created. For example

###### # Declare variable WindRef of type Item reference Set reference WindRef to LIB1.$windows.WindowName Do WindRef.$open('WindowInstance') Returns WindRef # WindRef now contains a reference to the window instance # ‘$root.$iwindows.WindowInstance’ which you can use elsewhere, e.g. Do WindRef.$forecolor.$assign(kBlue) ## changes the instance

You can use a null value instead of an instance name: therefore CLASS.$open(‘’) would force Omnis to use the class name as the instance name. Alternatively you can use an asterisk in place of the instance name and Omnis assigns a unique name to the instance, using the notation CLASSNAME\_number. You can return the instance name in an item reference variable and use the reference in subsequent code. For example

###### # Declare variable iMenuRef of type Item reference Do $menus.MCUSTOMERS.$open('\*') Returns iMenuRef # iMenuRef now contains a reference to the menu instance, which # will be something like ‘$root.$imenus.MCUSTOMERS\_23’

You can close an instance using the $close() method. For example, the following method opens a window instance, lets the user do something, and closes the instance

###### # initially WindRef contains a reference to the window class Do WindRef.$open('WindowInstance') Returns WindRef # let the user do something Do WindRef.$close()

You can close the current window from inside the instance using

###### Do $cwind.$close()

Classes that contain the $open() methods also have the $openonce() method. This method opens an instance if one does not already exist (excluding window menus, window toolbars, and cascaded menus). In the case of a window, $openonce() brings the window to the top if it is already open. $openonce() returns an item reference to the new or existing instance, like $open().

#### $assign() method

You can change the properties of an object, including the properties of a library, class, or field, using the *Do* command and the $assign() method. The syntax for the $assign() method is Notation.Property.$assign(Value) where Notation is the notation for the object, Property is the property of the object you want to change, and Value is a value depending on the context of the object being changed. Usually you can use an Omnis constant to represent a preset value, and for boolean properties, such as preferences, you can use kTrue or kFalse to set the property as appropriate. For example

###### Do $clib.$prefs.$mouseevents.$assign(kTrue) # turns on mouse events for the current library Do $cclass.$closebox.$assign(kTrue) # adds a close box to the current window class Do $cfield.$textcolor.$assign(kGreen) # makes the text in the current field green

#### $add() method

You can create a new object in your library using the $add() method. In the notation you are really adding a new object to a particular group of objects. For example, to create a new field on a window you need to add the object to the $objs group of objects for the window, as follows

###### Do $cwind.$objs.$add(kPushbutton,iTop,iLeft,iHeight,iWidth) # adds a pushbutton to the window with the # specified size and position

When using $add(), you can return a reference to the new object in a return field of type item reference. You can use the reference to change the properties of the new object. For example

###### # Declare variable WindRef of type Item reference Do $windows.$add('NewWindowName') Returns WindRef # now use the reference to change the new window Do WindRef.$style.$assign(kPalette) Do WindRef.$title.$assign('Window title') Do WindRef.$clickbehind.$assign(kTrue) Do WindRef.$keepclicks.$assign(kFalse) Do WindRef.$modelessdata.$assign(kTrue) Do WindRef.$backcolor.$assign(kRed) Do WindRef.$forecolor.$assign(kWhite) Do WindRef.$backpattern.$assign(2)

#### $redraw() method

*Note the $redraw() method is only relevant for fat client windows, not JavaScript Remote Forms which redraw content automatically.*

When you change an object or several objects on an open window using the *Do* command, you often need to redraw the window. However if you change an object before $construct() completes execution for the window instance, you don’t need to redraw the window. You can redraw an object, window, or all open windows using the $redraw() method. For example

###### Do $cfield.$redraw() # redraws the current field Do $cwind.$redraw() # redraws the current window Do $root.$redraw() # redraws all window instances

The $redraw() method has three parameters that allow you to specify the extent of the redraw for window fields and/or background objects: the parameters are: $redraw(bSetcontents,bRefresh,bBackObjects) where bSetcontents defaults to true, bRefresh defaults to false, and bBackObjects defaults to false.

###### $root.$redraw(kTrue,kTrue) # redraws the contents and refreshes all the field in all window $root.$redraw(kFalse,kFalse,kTrue) # redraws all background objects for all open windows

#### $sendall() method

You can send a message to all the items or objects in a group using the *Do* command and the $sendall() method. For example, you can redraw all the objects in a group, you can assign a value to all the members of an object group, or you can hide all the members of a group using the $sendall() method and the appropriate message. The full syntax for the method is:

###### $sendall({message|message,condition [,bIgnoreUnrecognizedCustomAttribute=kFalse]})

where *message* is the message you want to send to all the objects and *condition* is a calculation which the objects must satisfy to receive the message. For example

###### Do $iwindows.$sendall($ref.$objs.FieldName.$redraw()) # redraws the specified field on all window instances Do $cwind.$objs.$sendall($ref.$textcolor.$assign(kYellow)) # makes the text yellow for all the fields on the current window Do $cwind.$objs.$sendall($ref.$visible.$assign(kFalse),$ref.$order<=5) # hides the first five objects on the current window useful # for window subclasses if you want to hide inherited objects

The optional third argument bIgnoreUnrecognizedCustomAttribute causes $sendall() to ignore unrecognized custom attribute errors, which would otherwise cause a runtime error when the library preference $reportnotationerrors is kTrue. This argument defaults to kFalse if omitted.

#### $sendallref ()method

When using $sendall(), you can use $ref to refer to the group member receiving the message. However you can use $sendallref, which is an item reference to the item currently receiving the message sent with $sendall (note that $sendallref is not supported in client methods). Consider the case where a parameter passed to the message is evaluated by calling another method, or a function implemented in an external component. In this case, if you use $ref in the parameters passed to this other method or function, it will actually refer to the item involved in making the call to evaluate the parameter. This is where $sendallref() could be used, if you wish to pass some property of the group member receiving the message to the other method or function.

For example:

###### Do $cinst.$bobjs.$sendall( $ref.$text.$assign( StringTable.$gettext( $cclass.$bobjs.[$sendallref.$ident].$text)))

The example uses the text stored in the class as the row id in the string table, and assigns the text stored in the string table to the background object. In the example, $sendallref.$ident returns the ident of the background object receiving the message. If you were to use $ref.$ident, the $ref would refer to the custom attribute representing the external component function, and the call to $sendall would not have the desired effect.

#### $makelist() method

Quite often you need to build a list containing the names of all the objects in a group, and you can do this using the makelist() method. For example

###### Do $clib.$classes.$makelist($ref.$name) Returns cLIST # builds a list of all the classes in the current library and # places the result in cLIST Do $imenus.$makelist($ref.$name) Returns cLIST # builds a list of all the currently installed menus

###### Do $clib.$objects.[MyObject].$methods.$makelist (...)

###### # returns only the methods overridden if "MyObject" has a superclass.

#### Do inherited

The *Do inherited* command runs an inherited method from a method in a subclass. For example, if you have overridden an inherited $construct() method, you can use the *Do inherited* command in the $construct() method of the subclass to execute the $construct() method in its superclass.

#### Do default

You can use the *Do default* command in a custom method with the same name as a standard built-in method to run the default processing for method. For example, you can use the *Do default* command at the end of a custom $print() method behind a report object to execute the default processing for the method after your code has executed.

#### Do redirect

You can use the *Do redirect* command in a custom method to redirect method execution to another custom method with the same name that is contained in another object in your library. You specify the notation for the instance or object you want execution to jump to.

Inheritance and custom methods are further discussed in the Object Oriented Programming chapter.

### Calculate Command and Evaluating Expressions

This section describes how you use the Calculate command with an expression. It also discusses using square bracket notation for strings. You should also see the later section on error handling.

The *Calculate* command lets you assign a value to a variable calculated from an Omnis expression. Expressions can consist of variables, field names, functions, notation strings, operators, and constants. For example

###### Calculate var1 as var2+var3

in this case, “var2+var3” is the expression.

###### Calculate var1 as con('Jon', 'McBride')

Here the expression uses the *con()* function which joins together, or concatenates, the two strings ‘Jon’ and ‘McBride’. You must enclose literal strings in quotes.

See the *Omnis Studio* Help for a complete list of functions. In expressions, functions appear as the function name followed by parentheses enclosing the arguments to the function. The function returns its result, substituting the result into the expression in place of the function reference. Calling a function does not affect the flag.

The Omnis operators are shown below, in precedence order, that is, the order in which they get evaluated by Omnis. Operators in the same section of the table are of equal precedence, and are evaluated from left to right in an expression.

|  |  |
| --- | --- |
|  |  |
| Parentheses | () |
| Unary minus | - |
| Multiplication  Division | \*  / |
| Addition  Subtraction | +  - |
| Less than  Greater than  Equal to  Less than or equal to  Greater than or equal to  Not equal to | <  >  =  <=  >=  <> |
| Logical AND  Logical OR | &  | |

When you combine expressions with operators, the order of expressions will often make a difference in the interpretation of the expression: this is a consequence of the mathematical properties of the operators such as subtraction and division. You can group expressions using parentheses to ensure the intended result. For example

###### Calculate lv\_Num as 100 \* (2 + 7)

evaluates the expression in parentheses first, giving a value of 900. If you leave off the parentheses, such as

###### Calculate lv\_Num as 100 \* 2 + 7

Omnis evaluates the \* operator first, so it multiplies 100\*2, then adds 7 for a value of 207.

#### Square Bracket Notation

You can use a special notation in strings to force Omnis to expand an expression into the string. You do this by enclosing the expression in square brackets: Omnis evaluates the expression when the string value is required. You can use this in all sorts of ways, including the technique of adding a variable value to the text in the SQL or text buffer.

You can use square bracket notation wherever you can specify a single variable or field name, including

1. command parameters, for example, OK message

###### OK message {Your current balance is [lv\_curbalance]}

1. window or report fields: you can include values in text objects, such as

###### Your current balance is [lv\_curbalance]

1. variable or field names within a *Calculate* command or text object
2. function parameters

Square bracket notation lets you refer to a value indirectly letting you code general expressions that evaluate to different results based on the values of variables in the expression: this is called *indirection*. For example, you can include a variable name enclosed in square brackets in a text object to add the value to the text at runtime. However in general, there is a significant performance penalty in using indirection.

If you need to use [ or ] in a string but do not want the contents evaluated, then use [[ and ] to enclose the contents—double up the first or opening square bracket. This is useful when you use square bracket notation with external languages that also use square brackets, such as the VMS file system or DDE.

#### Type Conversion in Expressions

Omnis tries its best to figure out what to do with values of differing data types in expressions. For example, adding a number and a string generally isn't possible, but if Omnis can convert the string into a number, it will do so and perform the addition. Some other examples are

###### # Declare local variable lDate of type Date D m Y Calculate lDate as 1200 # 1200 is no. of days since 31st Dec 1900 Calculate lDate as 'Jun 5 93' # conv string to date in format D m Y OK message {Answer is [jst(lDate,'D:D M CY')]} ## reformat date Calculate lNum as lDate ## sets lNum to 1200, the no. of days

Boolean values have a special range of possibilities.

1. YES, Y, or 1 indicate a true status
2. NO, N, or 0 indicate a false status

FALSE and TRUE are not valid values: Omnis converts them to empty.

###### # Declare local variable LBOOL of type Boolean Calculate LBOOL as 1 ## is the same as... Calculate LBOOL as 'Y' ## or 'YES' # the opposite is Calculate LBOOL as 0 ## or 'NO' or 'N' OK message { The answer is [LBOOL] } Calculate LBOOL as 'fui' ## is the same as... Calculate LBOOL as ''

You can convert any number to a string and any string that is a number in string form to a number.

###### # Declare local variable lChar of type Character # Declare local variable lNum of type Number floating dp Calculate lChar as 100 OK Message { [lChar], [2 \* lChar], and [con(lChar,'XYZ')] } # Gives message output 100 200 and 100XYZ Calculate lNum as lChar Calculate lChar as lNum OK Message { [lChar], [lNum \* lChar], and [con(lChar,'ABC')] } # Gives message output 100 10000 and 100ABC

#### Constants

You will often find situations in Omnis where you must assign a value that represents some discrete object or preset choice. Omnis has a set of predefined constants you should use for this kind of data. For example, a class type can be one of the following: code, file, menu, report, schema, and so on. Each of these is represented by a constant: kCode, kFile, kMenu, kReport, kSchema, respectively. You can get a list of constants from the Catalog: press F9/Cmnd-9 to open the Catalog. You can use constants in your code, like this

###### Calculate obj1.$align as kRightJst ## or use Do Do obj1.$align.$assign(kRightJst) # aligns the object obj1 to the right

Although you can use the numeric value of a constant, you should use the predefined string value of a constant in your methods. In addition to ensuring you're using the right constant, your code will be much more readable. Moreover, there is no guarantee that the numeric value of a particular constant will not change in a future release of Omnis.

### Calling Methods

You can execute another method in the current class using Do method, or call a method in a code class using Do code method. These commands let you pass parameters to the called method and return a value in a return field. For example, the following method named Setup calls another method named Date and returns a value.

###### # Setup method Do method Date (lNum,lDate+1) Returns lDate OK message {Date from return is [lDate]}

###### # Date method, the called method # Declare Parameter var lpNum of type Number 0 dp # Declare Parameter var lpDate of type Short Date 1980..2079 OK message {Date from calling method is [lpDate], number is [lpNum]} Quit method {lpDate + 12}

Note that when you call a code class method from within an instance the value of $cinst, the current instance, does not change. Therefore you can execute code in the code class method that refers to the current instance and it will work.

WARNING Omnis does not stop a method calling itself. You must be careful how the method terminates: if it becomes an infinite loop, Omnis will exhaust its method stack.

### Quitting Methods

You can use the Quit command, and its variants, to quit methods at various levels.

1. Quit method  
   quits the current method and returns a value to the calling method, if any
2. Quit event handler  
   quits an event handling method
3. Quit all methods  
   quits all the currently executing methods, but leaves Omnis running
4. Quit all if canceled  
   quits all methods if you press Cancel
5. Quit Omnis  
   exits your application and Omnis

You can also clear the method stack with the Clear method stack command, which does the same thing as the debugger menu **Stack>>Clear Method Stack**: it removes all the methods except for the current one. If you follow Clear method stack with Quit method, it has the same effect as Quit all methods.

The *Quit method* command allows you to exit a method and return a value to the calling method. For example:

###### # Quit the method myMethod and return the flag # from the Yes/No message to the calling method

###### # the calling method Do method myMethod Returns lReturnFlag

###### # method myMethod Yes/No message {Continue ?} Quit method #F

It is possible to call another method in the return value of a *Quit method* command, but this can lead to unpredictable results, especially if the called method contains an *Enter Data* command, e.g.

###### Quit method Returns iOtherObject.$doSomeThingThatContainsEnterData

### Flow Control Commands

The *Constructs...* group contains many commands that let you control the execution and program flow of your methods. *If* statements let you test a condition and branch accordingly: loop commands iterate based on tests or sequences: the *Comment* command lets you comment your code: and reversible blocks let you manipulate objects and values and restore their initial values when the block terminates.

Several commands in this command group have starting and terminating commands (If and End if, for example). You must use the correct terminating command, or you will get unexpected results. If chromacoding is enabled, the beginning and terminating commands for most branching and looping constructs are highlighted. You can enable chromacoding using the View>>Show ChromaCoding menu option in the method editor.

#### Highlighting Blocks

The start and end of any block commands are highlighted when one of the statements that makes up the construct formed by the commands is selected in the method editor. For example, if a **For** statement is the current line, then the "End for" and "For" will both be highlighted. Or if a **Case** statement is the selected line, then all cases in the same switch, "Default", "Switch" and "End switch" will all be highlighted. The style or color of the highlighting uses a pair of chroma coding options, $currentblocktextcolor and $currentblockstyle.

#### Branching Commands

The If command lets you test the flag, a calculation, or a Cancel event. The Flag is an Omnis variable with a True or False value which is altered by some commands to show an operation succeeded, or by user input. The Else command lets you take an alternative action when the *If* evaluates to false, Else if gives you a series of tests. You must use the End If command to terminate all If statements.

A simple test of the flag looks like this:

###### If flag true   Do method Setup End If

You can do a sequential checking of values using a calculation expression:

###### If CollCourse ='French'   Do method Languages Else If CollCourse = ‘Science’   If CollSubCourse = ‘Biology’     Do method ScienceC1   Else     Do method ScienceC2   End If Else   OK message {Course is not available.} End If

#### While Loops

The While loop tests an expression at the beginning of a loop. The While command will not run the code block at all if the expression is false immediately. You would use a While command when you want to loop while an expression is true.

###### # Declare Count with initial value 1 While Count <= 10   OK message {Count is [Count]}   Calculate Count as Count + 1 End While

This loop will output 10 messages. If the condition was ‘Count <= 1’, it would run only once.

#### Repeat Loops

A Repeat loop lets you iterate until an expression becomes true. Repeat loops always execute at least once, that is, the test specified in the *Until* command is carried out at the *end* of the loop, after the commands in the loop are executed, whereas While loops carry out the test at the beginning of the loop.

###### # Declare Count of Integer type with initial value 1 Repeat   OK message {Count is [Count]}   Calculate Count as Count + 1 Until Count >= 10

This loop will output 9 messages.

#### For Loops

The For field value command lets you loop for some specific number of iterations, using a specified variable as the counter. The following example builds a string of ASCII characters from their codes using the functions *con()* and *chr().*

###### # Declare Count Calculate cvar1 as '' ## clear the string For Count from 48 to 122 step 1 ## set the counter range   Calculate cvar1 as con(cvar1,chr(Count)) ## add char to string   Do $cwind.$redraw() End For

The *For each line in list* command loops through all the lines in the current list.

###### Set current list LIST1 For each line in list from 1 to LIST1.$linecount step 1   # process each line End For

#### Switch/Case Statements

The Switch statement lets you check an expression against a series of values, taking a different action in each case. You would use a Switch command when you have a series of possible values and a different action to take for each value.

The following method uses a local variable lChar and tests for three possible values, “A”, “B”, and “C”.

###### # Parameter pString(character 10) ## receives the string Calculate lChar as mid(pString, 1, 1) ## takes the first char Switch lChar   Case ‘A’     # Process for A   Case ‘B’     # Process for B   Case ‘C’     # Process for C   Default     # do default for all cases other than A, B, or C End Switch

It is a good idea to use the Switch command only for expressions in which you know all the possible values. You should always have one Case statement for each possible value and a Default statement that handles any other value(s).

#### Escaping from Loops

While a loop is executing you can break into it at any time using the *break key* combination for your operating system: under Windows it is Ctrl-Break, under macOS it is Cmnd-period, and under Unix it is Ctrl-C. Effectively, this keypress ‘quits all methods’. When Omnis performs any repetitive task such as building a list, printing a report, or executing a Repeat/While loop, it tests for this keypress periodically. For Repeat/While loops, Omnis carries out the test at the end of each pass through the loop.

To create a more controlled exit for the finished library, you can turn off the *end of loop* test and provide the user with a working message with a **Cancel** button. When the **Cancel** button is visible on the screen, pressing the Escape key under Windows or Cmnd-period under macOS is the equivalent to clicking **Cancel**. For example

###### Disable cancel test at loops ## disables default test for loops Calculate Count as 1 Repeat   Working message (Cancel box) {Repeat loop...}   If canceled     Yes/No message {Do you want to escape?}     If flag true       Quit all methods     End If   End If   Calculate Count as Count+1 Until Count > 200

The If canceled command detects the Cancel event and quits the method. To turn on testing for a break, you can use the Enable cancel test at loops command*.*

The Break to end of loop command lets you jump out of a loop without having to quit the method, and the Until break provides an exit condition which you can fully control. For example

###### Repeat   Working message (Cancel box) {Repeat loop...}   If canceled     Yes/No message {Are you sure you want to break out?}     If flag true       Break to end of loop     End If   End If Until break OK message {Loop has ended}

If you have not disabled the cancel test at loops, a Ctrl-Break/Cmnd-period/Ctrl-C terminates all methods and does not execute the OK message. Having turned off the automatic cancel test at loops, you can still cause a Quit all methods when canceled. For example

###### Disable cancel test at loops Calculate Count1 as 1 Calculate Count2 as 1 Repeat   Repeat     Working message (Cancel box) {Inner repeat loop}     Calculate Count2 as Count2 + 1   Until Count2 > 12   Calculate Count2 as 1   Working message (Cancel box) {Outer repeat loop...}   Quit all if canceled   Calculate Count1 as Count1 + 1 Until Count1 > 20

If the user selects Cancel in the outer loop, the method quits, but from the inner loop there is no escape.

#### Optimizing Program Flow

Loops magnify a small problem into a large one dependent on the number of iterations at runtime, and other program flow commands can use a lot of unnecessary time to get the same result as a simpler command.

Here are some tips to help optimize your methods.

Use the For command instead of the equivalent While or Repeat commands. For has a fixed iteration, while the other commands test conditions. By eliminating the expression evaluation, you can save time in a long loop.

Use the Switch command instead of equivalent If/Else commands where possible. Arrange both the Case commands within a Switch and the several If and Else if commands so that the conditions that occur most frequently come first.

Use the Quit method command to break out of a method as early as possible after making a decision to do so. This can be a tradeoff with readability for long methods because you have multiple exits from the method: if falling through to the bottom of the method involves several more checks, or even just scanning through a large block of code, you can substantially improve performance by adding the Quit method higher up in the code.

Avoid using commands that don’t actually execute within a loop. For example, don’t put comment lines inside the loop. You can also use *Jump to start of loop* to bypass the rest of that iteration of the loop.

You can speed up a frequently called method by putting *Optimize method* at the start: refer to *Omnis Studio* Help for details of this command.

### Reversible Blocks

A reversibleblock is a set of commands enclosed by Begin reversible block and End reversible block commands: a reversible block can appear anywhere in a method. Omnis reverses the commands in a reversible block automatically, when the method containing the reversible block ends, thus restoring the state of any variables and settings changed by the commands in the reversible block.

###### # commands... Begin reversible block # commands... End reversible block # more commands...

Reversible blocks can be very useful for calculating a value for a variable to be used in the method and then restoring the former value when the method has finished. Also you may want to change a report name, search name, or whatever, knowing that the settings will return automatically to their former values when the method ends.

The Omnis Help (press F1) indicates which commands are reversible.

Consider the following reversible block.

###### Begin reversible block   Disable menu line 5 {Menu1}   Set current list cList1   Define list {cvar5}   Build window list   Calculate lNum as 0   Open window instance Window2 End reversible block # more commands...

When this method terminates:

1. Omnis closes window Window2
2. Omnis restores lNum to its original value
3. The definition of cList1 returns to its former definition
4. Omnis restores the former current list
5. Omnis enables line 5 of Menu1

At the end of the method, Omnis steps back through the block, reversing each command starting with the last. If there is more than one reversible block in a method, Omnis reverses the commands in each block, starting from the last reversible block. If you nest reversible blocks, the commands in all the reversible blocks are treated as one block when they are reversed, that is, Omnis steps backward through each nested reversible block reversing each command line in turn. You cannot reverse any changes that the reversible block makes to Omnis data files or server-based data unless you carefully structure the server transaction to roll back as well.

#### Losing property values

Certain notation properties affect other properties when they are assigned, for example, assigning $calculated to kFalse clears $text for the field. Therefore, if the $calculated property is set within a reversible block, and the state of $calculated is reversed, the value in the $text property is not reinstated. Such relationships between properties are not supported by the reversible block mechanism. If you wish to maintain the value of a property that may get cleared during notation execution, you should store the value in your own variable and assign the value to the property at runtime.

### Error Handling

When you enter a command, Omnis automatically checks its syntax. When a command is executed in a method, you can get a *runtime error*, a processing error rather than a syntax error. *Fatal errors* either display a message and stop method execution or open the debugger at the offending command.

You can cause a fatal error to occur with the Signal error command, which takes an error number and text as its argument. This lets you define your own errors, but still use the standard Omnis error handler mechanism.

In addition, Omnis maintains two global system variables #ERRCODE and #ERRTEXT that report error conditions and warnings to your methods. Fatal errors set #ERRCODE to a positive number greater than 100,000, whereas warnings set it to a positive number less than 100,000.

You can trap the errors and warnings by adding a method to test for the various values of #ERRCODE and control the way Omnis deals with them: this is called an *error handler*. The command Load error handler takes the name of the method and an optional error code range as its parameters:

###### Load error handler Code1/1 {Errors} # warnings and errors will be passed to handler in code class

Once you install it, Omnis calls the error handler when an error occurs in the specified range. Please refer to the *Omnis Studio* Help for a detailed description of the Load error handler command and examples of its use.

There are several commands prefixed with SEA, which stands for Set error action. Using these commands, you can tell Omnis what to do after an error:

1. SEA continue execution  
    continues method execution at the command following the command that signaled the error: if the error handling routine has not altered them, #ERRCODE and #ERRTEXT are available to the command
2. SEA report fatal error  
   if the debugger is available, it displays the offending command in the method window and the error message in the debugger status line
3. SEA repeat command  
   repeats the command that caused the error.

Repeating a command should be done with care since it is easy to put Omnis into an endless loop. If the error has a side effect, it may not be possible to repeat the command. If an ‘Out of memory’ condition occurs, it may be possible to clear some lists to free up enough memory to repeat the command successfully.

#### Errors in the JavaScript Client

You can return the values of #ERRCODE and #ERRTEXT in client executed methods in the JavaScript using the functions errcode() and errtext().

#### Calculation Errors

The library preference $reportcalculationerrors (default is true) specifies whether or not calculation errors are reported. When true, Omnis will report errors that occur when evaluating calculations, such as divide by zero errors. The report message is sent to the trace log, containing the error and code that caused the problem.

In addition, when executing calculations using *Do* and *Calculate*, Omnis enters the debugger when the error occurs (provided debugging is allowed). (This will not occur when these commands execute in the Omnis Web Client plug-in.)

### Redrawing Objects

There are a number of commands that let you redraw a particular object or group of objects. The Redraw command has the following variants.

1. *Redraw* field or window  
   redraws the specified field or window, or list of fields or windows: note this command with refresh all instances of the window
2. *Redraw lists*  
   redraws all list fields on the current window or redraws all lists in your library
3. *Redraw menus*  
   redraws all the currently installed menus
4. *Redraw toolbar*  
   redraws the specified custom toolbar

You can use the $redraw() method to redraw a field or fields, a window or all windows, as described earlier in this chapter.

#### Refreshing window instances

You can use the $norefresh window instance property to control the refreshing of windows. When set to kTrue screen updates are disabled and the window is not refreshed. You can use this property to improve performance, for example when setting a large number of exceptions for a complex grid. Setting the $norefresh property to kFalse will enable screen refreshing.

### Message Boxes

There are a number of message boxes you can use in your library to alert the user. The commands for these messages are in the *Message boxes*... group. They include

1. *OK message*   
   displays a message in a box and waits for the user to click an OK button. For emphasis you can add an info icon and sound the system bell. You can use square bracket notation in the message text to display the current value of variables or fields. For example, *OK message {[sys(5)]}* will display the user’s serial number. You can use the kCr constant enclosed in square brackets to force a line break in the message, e.g. 'First line[kCr]Second line'.
2. *Yes/No message*, and *No/Yes message*   
   displays a message in a box and waits for Yes or a No answer from the user. Either the Yes or the No button is the default. You can use the kCr constant enclosed in square brackets to force a line break in the message, e.g. 'First line[kCr]Second line'.
3. *Prompt for input*   
   displays a dialog prompting the user for input
4. *Working message*   
   displays a message while the computer is processing data or executing a method: with a Cancel button the user can break into the processing with Ctrl-Break/Cmnd-period/Ctrl‑C

## Events

Events are reported in Omnis as ***Event Messages***. These messages are sent to the event handling methods as one or more ***Event p***

***Parameters***. The first parameter of an event message, pEventCode, contains an *event code* representing the event. Event messages may contain a second, a third, or subsequent parameter that tell you more about the event. For example, a click on a list box will generate an evClick event plus a second parameter pRow telling you the row number clicked on. Note that all event codes are prefixed with the letters “ev”, and all event parameters are prefixed with the letter “p”. You can use the event codes in your event handling methods to detect specific events, and the event parameters to test the contents of event messages.

When an event occurs the *default action* normally takes place. For example, when the user presses the tab key to move to the next field on a data entry field, the default action is for the cursor to leave the current field and enter the next field on the window, and normally this is exactly what happens. However you could put a method behind the field that performs any one of a number of alternative actions in response to the tab. That is, the event handling method could use the tab to trigger a particular piece of code and then allow the default action to occur, it could pass the event to somewhere else in your library, or it could discard the event altogether and stop the default action from happening.

### Event Handling Methods

You can write an event handling method for each field and object contained in window, menu, toolbar, and report classes. The other class types do not generate events. You add the event methods for window and report fields in the *Field Methods* for the class. For menu classes you can add an event method to the *Line Methods* for a menu line, and for toolbar classes you can enter an event method in the *Tool Methods* for each toolbar control.

Events for remote forms and web components are described in the *Developing Web Applications* chapter in this manual.

Window fields, toolbar controls, and menu lines contain a default event handling method called $event, and report fields contain a default event handling method called $print. If you open the field methods for a window field, toolbar control, or menu line you will see an $event method, and for each report field you will see a $print method for the object. These are the default event handling methods for those objects.

To view the event handling method for a field or object

1. Show the design screen for the class
2. Right-click on the field, menu line or toolbar control
3. Choose Field Methods, Line Methods, or Tool Methods, as appropriate

The method editor opens showing the first method in the list for the field or object. If this is not the $event method, select it from the list to view it. Some event handlers will contain code to handle a range of possible events in the object.

The event handling method for some types of field may be empty, because there is only one possible event for the object. For example, the event handling method for a menu line is empty since you can only select a menu line. Therefore any code you put in the $event method for a menu line runs automatically when you select the line.

To enter the code for an event handling method

1. Assuming you have opened a default $event method for a field, click on the next command line after the *On* command

or, for an empty $event method

1. Select the first line of the method

For example, you can open the event method for a pushbutton, that contains a single *On* *evClick* command which will detect a click on the button.

1. Enter the code you want to run for that event

You could use the *Do* command and some notation in your event handling method, or you can use the *Do*method command to run another method in the current class or instance, or the *Do code method* command to run a method in a code class: in all cases, you can put literally any code in an event handling method and it will run given the right event.

#### The On Command

You can use the *On* command to detect events in your event handling methods. Fields from the Component Store may contain a default event handling method with one or more *On* commands to detect different events. For example, an entry field contains the method

###### On evBefore ## Event Parameters - pRow ( Itemreference )

###### On evAfter ## Event Parameters - pClickedField, pClickedWindow, pMenuLine, pCommandNumber, pRow

These lines detect the events **evBefore** and **evAfter**, which are the event codes contained in the message sent when the user enters or leaves the field, respectively. The in-line comments indicate which event parameters Omnis supplies for that event. In most cases, the event parameters are references containing values to do with the context of the event: the field clicked on, the list row number, the menu line number, and so on.

When you select the *On* command in the method editor, the list of possible events changes to reflect the events supported by the object to which $event belongs. You can also view events divided into categories, by using the Events tab of the Catalog.

You can use the default event handling method for a field or add your own. The following event handler for a data entry field detects an evBefore as the user enters the field and performs a calculation changing the value of the field.

###### On evBefore ## user tabs into date field   Calculate cDate as #D ## cDate is the dataname of the field   Redraw {DateField} ## the current field   Quit event handler

Code which is common to all events should be placed at the start of the event handling method, *before* any *On* commands. You can use the *On default* command to handle any events not covered by an earlier *On* command line. The general format is

###### # code which will run for all events On evBefore   # code for evBefore events On evAfter   # code for evAfter events On default   # code for any other events

When you enter the *On* command in an event handling method, it displays a list of all the available event codes in the command palette. You can click on the one you want, or you can enter more than one event code for a single *On* command, for example *On evClick, evDoubleClick*. *On* commands cannot be nested or contained in an *If* or loop construct*.*

When you have entered the *On* command line for a particular event and selected the next command line, you can open the Catalog to view the event parameters for that event code.

1. Click on the line ***after*** an *On evClick* command line
2. Open the Catalog (F9/Cmnd-9)
3. Click on Event Parameters under the Variables tab

For example, an *On evClick* command displays the parameters pEventCode and pRow in the Catalog. You can use these event parameters in your event handling methods to test the event message. A click on a list box generates an evClick event message containing a reference to the row clicked on, held in the pRow event parameter. You can test the value of pRow in your code

###### On evClick ## method behind a list box   If pRow=1 ## if row 1 was clicked on     # Do this...   End If   If pRow=2 ## if row 2 was clicked on     # Do that...   End If

All events return the parameter pEventCode, which you can also use in your event handling methods.

###### On evAfter,evBefore ## method behind field   # Do this code for both events   If pEventCode=evAfter     # Do this for evAfter events only   End If   If pEventCode=evBefore     # Do this for evBefore events only   End If

The parameters for the current event are returned by the *sys(86)* function, which you can use while debugging or monitoring which events are handled by which methods. For example, you could use the *Send to trace log* command and the functions *sys(85*) and *sys(86),* to report the current method and events, in the $event method for a field

###### # $event method for field 10 on the window Send to trace log {[sys(85)] - [sys(86)]} # sends the following to the trace log when you tab out of the field WindowName/10/$event - evAfter,evTab WindowName/10/$event - evTab

You can use any of the parameters reported for an event in your event handling methods. However, if you enter an event parameter not associated with the current event, the parameter will be null and lead to a runtime error.

### Mouse Events

Mouse events allow you to detect user mouse clicks inside fields and the background of a window. Mouse and right- mouse button events are generated only if the $mouseevents and $rmouseevents library preferences are enabled. Under macOS, right-mouse events are generated when you hold down the Ctrl key and click the mouse.

1. **evMouseDouble** and **evRMouseDouble**  
   the mouse, or right-mouse button is double-clicked in a field or window
2. **evMouseDown** and **evRMouseDown  
   evMouseUp** and **evRMouseUp**  
   the mouse, or right-mouse button is held down in a field or window, or the mouse button is released: for the mouse-down events you can detect the position of the mouse, se below
3. **evMouseEnter** and **evMouseLeave**  
   the mouse pointer enters, or leaves a field
4. **evDrag**  
   the mouse is held down in a field and a drag operation is about to start: the parameters report the type and value of the data
5. **evCanDrop**  
   whether the field or window containing the mouse can accept a drop: the parameters reference the object being dropped, the type and value of the data
6. **evWillDrop**  
   the mouse is released at the end of a drag operation. The parameters reference the object being dropped, the type and value of the data
7. **evDrop**  
   the mouse is released over the destination field or window at the end of a drag operation. The parameters reference the object being dropped, the type and value of the data

For the evMouseDown, evMouseUp, evRMouseDown and evRMouseUp events you can return the position of the mouse as the X-Y coordinates relative to the window background or field.

1. **pMouseX**   
   Mouse x coordinate
2. **pMouseY**   
   Mouse y coordinate
3. **pMouseHwnd**   
   window identifier of the hwnd receiving the mouse event: the mouse coordinate parameters are relative to this hwnd

The coordinate origin is the top-left of the hwnd.

#### Move Behind Events

From Studio 10, the $movebehind property allows you to control whether or not to allow mouse move events on fields in window classes, other than the top window. By default $movebehind is set to kTrue and will allow mouse move events to be processed in other windows when the window is the top window, e.g. evMouseEnter and evMouseLeave. Set the property to kFalse to turn off this behavior for the window.

In previous versions, only evMouseEnter and evMouseLeave in a complex grid in a window that was not top were allowed. To revert to the legacy behavior, add an entry called "oldMouseMoveBehindBehaviour" to the "defaults" group in the config.json file and set it to true.

### Discarding Events

In certain circumstances you might want to detect particular events and discard them in order to stop the default action from occurring. You can discard or throw away events using the *Quit event handler* command with the Discard event option enabled. Note however, you *cannot* discard some events or stop the default action from taking place since the event has already occurred by the time it is detected by an event handling method. In this case, a *Quit event handler (Discard event)* has no effect for some events.

Being able to discard an event is useful when you want to validate what the user has entered in a field and stop the cursor leaving the field if the data is invalid. The following method displays an appropriate message and stays in the field if the user does not enter the data in the correct format.

###### On evAfter ## as user leaves the field   If len(CustCode <> 6) ## check a value has been entered     If len(CustCode = 0) ## field left blank       OK message {You must enter a customer code}     Else      ## wrong length code entered       OK message {The customer code must have 6 digits}     End If

###### Quit event handler (Discard event) ## stay in the field   End If

You can also handle or discard events using the *Quit method* command with a return value of kHandleEvent or kDiscardEvent, as appropriate.

#### The Quit event handler Command

If you want to discard or pass an event you can use the *Quit event handler* command to terminate an *On* construct. A field event handling method might have the following structure.

###### # general code for all events On evBefore   # code for evBefore events On evAfter   # code for evAfter events On evClick,evDoubleClick   # code for click events   Quit event handler (pass event) On default   # code for any other events

The *Quit event handler* command has two options

1. **Discard event**for some events you can discard the event and stop the default action taking place
2. **Pass to other handlers**passes the event to the next handler in the chain

### Window Events

So far the discussion has focused on field events, which you would normally handle in the field using an event handling method. However you can enter methods to handle events that occur in your window as well. Like fields, the event handling method for a window class is called $event, and you enter this method in the *Class Methods* for the window class.

Window classes do not contain an $event method by default, but you can insert a method with this name. You enter the code for a window $event method in exactly the same as for fields using the *On* command to detect events in your window.

Window events affect the window only and not individual fields. They include clicks on the window background, bringing the window to the front or sending it to the back, moving it, sizing it, minimizing or maximizing the window, or closing it. For example, when you click on a window’s close box, the evCloseBox and evClose events are generated in the window indicating that the close box has been clicked and the window has been closed. You could enter an $event method for the window to detect these events and act accordingly.

The following window $event method detects a click on a window behind the current window, and discards the click if the user is inserting or editing data.

###### On evWindowClick ## user has clicked on a window behind   If cInserting | cEditing ## vars to detect current mode     OK message {You cannot switch windows while entering data}     Quit event handler (Discard event) ## keep window on top   End If   Quit event handler

The following window $event method checks for events occurring in the window and runs the appropriate methods elsewhere in the class. Note you cannot trap an evResize and discard it since the resizing has already occurred, but you can reverse the resizing by setting the size of the open window back to the size stored in the class.

###### On evToTop   Do method Activate   Quit event handler

###### On evWindowClick   Do method Deactivate   Quit event handler

###### On evClose   Do method Close   Quit event handler

###### On evResized   Do $cwind.$width.$assign($cclass.$width)   Do $cwind.$height.$assign($cclass.$height)   Quit event handler (Discard event)

#### Window Event Handling (macOS)

Under macOS, whenever the end-user clicks on a window title bar (or a button on the window title bar) the evWindowClick event is generated. The event parameter pStayingBehind is true if the window receiving the click will not come to the front as a result of the click (this event can only ever be true on macOS). For example, when the user clicks on the zoom box of a window that is not on top, the window will zoom or restore, but will not come to the top.

### Key events

You can detect which key or key combination is pressed by trapping the evKey event. The $keyevents library preference must be set to kTrue to enable the evKey event to be called for all window class foreground objects, including entry fields. The evKey event is sent to the target field when a key is pressed, and has three parameters as follows:

1. pEventCode  
   The event code
2. pKey  
   The key pressed
3. pSystemKey  
   The system key pressed represented by a code, as follows:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| 0 | Letter/Number | 26 | End |
| 1...12 | F1...F12 | 27 | Tab |
| 17 | Up Arrow | 28 | Return |
| 18 | Down Arrow | 29 | Enter |
| 19 | Left Arrow | 30 | Backspace |
| 20 | Right Arrow | 32 | Esc |
| 21 | Page Up | 34 | Delete |
| 22 | Page Down | 35 | Insert |
| 25 | Home | 53 | Context Menu |

### Control Methods and Passing Events

As already described, you handle events for fields using an event handling method contained in the field, but you can add a further level of control over field events by adding a method called $control to your window. This method is called a *window control method*. To allow this method to handle events you must pass events to it from the field event handling methods. You do this by including in your field event handler the *Quit event handler* command with the **Pass to next handler** option enabled.

As a further level of control, you can add a method called $control to your tasks. This method is called a *task control method*. Events are passed to the task control method from the window control method contained in the window belonging to the task. Therefore, an event may be generated in the field, passed to the window control method, and then passed to the task control method.

Window events that are handled in the $event method for a window can be passed to the task $control method as well.

At each level an event handling method can discard the event or pass it on to the next event handler. At the task level, the highest level of control, the event can be processed and the default action takes place, or the event can be discarded and no further action occurs.

The Omnis event processing mechanism gives you absolute control over what is going on in your application, but it also means you need to design your event handling methods with care. It is important not to pass on an event to higher levels unnecessarily and to keep control methods short, to limit the time spent processing each event.

In the following example, the $control method is contained in an Omnis data entry window. It sets the main file for the window when it is opened or comes to the top, and does not let the user close the window if Omnis is in data entry mode.

###### On evToTop   # window comes to the top or is opened   Set main file {FCUSTOMERS}   Quit event handler

###### On evClose   If cInserting | cEditing ## vars to detect current mode     # User closes window when in enter data mode     OK message {You can't close in enter data mode}     Quit event handler (Discard event)   End If

#### Event Processing and Enter Data Mode

Normally, the default processing for an event takes place when all the event handler methods dealing with the event have finished executing. It is not possible to have active unprocessed events when waiting for user input so the default processing is carried out for any active events after an *Enter data* command has been executed or at a debugger break. Therefore if required, you can use the *Process event and continue* command to override the default behavior and force events to be processed allowing an event handling method to continue.

The ***Process event and continue*** *(Discard event)* option lets you discard the active event. For example, in an event handler for evOK the following code would cause the OK event to be thrown away before the subsequent enter data starts.

###### On evOK **Process event and continue** (Discard event)   Open window instance {window2}   Enter data

### Container Fields and Events

Container fields are fields that contain other fields: examples of container fields include subwindows, tab panes, page panes, scroll boxes, and complex grid fields. The logic for handling and passing events within a container field is the same as for simple fields, it just has more levels of control.

For the purposes of event handling, you can regard the container field as both a field on the parent window, and a window since it contains other fields. In this respect, a container field can have an $event method that handles events for the container field itself, and a $control method that handles events passed to it from the individual fields inside the container field. Each field in the container field has a $event method to handle its own events. If the control method for your container field allows it, events are passed to the parent window control method, which in turn can be passed onto the task control method or discarded as appropriate.

You can nest container fields such as subwindows and tab panes, but nested container fields do not pass events.

### Queuing Events

Some user actions generate a single event which is handled as it occurs by your event handling methods. The event may be dealt with completely in the field or it may be passed up the event chain as required. However some user actions generate a whole series of events, one after another. These events are placed in an *event queue*. Each event is handled by your event handling methods strictly in turn on a first-in, first-out basis. For example, when the user tabs from one field to another the current field is sent an evAfter and then an evTab event, then the new field is sent an evBefore event: all these events are placed in the event queue in response to a single user action, the tab. Similarly when you close a window, the current field is sent an evAfter, the window is sent an evCloseBox event, then it is sent an evClose event. Each one of these events is sent to the appropriate object and is handled by your event handling methods *before* the next event in the queue is handled.

In addition to events generated by user actions, you can append an event to the event queue using the *Queue* commands in the *Events...* group.

###### Queue bring to top Queue close Queue cancel Queue set current field Queue click Queue double-click Queue keyboard event Queue OK Queue scroll (Left|Right|Up|Down) Queue tab Queue quit

These commands let you simulate user actions such as key presses and clicks on buttons or windows. For example, the *Queue bring to top {Windowname}* command brings the specified window instance to the top and simulates a user clicking behind the current window. Events generated by these commands are handled after those that are currently queued. You can queue several events in succession.

## Using Tasks

Omnis contains two environments, a *design mode* and a *runtime mode*. In design mode, you can create and store classes in your library. In runtime mode, various objects or instances are created as you run your application. You can group and control the runtime objects in your application by opening or instantiating them in a *task*. You can manipulate whole groups of instances by manipulating their task, rather than having to deal with each separate instance. You define a task in your library as a *task class.*

Task classes can contain variables and methods, and you can define custom properties and methods for a task class as well. When you open a task you create an instance of that task class. The task instance is unique in Omnis in that it can contain other instances including window, report, toolbar, and menu instances. Task instances cannot contain other tasks. When you open an instance from within a task it *belongs to* or is *owned by* that task.

By opening and closing different tasks, or by switching from one task instance to another, you can control whole groups of objects. Omnis provides certain default actions which happen as the task context switches. You define exactly what happens in the task by creating methods in the task class. For example, in the task you can specify which windows are opened and which menus are installed using commands or the notation.

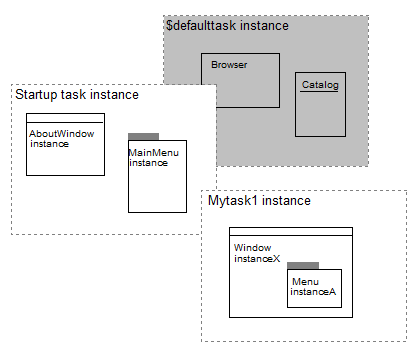
Each library contains a group of task classes called $tasks, and Omnis has a group containing all open task instances called $root.$itasks in the order that they were opened.

For web applications, Omnis provides remote tasks to handle client connections and remote form instances. These are described later in this chapter.

### Default and Startup Tasks

When Omnis opens, it creates a task instance for the IDE to run in. This task is called the *default task*, and is represented in the notation as $root.$defaulttask. This task instance contains all the IDE objects such as the Browser, Catalog, Property Manager, and so on.

When you create a new library, it contains a task class called *Startup\_Task* by default*.* When you open a library, an instance of the startup task is created automatically. From there on all instances opened in the library are owned by the startup task. You can delete the startup task, or you can create other tasks for your application components to run in.



It is not essential to add tasks to your library, your library will safely execute in the startup task, or the default task along with the IDE objects.

The startup task instance has the same name as your library. For a simple application, the startup task will probably be all you need, with all the other class instances belonging to it. The startup task remains open for as long as the library is open, but you can close it at any time using a command or the notation. You can change the name of the task to be opened on startup by setting the library preference $startuptaskname: for all new libraries this is set to Startup\_Task by default.

If you have an application that spans multiple libraries, often only the library used to start the application will have a startup task. If a library is opened using the *Open library* command with the option *Do not open startup task*, the startup task is not instantiated. In design mode, you can stop a library’s startup task from running if you hold down the Alt/Option key as you open your library.

#### Handling application focus events

You can control application focus events generated by the Operating System using the task methods $omnistofront and $omnistoback. These methods can be added to the Startup task in your library.

1. **$omnistofront**  
   called when Omnis is brought to the front by the Operating System as a result of a user action such as a mouse click or alt-tab key combination. On macOS, an extra boolean parameter is passed to $omnistofront. If the value of this parameter is kTrue, the user has clicked on the Omnis Icon in the Dock. If a user clicks on the Omnis Dock Icon, $omnistofront will be called twice. The first call will be generated as a result of an OS AppActivated event and the parameter value will be kFalse. The second call will be generated as a result of the Dock click and the parameter value will be kTrue.
2. **$omnistoback**  
   called when Omnis is sent to the back by the Operating System as a result of a user action such as a mouse click or alt-tab key combination. For $omnistoback you should only use Omnis code which produces a non-visual result. Attempting to open windows, and so on, while Omnis is losing the application focus may have undesirable affects and is not supported.

### Creating Task Classes

This section describes how you create a task class from the Browser.

To create a task class

1. Open your library and select it in the Browser
2. Click on New Class and then the Task option
3. Name the new task
4. Double-click on the task class to modify it

You modify a task class in the method editor. You can place in the $construct method any code that you want to run when the task is opened. For the Startup\_Task, the $construct method is executed when you open your library. You can add any other custom properties and methods to the task, as well as any type of variable.

### Opening Tasks

Apart from the startup task instance, which is opened automatically when you open your library, you can open a task using the *Open task instance* command or the $open() method. Any parameters you supply with the command are sent to the task’s $construct method.

###### Open task instance MyTask/TaskInstance2 (p1,p2,...) # opens the task, assigns an instance name, and sends parameters

Alternatively you can open a task instance using the $open() method.

###### Do MyTask.$open(‘TaskInstance2’,p1,p2,...) Returns iTaskRef # does the same as above & returns a reference to the task instance

### Current and Active Tasks

Omnis keeps references to two different tasks, the *active task* and the *current task,* to keep track of the tasks that own the topmost instance or GUI object and the currently executing method. The active task is the task that owns the topmost open window, installed menu, or toolbar currently in use. The current task is the task that owns the currently executing method.

A *task context switch* occurs when Omnis changes the current or active tasks. As Omnis runs your library, the current and active tasks may point to different task instances depending on the user’s actions.

#### The Active Task

The active task is affected by the user, and is typically the task containing the topmost open window. When an instance belonging to another task is selected, Omnis performs a task context switch. As part of the context switch, messages are sent to both tasks. The active task gets sent a $deactivate() message, and the new active task is sent an $activate() message.

When the active task changes, you can use the $activate() and $deactivate() messages to perform other relevant actions such as hiding other windows, installing menus, and any other requirements your application has.

In order for Omnis to perform an automatic task context switch when the user selects an instance belonging to another task, the task’s $autoactivate property must be set to kTrue.

Omnis can install and remove menus and toolbars automatically during task context switches. Menu and toolbar instances each have a $local property that you can set. When set to true, the menu or toolbar instance is made local to the task that owns it. When a task context switch occurs, local menus for the previously active task will be removed from the menu bar, and any local menus instances owned by the new active task will be installed. Toolbars behave similarly. If the tasks use different docking areas, Omnis will not hide the docking areas, only the toolbars.

You can change the active task using the notation, rather than waiting for the user to initiate a task context switch. To do this, you can set the property $root.$activetask to a different task instance name to switch tasks.

#### The Current Task

The current task is under the control of Omnis itself, and is the task instance which contains the currently executing method. When a custom attribute or event is sent to an instance, the current task is switched to the task which owns the instance, and when control returns from that attribute or event, the previous task is restored.

When the current task changes, messages are sent to both tasks. The current task is sent a $suspend() message, and the new current task gets a $resume() message. If the new current task is being instantiated for the first time, it gets a $construct() message rather than the $resume().

In order to avoid endless recursion a task does not get suspend or resume messages during the execution of a suspend or resume method.

Since $suspend() and $resume() are likely to be called frequently, it is important that the code for them should be kept as short and efficient as possible and should not:

1. alter the user interface
2. open or close an instance
3. switch tasks

You can find out the name of the current task using the notation $ctask().$name, and the task that owns the instance by using InstanceName.$task().$name.

### Closing Tasks

You can close a task instance using the *Close task* command or the $close() method. When you close a library all its task instances are closed, and when you quit Omnis the default task is closed and all instances belonging to the default task are closed.

When you close a task, all instances belonging to that task are closed or destructed providing they can be closed. When instances are closed, a message is sent to the instance asking it to confirm whether or not it can be closed. If the instance returns a false message, Omnis will not close that instance. For tasks, each instance belonging to the task is sent the message, and then the task itself is sent the message. If any of the instances belonging to the task cannot be closed, none of the instances nor the task instance are closed.

### Task Variables

Task classes can contain both class and instance variables of any standard Omnis data type. Tasks can also contain *task variables*, which are accessible to any instance owned by the task. As with other variables, you create task variables in the variable pane of the method editor.

When two or more types of variable use the same variable name (this is not recommeded), a reference to that variable may be ambiguous. In this situation, Omnis uses the variable with the smallest scope automatically. All other variable scopes have precedence over task variables.

When a method in a code class is called from another class or instance using the *Do code method* command, the current task continues to point to the calling method. This allows methods in a code class to have access to the task variables from the calling method.

#### The Design Task

In order for task variables to be available to you for use in design mode, you must establish a connection between a class and the task whose variables you want to access. You do this by setting the design task ($designtaskname property)for the class. The design task determines which task variables are available to the class: if no design task has been set, the method editor does not let you declare or see any task variables.

Setting the design task for a class doesn’t guarantee that the task will be available in runtime when you open your class, nor will Omnis automatically create an instance of the task. The design task is simply a way to give you access to a set of task variables while you create the classes in your library.

You can also access task variables without setting a design task by referring to the variable as $ctask.variablename. This assumes that the variable will always belong to a task and can therefore default to the current task.

If you attempt to access a task variable in an instance, and that variable is not available in the task, a runtime error of ‘Unrecognized task variable’ will be generated, and the variable will have a NULL value.

If you rename a task variable, any references to it are not renamed. Also if one with that name ceases to exist, references to it which were entered as VariableName are shown in design mode as $ctask.VariableName. Similarly, if some code containing a task variable is pasted into a different class, any task variables used by that code are not copied into the destination class.

### Private Instances

Normally an instance is visible to other tasks and you can reference it using the notation from anywhere in your library. However you can override this default behavior by making an instance *private* to the task that owns it. You can do this by setting the instance’s $isprivate property to kTrue.

When you make an instance private, you cannot destruct it, make references to it, or even see it unless you are within the task that owns it. A task can even be private to itself, so it can be closed only when it is the current task. If access to a private instance is required from outside of the task, an item reference can be set to the instance, and the item reference can be passed outside of the task. Once this has occurred, the item reference can be used to manipulate the instance.

The $root object has a series of object groups, one for each instance type, that are represented in the notation as $iwindows, $imenus, $itoolbars, $ireports, $itasks. Each of these object groups displays all public instances, as well as instances which are private to the current task. As the current task changes, the contents of these groups may change to reflect the private instances present in your library.

### Private Libraries

Libraries can be private to a task, and both the library and its classes are visible only to that task.

The group of open libraries, $libs, contains a private library only when the task which owns that library is the active task. The Browser does not display classes from a private library. Standard entry points to the debugger such as shift-click on a menu line do not enter the debugger if the menu belongs to a private library.

As with private instances, if an item reference to any object within a private library is passed to an object outside the library, it is able to access the library using notation.

You can make a library private by setting its $isprivate property to true. This is normally done immediately after opening the library, but can be done at any time as long as the task which owns the library is the active task. Libraries also have the $alwaysprivate property, which, if set, means they are always and immediately private to their startup task.

Private libraries have an additional property, $nodebug, which keeps the debugger from being entered for any reason when code from that library is executing, including errors, breakpoints, and the stop key. Code from a private library with $nodebug set does not appear in the stack menu or the trace log.

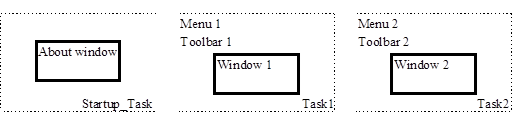
When a task is closed, it closes all its private libraries unless they cannot be closed. This can occur if, for example, the library has instances belonging to other tasks. If a private library cannot be closed, it will become non-private.

### Multiple Tasks

When designing an application, you might want to partition your library by creating modules containing all of the windows, reports and methods of like functionality. Each module can have its own menus and toolbars. An example containing such modules might be an accounting package, with General Ledger, Accounts Payable and Accounts Receivable modules.

In a totally modal application, where the user switches between modules, it is easy to ensure that the user sees the correct menus and tools for the current module. In a modeless, multi-window environment, controlling this can sometimes be difficult. Tasks automate the process of creating modular applications by providing all the management of menus and tools for you.

Consider the following example in which a single library is running three tasks: the *Startup\_Task* and two user tasks *Task1* and *Task2*. The startup task, which opens automatically when the library opens, contains an About window. The other two tasks each contain a window, a menu, and a toolbar. When the user selects a window from either Task1 or Task2, you may want Omnis to display the correct tools and menus for that window automatically.



When the library opens, the startup task opens and displays the About window and then opens the other tasks, each of which opens its window and installs its menu and toolbar. The startup task can close itself once the About window is closed if it’s no longer needed.

To open the two tasks, you should execute the following in the $construct method of the startup task

###### Open window instance AboutWindow Open task instance MyTaskClass1/Task1 Open task instance MyTaskClass2/Task2 Close task instance LibraryName ## close Startup\_Task instance

Every task has a property $autoactivate, that allows the task to take control whenever the user tries to bring a window it owns to the front. If the property is set to false, the window won’t come to the front. To activate each task automatically, you need to execute the following in the $construct of each task

###### Do $ctask.$autoactivate.$assign(kTrue)

To ensure that your menus and toolbars show and hide appropriately as the tasks change, you need to set the $local property for each class. By making each menu and toolbar local to the task that owns it, Omnis hides and shows them automatically as the task context changes.

In the $construct for a task, you can install your menu and toolbar, and set their $local property. For example

###### # $construct for task1... Do $menus.MyMenuClass1.$open(‘Menu1’) Returns iMenuRef Do iMenuRef.$local.$assign(kTrue) Do $toolbars.MyToolbarClass1.$open(‘Toolbar1’) Returns iToolRef Do iToolRef.$local.$assign(kTrue) Do $windows.MyWindowClass1.$open(‘Window1’) Returns iWinRef

You can do the same for the other task.

###### # $construct for task2... Do $menus.MyMenuClass2.$open(‘Menu1’) Returns iMenuRef Do iMenuRef.$local.$assign(kTrue) Do $toolbars.MyToolbarClass2.$open(‘Toolbar1’) Returns iToolRef Do iToolRef.$local.$assign(kTrue) Do $windows.MyWindowClass2.$open(‘Window1’) Returns iWinRef

This functionality will change menus and toolbars as you switch from one window to the other.

### Preferences on macOS

The macOS application menu has a Preferences item. You can arrange for Omnis to call a method in a task, when the user selects this menu item. To do this, define a method called $prefs in your task. When the user selects Preferences from the application menu, Omnis calls the $prefs method.

If more than one task defines $prefs, Omnis installs a hierarchical menu on the Preferences menu item. Each task has an item in the hierarchical menu. In this case, each task must also contain a method called $prefsmenutext. This method must return the text to display for the task’s item in the hierarchical menu, for example

###### Quit Method “My Library”

## External Component Notation

The $components group under $root contains all the installed external components available in your XCOMP folder. You can view the contents of the $components group using the Notation Inspector.

Note that you manipulate an external component via its custom field properties, as shown below, not via the $root.$components...$compprops or $compmethods groups for the control. The groups under $root.$components is simply a convenient way of viewing the contents and functions of any external library or control.

The $components group has the standard group properties and methods, including $add() and $remove(), and you can list the components using the $makelist() method.

###### # declare variable cCompList of type List Do $root.$components.$makelist($ref.$name) Returns cCompList

You can drag a reference to any of the components from the Notation Inspector to your code, in the same way as other built-in objects. You can click on a component library in the Notation Inspector and view its properties in the Property Manager. Each component library has the following properties

1. $name  
   the name of the component which must be unique
2. $pathname  
   the name and path of the external library file: this will vary across different platforms
3. $functionname  
   the name of the external function
4. $controlhandler  
   Boolean that indicates whether the external is a control handler, for example, an ActiveX is a control handler
5. $constprefix  
   String used as a prefix for all constants within the external
6. $flags  
   indicates the external flags, for example, whether it is loaded
7. $usage  
   Current number of controls that are using this external
8. $version   
   the version information

You can view the contents of an external library in the Notation Inspector. Each component library has a group called $controls containing all the controls in the library. Some libraries may contain only one control, for example, the Slider Component Library contains the Slider Control only. A control contains its own events, functions (or methods), and properties in their own respective groups, as follows

1. $compevents  
   group of events for the control
2. $comprops  
   group of properties for the control
3. $compmethods  
   group of methods for the control

In the notation you treat an external component property or function as you would a standard built-in property or method, that is, you can use property and method names in the notation to manipulate and send messages to an external component field. Note that property and method names should include a dollar sign when you use them in the notation.

###### Do $cwind.$objs.ClockField.$facecolor.$assign(kBlue) # assigns a color to the face of a clock component # using the $facecolor property Do $cwind.$objs.QTfield.$Play() # executes the $Play() function for a QuickTime component

In general, the properties of an external component are unique to the object and their names will not clash with standard Omnis field properties. However when an external component property has the same name as an Omnis property, you must access the external property using a double colon (::) before its name. For example, the Icon Field control has the property $backcolor which you must refer to using

###### Do $cinst.$objs.iconfield.$::backcolor.$assign(kRed) # would not work without the ::

At runtime you can add an external component to an open window using the $add() method. You need to specify the kComponent object type, external library name, external control name, and the position of the field. For example, the following method adds the Marquee Control to the current window instance, positions the new object, and sets some of its properties

###### # declare local variable Objref of type item reference Do $cinst.$objs.$add(kComponent,'Marquee Library','Marquee Control',0,0,15,100) Returns Objref Do Objref.$edgefloat.$assign(kEFposnStatusBar) # repositions the object at the bottom of your window Do Objref.$message.$assign('I hope you can read quickly!') Do Objref.$steps.$assign(20) ## number of pixels to step Do Objref.$speed.$assign(20) ## lower number is faster Do Objref.$::textcolor.$assign(kBlue) ## note :: notation Do Objref.$::backcolor.$assign(kRed)

#### Version Notation

All external components have the $version property. To get the version of an external component you must access it via the $root.$components group, not the external component field on a window or report. For example

###### Do $root.$components.Marquee Library.$version Returns lvXversion # returns “1.2” for example

If you have created any external components of your own to run under Omnis Studio version 1.x, you must recompile them for Omnis Studio 2.0.

#### Java Beans

Note the Java Bean ext comp is not longer installed in Studio 10, but it can be obtained by contacting support.

The Java Bean external component has commands that let you control it in a Runtime Omnis. Note the Java Beans external is available under Windows only.

You request a command using the $cmd() method as follows:

###### $root.$components.JavaBean.$cmd(parameter list)

The parameters can be:

|  |  |
| --- | --- |
| Parameter list | Command |
| "GetPaths", List | Populates the specified single column list with the Java Bean search paths: no return value |
| "AddPath", NewPath | Adds the specified path to the Java Bean search paths: returns true for success, or if the path is already present in the search paths |
| "DeletePath", DelPath | Deletes the specified path from the Java Bean search paths: returns true for success |
| "EnumBeans" | Enumerates Java Beans: returns the number of Beans found |
| "StartVM" | Starts the Java virtual machine (to test if Java is installed): returns a string containing an error, or an empty string to indicate success |
| "SetupDialog". | Opens the Java Bean component setup dialog |
| "RequestPath" | Opens the “Prompt for Java Bean Path” dialog: returns a string containing the new path: empty if none selected |

### Debugging errors in the OCX handler

If you experience problems while using an Omnis component due to errors in the OCX handler (OCXHNDLR) you should report this problem to Omnis Support. The problem is usually caused by a faulty External DLL, but it may be difficult to detect which one is causing the problem.

To diagnose the problem, create a text file called OCXHNDLR.LOG and place it in the XCOMP folder. This file will log all the activity in the OCX handler during the lifetime of the Omnis session, including the load status of each external DLL that Omnis attempts to load.

If you contact Omnis Support, please include the OCXHNDLR.LOG file so we can diagnose the problem. Ensure that you delete the file when logging is no longer required, as performance will be affected.