# Chapter 5—Object Oriented Programming

This chapter discusses the object-oriented features used in Omnis Studio, including inheritance, custom properties and methods, and creating and using object classes and external objects.

## Inheritance

When you create a new class you can derive it from an existing class in your library. The new class is said to be a subclass of the existing class, which is in turn a superclass of the new class. You can make a subclass from all types of class except code, schema, file, and search classes. The general rule is that if you can open or instantiate a class, you can make a subclass of that type of class. Omnis does not support mixed inheritance, that is, you cannot make a subclass of one type from another type of class.

When you make a subclass, by default, it inherits all the variables, methods, and properties of its superclass. Window subclasses inherit all the fields and objects on the window superclass, and menu and toolbar subclasses inherit all menu lines and tools from their respective superclass.

Inheritance saves you time and effort when you develop your application, since you can reuse the objects, variables, and methods from a superclass. When you make a change in a superclass, all its subclasses inherit the change automatically. From a design point of view, inheritance forces uniformity in your GUI by imposing common properties, and you get a common set of methods to standardize the behavior of the objects in your library.

### Making a Subclass

You can make subclasses from the following types of class.

1. **Window**  
   inherits variables, methods, and properties from its superclass, as well as all fields on the window superclass
2. **Menu**  
   inherits variables, methods, and properties from its superclass, as well as the menu lines in the menu superclass
3. **Toolbar**  
   inherits variables, methods, and properties from its superclass, as well as the toolbar controls in the toolbar superclass
4. **Report**  
   inherits variables, methods, and properties from its superclass: note that a report class *does not* inherit the fields, sections, and graphics from its superclass
5. **Task**  
   inherits variables and methods from its superclass, but none of its properties
6. **Table**  
   inherits variables and methods from its superclass, and only some of its properties
7. **Object**  
   inherits variables and methods from its superclass, but none of its properties

To make a subclass

1. Open your library in the Browser and show its classes
2. Right-click on a class and choose Make Subclass from its context menu

When you make a subclass Omnis creates a new class derived from the selected class. The new class inherits all the objects, variables, and methods from its superclass. Omnis supports up to 10 superclass levels, that is, a single class can inherit objects from up to ten other superclasses that are directly in line with it in the inheritance tree. If you create further levels of subclass they do not inherit the objects from the superclasses at the top of the tree.

You can view and edit a subclass, as you would any other class, by double-clicking on it in the Browser. When you edit the methods for a subclass in the method editor, you will see its inherited variables and methods shown in a color. When you view the properties of a subclass its inherited properties are shown in a color in the Property Manager. You can set the color of inherited objects using the **inheritedcolor** Omnis preference. Note that all the Appearance and Action properties for a window are inherited too.

Standard properties for the class, such as the window name and class type, are not inherited, neither are the grid properties. The properties to do with the general appearance of the window, such as **title** and **hasmenus**, are inherited and shown in a color, which defaults to bright blue. You cannot change inherited properties unless you overload them: their values are grayed out in the Property Manager. If you change the properties of the superclass, the changes are reflected in the subclass when you next open it in design mode.

There are some general properties of a class that relate to inheritance. These are

|  |  |
| --- | --- |
|  |  |
| **superclass** | the name of the superclass for the current class; the superclass can be in another library, in which case the class name is prefixed with the library name |
| **inheritedorder** | for window classes only, determines where the inherited fields appear in the tabbing order: by default inherited fields appear before non-inherited fields on your window |
| **issupercomponent** | if true the class is shown in the Studio Browser as a superclass |
| **componenticon** | icon for the superclass when it is shown in the Studio Browser |

#### Making a Subclass Manually

You can set the **superclass** property for a class manually, to make it a subclass of the specified class, either using the notation or in the Property Manager.

However when you make a subclass in this way it *does not inherit* any of the properties of the superclass. Only objects are inherited from the superclass. You have to open the Property Manager and inherit the properties manually using a context menu.

To inherit a property manually

1. View the properties of the subclass in the Property Manager
2. Right-click on the property and select Inherit Property from its context menu

If the property cannot be inherited the context menu will display Cannot Inherit and will be grayed out. If the class does not have a superclass the inheritance context menu does not appear or is grayed out.

To inherit a method manually

1. Open the method editor for the subclass
2. Right-click on the method and select Inherit Method from its context menu

When you inherit a method in this way, Omnis will delete the current method. Any comments in the inherited method will be shown on the left-hand side of the ‘Notes’ tab in the Variable pane in the Method Editor.

#### Inherit or Override method Shortcut

The Inherit method or Override method options are present in the method editor Modify menu when it is appropriate to include the command. Both have the shortcut Ctrl+Shift+I to inherit or override the current method.

### Overloading Properties, Methods, and Variables

Having created a class from another class using inheritance you can override or overload any of its inherited properties, methods, and variables in the Property Manager or the method editor as appropriate. All inherited objects are shown in a color. To overload an object, you can Right-click on the object and select Overload from the object’s context menu. Note that for windows, menus, and toolbars you cannot overload or delete inherited fields, menu lines, or toolbar controls. If you don’t want certain objects to be displayed in a subclass you can hide them temporarily at runtime using the notation.

To overload a property

1. View the properties of the subclass in the Property Manager
2. Right-click on the inherited property and select Overload Property from its context menu

When you overload a property its inherited value is copied to the class and becomes editable in the Property Manager. You can overload any number of inherited properties in the class and enter values that are local to the class.

To reverse overloading, you Right-click on a property and select Inherit Property: the local value will be overwritten and the value inherited from the superclass will appear in the Property Manager.

To override a method

1. View the methods for the subclass in the method editor
2. Right-click on the inherited method and select Override Method from its context menu

When you override a method it becomes like a non-inherited method in the class, that is, you can select it and add code to it.

To reverse this process and inherit the method with the same name from the superclass, you can right-click on the method and select Inherit Method: the code in the non-inherited method will be deleted and the inherited method will now be in place. Alternatively, if you override a method and then delete the local one, the inherited method will reappear when you close and reopen the class.

To override a variable

1. View the methods for the subclass in the method editor and click on the appropriate tab in the variable pane to find your variable
2. Right-click on the inherited variable and select Override Variable from its context menu

When you override a variable it becomes like a non-inherited variable in the class and is only visible in that class.

To reverse this process and inherit the variable with the same name from the superclass, you can Right-click on the variable and select Inherit Variable.

### Inheritance Tree

The Inheritance Tree shows the superclass/subclass structure of the classes in your library. All classes below a particular class in the hierarchy are subclasses of that class. When you select a class in the Browser and open the Inheritance Tree it opens with that class selected, showing its superclasses and subclasses above and below it in the tree.

To open the Inheritance Tree for a class

1. Select the class in the Browser
2. Right-click on the class and select Inheritance Tree from its context menu

### Showing Superclasses in the Studio Browser

You can show any class that supports inheritance in the Studio Browser by setting its $issupercomponent property to kTrue. You can specify the icon for a superclass by setting its $componenticon property. If you create a class from a superclass displayed in the Studio Browser, the new class will be a subclass of the class in the Studio Browser automatically.

To make a subclass from a supercomponent

1. Select the target library in the Studio Browser
2. Click Class Wizard option
3. Click on the Class Type option
4. Click on the Subclasses option

At this stage, all the supercomponents of that type (i.e. classes that have their $issupercomponent set to kTrue) should be displayed in the wizard.

1. Click the supercomponent (superclass) you want to create a subclass of
2. Enter a name for the subclass and click on the Create button

Such supercomponents will only appear in the Studio Browser if the library containing it is open, since the class actually remains in your library and is only displayed in the Studio Browser.

Note that classes that appear in the Studio Browser in this way cannot be made the default object for that class.

### Inheritance Notation

You can use the $makesubclass() method to make a subclass from another class. For example

###### Do $windows.Window1.$makesubclass(‘Window2’) Returns ItemRef # creates Window2 which is a subclass of Window1 in the # current library, ItemRef contains a reference to the new class

You can test if the current class can be subclassed by testing the $makesubclass() method with the $cando() method, as follows

###### If $cclass.$makesubclass().$cando() Do $cclass.$makesubclass(sClass) Returns ItemRef # creates a subclass of the current class

The $makesubclass() method has a Boolean optional second argument (bAddLibPrefixForClib, default kFalse), which when kTrue, causes the $superclass property of the new subclass to include the library prefix for the current library.

You can test if a particular class is a superclass of another class using the CLASS.$isa(SUPERCLASS) method as follows

###### Do $windows.window2.$isa($windows.window1) Returns lresult # returns true if window1 is a superclass of window2

You can change the superclass of a class by reassigning $superclass

###### Do $cclass.$superclass.$assign(‘DiffClassName’) Returns lresult

You can test if a property can be inherited using

###### Do $cclass.$PropertyName.$isinherited.$canassign() Returns lresult

If a property can be inherited, you can overload or inherit the property by assigning $isinherited. For example, to overload a property

###### Do $cclass.$PropertyName.$isinherited.$assign(kFalse) Returns lresult

A superclass can be in a different library to a subclass. If you open an instance of a subclass when the superclass is not available, perhaps because the library the superclass belongs to is not open or has been renamed, a kerrSuperclass error is generated.

#### Libraryname prefix for subclasses

By default, Omnis does not prefix $superclass with the library name (if both classes are in the same library).  You can manually change the $superclass property to include the library name if you wish.

#### Calling Properties and methods

When a property or method name is referenced in a subclass, Omnis looks for it first in the subclass and progressively up the chain of superclasses in the inheritance tree for the current library. Therefore if you haven’t overridden the property or method in the subclass, or at any other level, the property or method at the top of the inheritance tree will be called. For example, the following command in a subclass

###### Do $cinst.$MethodName() # will call $MethodName() in its superclass

However, if you have overridden a property or method in the subclass the method in the subclass is called. You can still access the inherited property or method using the $inherited property. For example, assuming $MethodName() has been overridden in the subclass

###### Do $cinst.$inherited.$MethodName() # will call the subclass method Do $windows.MySubWin.$MethodName().$inherited # will call $MethodName() in the superclass

#### Referencing Variables

When a variable is referenced in a subclass, Omnis looks for its value first in the subclass and progressively up the chain of superclasses in the inheritance tree for the current library. Therefore if you haven’t overridden the variable in the subclass, or at any other level, the value at the top of the inheritance tree is used.

However, if you have overridden a variable in the subclass the value in the subclass is used. You can access the inherited variable using $inherited.VarName, in which case, the value at the top of the inheritance tree is used.

A superclass cannot use the instance and class variables of its subclasses, although the subclass can pass them as parameters to superclass methods. References to class and instance variables defined in a superclass are tokenized by name and the Remove Unused Variables check does not detect if a variable is used by a subclass. If an inherited variable does not exist, its name is displayed as $cinst.VarName in design mode and will create a runtime error.

#### Inherited Fields and Objects

All inherited window fields on a window subclass are included in the $objs for an instance. Since some field names may not be unique, when $objs.name is used Omnis looks first in the class containing the executing method, then in the superclasses and lastly in the subclasses. The $objs group for report fields, menu lines, and toolbar controls behave in the same way as window fields.

You should refer to fields at runtime by name, since at runtime Omnis assigns artificial $idents to inherited fields. The $order property of a field may also change at runtime to accommodate inherited fields into the tabbing order.

#### Do inherited Command

You can use the *Do inherited* command to run an inherited method from a method in a subclass. For example, if you have overridden an inherited $construct() method, you can use the *Do inherited* command in the $construct() method of the subclass to execute the $construct() method in its superclass. You could use this command at the end of the $construct() method in the subclass to, in effect, run the code in the subclass $construct() and then the code in the superclass $construct() method.

#### Inherited Object Notation

You can access the methods of inherited objects in remote forms, windows, menus, toolbars and reports in your Omnis code. Inherited objects are exposed as a new group within the notation group called $inheritedobjs. The members of this group are the inherited objects from all of the superclasses of the class.

Each member of the $inheritedobjs group has three properties: $name, $ident and $isorphan (true when the object no longer belongs to a superclass, but has methods so it cannot be removed without developer approval). In addition, each member has a child $methods group which are the methods implemented in the class for the inherited object. This is just like any other methods group, and methods can be manipulated as you would expect. To override a method from the superclass, simply add a method with the same name as the inherited object method. To inherit a method, delete the method with the same name from the inherited object methods.

## Custom Properties and Methods

You can add methods to the objects in your library and call them what you like; you execute these methods from within the class or instance using the *Do method* command.

You can also create your own properties and methods and execute them using the notation, as you would the standard properties and methods. These are called *custom properties* and *custom methods*, or collectively they are referred to as *custom notation*. Custom notation can only be executed at runtime, in an instance of the class, and applies either to the instance, or an object contained in the instance.

The name of a custom property or method is case-insensitive. It can be any name starting with the dollar “$” sign, except for the set of names in the following table.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| $add | $addafter | $addbefore | $appendlist |
| $assign | $att | $attcount | $attributes |
| $canassign | $cando | $canomit | $chain |
| $chaincount | $class | $count | $data |
| $default | $deletelines | $desc | $findident |
| $findname | $first | $fullname | $group |
| $ident | $insertlines | $insertlist | $inst |
| $isinherited | $last | $lib | $minparms |
| $makelist | $maxcachedclasses | $maxparms | $method |
| $name | $next | $prev | $obj |
| $ref | $remove | $runapplescript | $sendall |
| $serialize | $sublen | $subtype | $type |
| $wind |  |  |  |

Any class that can be instantiated can contain custom notation, including window, report, table, and object classes. In practice you can use custom notation to override the behavior of the standard notation, or to add your own properties and methods to an object.

With the exception of the names in the above table, if the name of a custom property or method is the same as a standard one, such as “$printrecord()”, it will override the standard one.

You create custom properties and methods for an object in the method editor. You enter custom notation for a field in the Field Methods for the field, and for a class in the Class Methods for a class.

The code for a custom method must define the method parameters as you would for any other method, and then return the result of executing the method.

The code for a custom property typically comprises two methods. The first, called $propertyname returns the value of the property using the Quit method command. The second called $propertyname.$assign defines a single parameter, which is the new value to be assigned to the custom property; this method usually returns a Boolean value, which is true if the property was successfully assigned. Note that $canassign is always true for custom properties.

An instance of a class contains whatever custom notation you define in the class, together with the properties and methods for that type of instance. The object group $attributes contains all the built-in and custom notation for an instance. You can use $first() and $next() against $attributes, but $add() and $remove() are not available.

You can reference custom notation using the notation Notation.$xyz, where Notation is some notation for an instance of a class and “$xyz” is the name of your custom property or method. If you specify parameters, such as Notation.$xyz(p1,p2,p3), they are passed as parameters to the custom method, and a value may be returned.

You can use the *Do default* command within the code for custom notation , to execute the default behavior for a property or method with the same name as the custom notation. You can use the *Do redirect* command to redirect execution from custom notation in one instance to another instance containing custom notation with the same name.

To create a custom method

1. Open the Class or Field methods for your class
2. Right-click on the method names list, and select Insert New Method from the context menu
3. Enter a name for your custom method, including a dollar sign at the beginning of its name
4. Enter the code for the custom method as you would any other method

Userinfo property

File, window, report, menu, toolbar, schema, and query classes have the $userinfo property which you can use to store your own value. The Property Manager only allows you to assign to $userinfo if its current value is empty, null or has character or integer data type. The data stored in $userinfo can be of any other type but it must be assigned to the class at runtime using the $assign() method.

### Using Custom Methods

The following example uses a task class containing a custom method called $printprinter(). You can call this method from anywhere inside the task instance using

###### Do $ctask.$printprinter()

The $printprinter() method sets the print destination and calls another class method depending on whether the user is accessing SQL or Omnis data; it contains the following code

###### # $printprinter() custom method Begin reversible block   Send to printer   Set report name REPORT1 End reversible block If iIsSQL   Do method printSQLData Else   Begin reversible block     Set search name QUERY1   End reversible block   Do method printOmnisData End If

The next example uses a window that contains a pane field and a toolbar with three buttons. When the user clicks on a button, the appropriate pane is selected and various other changes are made. Each button has a $event() method that calls a custom method called $setpage() contained in the window class. Note that you can send parameters with the custom method call, as follows

###### # $event() method for second toolbar button Do $cwind.$setpage(2)

The $setpage() custom method contains a parameter variable called pPage, and has the following code

###### # $setpage() custom method Switch pPage   Case 1     Do $cwind.$objs.MainPane.$currentpage.$assign(1)     Do $cwind.$title.$assign('Queries')   Case 2     Do $cwind.$objs.MainPane.$currentpage.$assign(2)     Do $cwind.$toolbars.$add('tbModify1')

###### # installs another toolbar     Do $cwind.$title.$assign('Modifying')   Case 3     Do $cwind.$objs.MainPane.$currentpage.$assign(3)     Do $cwind.$menus.$add('MReports')     # installs a menu in the window menu bar     Do $cwind.$title.$assign('Reports')   Default     Quit method kFalse End Switch

The final example uses a window containing a subwindow, which in turn contains a tree list. The subwindow contains a custom method called $buildtree() that builds and expands the tree list. You can call the $buildtree() method from the parent window and send it parameters, using the notation

###### Do $cwind.$objs.SubWin.$buildtree(lv\_ClassList)

The $buildtree() method contains a parameter variable called pv\_SourceList of List type that receives the list passed to it, and a reference variable called TreeRef set to the tree list field, and contains the following code

###### # $buildtree() custom method Do TreeRef.$setnodelist(kRelationalList,0,pv\_SourceList) Do TreeRef.$expand()

## Object Classes

*Object classes* let you define your own structured data objects containing variables and methods. You can create an object variable based on an object class which contains all the variables and custom methods defined in the class. When you reference an object variable an instance of the object class is created containing its own set of values. You can store an object variable instance and its values on a server or Omnis database. The structure and data handling capabilities of the object instance is defined by the types of variables you add to the object class; similarly, the behavior of an object variable is defined by the methods you add to the object class.

Object classes have the general properties of a class and no other special properties. They can contain class and instance variables, and your own custom methods. You can make a subclass from an object class, which inherits the methods and variables from the superclass.

To create an object class

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

Alternatively, you may want to create an object class using one of the wizards. In this case, you should click on Class Wizard, then Object, and select one of the wizards.

When you modify an object class, Omnis opens the method editor for the class. This lets you add variables, and your own custom properties and methods to the class.

When you have set up your object class you can create any other type of Omnis variable based on your object class: an object variable has type Object and its subtype is the name of your object class. For example, you can create an instance variable of Object type that contains the class and instance variables defined in the object class. When you reference a variable based on an object class you create an instance of that object class. You can call the methods in the object class with the notation ObjVarName.$MethodName(), where $MethodName() is any custom method you define in the object class.

You can store object variable instances in an Omnis data file, or in a server database that stores binary values. When you store an instance of an object variable in a database, the value of all its contained instance variables are also stored. When the data is read back into memory the instance is rebuilt with the same instance variable values. In this respect you can store a complete record or row of data in an object variable.

You can store object instances in a list. Each line of the list will have its own instance of the object class. Object instances stored in a task, instance, local, or parameter variable belong to the same task as the instance containing that variable. Similarly object instances stored in a list or row belong to the same task as the instance containing the list or row variable. All other object instances have global scope and are instantiated by the default task and belong to the default task.

You cannot make an object instance a private instance. If you delete the object class an object variable uses, the object instance will be empty.

An object instance stored as a class variable in a task is destroyed as soon as its original task is destroyed.

To add variables and methods to an object class

1. Open your object class in design mode
2. Right-click in the variable pane of the method editor and select the Add New Variable option from the context menu
3. Name the variable, give it a type and subtype as appropriate
4. Right-click in the Method Names pane of the method editor and select the Add New Method option from the context menu
5. Name the method, including the dollar prefix

If you right click on an object variable, and use the Variable <name>... entry in the context menu, the variables list window opens, initially showing instance variable values.

#### Missing Object Variables

An error is generated when you open a library that tries to construct an Object variable and the Object class it is based on does not exist or is in a library that is not open. In this scenario, you will get a runtime error and execution blocks with the following error:

E100101: Class not found when constructing an object variable

The class name is TESTB.oTestB

Either the class does not exist or it has the wrong type e.g. remote object

In versions prior to Studio 10 in this case, you would have received the error “Class not found”, but code execution would have continued which may have led to further errors in the application.

#### Variable Count

The $usage property reports the current number of object variables that are sharing the underlying external component object. A NULL value means the object is neither an external component object, nor is it subclassed from an external component object.

Note that copies of the object such as $statementobject and $sessionobject contribute to the count.

Using Object Classes

This section describes an invoices example and uses an object class and simple invoices window; it is intended to show what you can do with object classes, not how to implement a fully functional invoices application. You can create the data structure required for an invoice by defining the appropriate variables and custom methods in an object class.

The following example uses an object class called o\_Invoice, which contains the variables you might need in an invoice, such as the invoice ID, quantity, value, and a description of the invoice item. The o\_Invoice object class also contains any custom methods you need to manipulate the invoice data, such as inserting or fetching invoices from your database. The methods in the object class contain the following code

###### # $SaveInvoice() method contains # local var lv\_InvoiceRow of type Row and # local var lv\_Bin of type Binary and # parameter var pv\_Object of type Field reference Do lv\_InvoiceRow.$definefromtable(t\_Invoices) Calculate lv\_bin as pv\_Object Calculate lv\_InvoiceRow.InvoiceObject as lv\_bin Do lv\_InvoiceRow.$insert() Returns #F # $SelectInvoice() method contains # local var lv\_Row of type Row Do lv\_Row.$definefromtable(t\_Invoices) Do lv\_Row.$select() Do lv\_Row.$fetch() Returns #S1 Quit method lv\_Row.InvoiceObject # $FetchInvoice() method contains # local var lv\_Row of type Row Do lv\_Row.$definefromtable(t\_Invoices) Do lv\_Row.$fetch() Quit method lv\_Row.Inv\_Object # $total() method Quit method iv\_QTY \* iv\_Value

The invoice window can contain any fields or components you want, but would contain certain fields that reference the instance variables in your object class, and buttons that call the methods also in your object class.

The invoice window contains no class methods of its own. All its functionality is defined in the object class. The window contains a single instance variable called iv\_Invoice that is based on the o\_Invoice object class.

The instance variable would have the type Object and its subtype is the name of your object class, o\_Invoice in this case. If the object class is contained in another library, the object class name is prefixed with the library name.

When you open the invoice window an instance of the object class is created and held in iv\_Invoice. Therefore you can access the instance variables and custom methods defined in the object class via the instance variable in the window; for example, iv\_Invoice.iv\_QTY accesses the quantity value, and iv\_Invoice.$SaveInvoice() calls the $SaveInvoice() method. Each field on the invoice window references a variable in the object class; for example, the dataname of the quantity field is iv\_Invoice.iv\_QTY, the dataname of the item or description field is iv\_Invoice.iv\_Item, and so on.

The buttons on the invoice window can call the methods defined in the object class, as follows.

###### # $event() method for the Select button

###### On evClick   Do iv\_Invoice.$SelectInvoice(iv\_Invoice.iv\_ID) Returns iv\_Invoice   Do $cwind.$redraw() # $event() method for the Fetch button

###### On evClick   Do iv\_Invoice.$FetchInvoice Returns iv\_Invoice   Do $cwind.$redraw() # $event() method for the Save button

###### On evClick   Do iv\_Invoice.$SaveInvoice(iv\_Invoice)

When you enter an invoice and click on the Save button, the $SaveInvoice() method in the object class is called and the current values in iv\_Invoice are passed as a parameter. The $SaveInvoice() method receives the object instance variable in the parameter pv\_Object and executes the following code

###### Do lv\_InvoiceRow.$definefromtable(t\_Invoices) Calculate lv\_bin as pv\_Object Calculate lv\_InvoiceRow.InvoiceObject as lv\_bin Do lv\_InvoiceRow.$insert() Returns #F

The row variable lv\_InvoiceRow is defined from the table class t\_Invoices which is linked to the schema class called s\_Invoices which contains the single column called InvoiceObject. The binary variable lv\_bin, which contains the values from your object instance variable, is assigned to the row variable. The standard $insert() method is executed which inserts the object variable into your database. The advantage of using an object variable is that all the values for your invoice are stored in one variable and they can be inserted into a binary column in your database via a single row variable. If you want to store object variables in an Omnis database you can create a file class that contains a single field, called InvoiceObject for example, that has Object type, rather than Binary, and use the appropriate methods to insert into an Omnis data file.

### Libraryname prefix for object variables

There is an "Include library prefix" check box on the object class selection dialog. This allows you to add the library name to the object class name for the variable, but when the object class is in the same library you don’t need to add the libraryname prefix.

### Dynamic Object Instances

The object class has a $new() method that lets you create an object instance dynamically and store it in an object variable, for example

###### Do $clib.$objects.objectclass.$new(parm1,parm2,...) Returns objectvar

where parameters parm1 and parm2 are the $construct() parameters for the object instance. When the instance is assigned, any existing instance in the object variable is destroyed. It would be normal practice to put no class in the variable pane for object variables which are to be set up dynamically using $new(), but there is no class checking for instance variables so no error occurs if the class shown in the variable pane is different from the class of the new instance.

You can do a similar thing with an external function library if it contains instantiable objects, such as Fileops. For example

###### Do Fileops.$objects.fileops.$new() Returns objectvar Do objectvar.$openfile(pFilename)

The following example uses an Object variable ivSessionObj which has no subtype defined in the method editor. When this code executes ivSessionObj is instantiated based on the object class SessionObj which was created using the Session Wizard in Omnis. Once the object instance exists the $logon() method is called.

###### Do $clib.$objects.SessionObj.$new() Returns ivSessionObj # runs $construct() in the SessionObj object class Do ivSessionObj.$logon() # runs the $logon() method in SessionObj

### Self-contained Object Instances

Object classes have the $selfcontained property. If set to kTrue the class definition is stored in all instances of the object class. An instance of such an object class is disconnected from its class and instead relies on its own class data for method and instance variable definitions. When the object instance is stored on disk the class definition is stored with the instance data and is used to set up a temporary class whenever the instance is read back into memory. Any changes to the original object class have no effect on existing instances, whether on disk or in memory.

Once an instance is self-contained it is always self-contained, but you can change its class definition by assigning to $class, for example

###### Do file1.objvar1.$class.$assign($clib.$objects.objectclass1)

causes the class definition stored inside objvar1 to be replaced by the current method and variable definitions for objectclass1. The instance variable values are maintained provided the new instance variable definitions are compatible with the old ones (Omnis can handle small changes in the type of variables but won’t carry out substantial conversions). Note that the old instance variables are matched to the new ones by $ident and not by name, so to avoid problems the new class should be a descendant of the original class with none of the original variables having been deleted.

Only the main class is stored with the object instance, inheritance is allowed but any superclasses must exist in open libraries whenever the instance is present in memory. Assigning to $class does not change the superclass structure of self-contained instances.

## Object References

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

You can use the Object reference data type with local, instance, class and task variables. Object references have no subtype. To create a new Object instance, referenced by an Object reference variable, you use the methods $newref() and $newstatementref(). These are analogous to the $new() and $newstatement() methods used to create object variables, and they can be used wherever $new() and $newstatement() can be used. When created, an object reference will belong to the current task, and are no longer valid after the task closes.

Once you have associated an Object instance with an Object reference variable, you can use it to call methods just like you would with an Object variable.

The instance associated with an object reference variable will be destroyed in the following circumstances:

1. When they are no longer required, object references are deleted automatically in order to free up memory (when a variable or list column no longer contains the reference, for example).
2. You can use the $deleteref() method to delete the object reference, if you want to release memory sooner than would otherwise occur under the automatic process.
3. The task that created the variable is closed.
4. The library is closed.

Until one of these occurs, Object reference variables can be passed as parameters, copied, stored in a list column, and so on. All copies of the variable address the same single instance of the object. As soon as the object is deleted, other references to the object become invalid. Prior to deleting the object, Omnis calls the destructor method.

Note that this approach means that you must delete objects when you have finished with them, otherwise significant memory and resource leaks can occur.

There is also a method $listrefs() which you can call to list the instances of an object class, external object, or in fact *all* instances created in this way. This is useful both for leak-checking, and for clean-up.

You can use $newref() in conjunction with session pools. The only differences are that $deleteref() returns the object instance to the pool, rather than destroying it, and $listrefs() does not include objects from a session pool.

There are two further methods that can be used with object reference variables. $copyref() creates an object instance which is a copy of the instance referenced by the variable, and $validref() returns a Boolean which is true if and only if the variable references a valid object instance. Note that if $copyref() takes a copy of a DAM session object, both copies reference the same DAM session.

The method $objref() can also be used. If there is an object reference associated with the instance, the method returns the object reference, otherwise returns #NULL, that is, the object MUST have been created with a $newref().

### Constructing new objects

…from an object class

###### Calculate myObjectReference as $clib.$objects.myObject.$newref(“Test”)

…from an external object

###### Calculate myObjectReference as $extobjects.ODBCDAM.$objects.ODBCSESS.$newref()

…from a session object

###### Calculate myObjectReference as mySessionObjectReference.$newstatementref()

…storing the reference in a list column of type Object reference

###### Do theList.$add($extobjects.ODBCDAM.$objects.ODBCSESS.$newref())

### Listing instances created using an Object reference

…instances of an object class

###### Calculate myList as $clib.$objects.myObject.$listrefs()

…instances of an external object

###### Calculate myList as $extobjects.ODBCDAM.$objects.ODBCSESS.$listrefs()

…all instances

###### Calculate myList as $listrefs()

### Destructing objects

…using an object reference variable

###### Do myObjectReference.$deleteref() ## Calls the destructor of the object instance

…all instances referenced by a list column of type Object reference

###### Do myList.$sendall($ref.1.$deleteref())

…using the $listrefs() method

###### Do $extobjects.ODBCDAM.$objects.ODBCSESS.$listrefs().$sendall($ref.1.$deleteref()) Do $libs.DAMTEST.$objects.Doracle8.$listrefs().$sendall($ref.1.$deleteref())

### Testing an object reference for validity

…using an object reference variable

###### If myObjectReference.$validref()

…using a list column of type Object reference

###### If myList.1.myObjRef.$validref()

### Copying an object reference

…using an object reference variable…

###### Calculate objectReference as myObjectReference.$copyref()

…using a list column…

###### Calculate objectReference as myList.1.myObjRef.$copyref()

## External Objects

*External objects* are a type of external component that contain *methods* that you can use by instantiating an object variable based on the external object. External objects can also contain *static functions* that you can call without the need to instantiate the object. These functions are listed in the Catalog under the Functions pane.

Some external objects are supplied with Omnis Studio; these include equivalents to the FileOps and FontOps externals, a Timer object, as well as the new multi-threaded DAMs. Writing your own external objects is very similar to writing external components, which is described in a separate manual available to download from the Omnis website. The FileOps and FontOps functions are documented in the Omnis Help.

External objects are created and stored in component libraries in a similar manner to external components, and in future releases are intended to replace external functions and commands, although to maintain backward compatibility, the old external interface is still supported at present.

External object libraries are placed in the XCOMP folder, along with the visual external components. They must be loaded in the same way as external components using the External Components option, available in the Browser when your library is selected.

### Using External Objects

You can add a new object in the method editor by inserting a variable of type Object and using the subtype column to select the appropriate external object. You can click on the subtype droplist and select an external object from the Select Object dialog. This dialog also appears when you create an object elsewhere in Omnis, such as the file class editor. An icon in the variable subtype cell shows whether the variable is based on an object class or an external object.

When an instance of the external object has been constructed, you can inspect its properties and methods using the Interface manager.

To use the object’s methods in your code, you can drag the method you require from the Interface Manager into the command parameters box.

For some objects it is important to note that for the Interface manager to interrogate an object it will need to be constructed. For example if the Interface Manager was used on an Automation object, the Automation server needs to be started.

External objects are contained in the notation group $extobjects, which you can omit from notation.

### External Object Events

External objects do not support events in the GUI sense. They can however define notification methods which they call when certain events occur.

You can subclass an external object and then override the notification method so your code is informed of the event. The Timer object supplied in Omnis is an example of this. To subclass an object, you can either set the superclass property in the Property Manager, or use the New Subclass Object wizard available in the Browser (using the Class Wizard>>Object option).

### External Object Notation

All the components available to Omnis are listed in the $root.$components group. Individual components have their own methods, in addition to the $cmd() method which allows you to send a component specific command to the component or object.

The following examples show some commands that you can send to an ActiveX component to manage timeouts:

###### Do $components.ActiveX.$cmd("RequestPendingTimeout",10000) # Set timeout in milliseconds Do $components.ActiveX.$cmd("ServerBusyTimeout",10000) # Set timeout in milliseconds Do $components.ActiveX.$cmd("ResponseDialog",kFalse) # Prevent response dialog from appearing Do $components.ActiveX.$cmd("BusyDialog",kFalse) # Prevent busy dialog from appearing

## Interface Manager

The Interface Manager displays the public methods and properties for objects in Omnis Studio, that is, any class that can contain methods and can be instantiated, including window, menu, toolbar, report, task, table, and object classes (not available for code classes). Furthermore, for window, report, toolbar, and menu classes the Interface Manager displays the methods for the objects in the class. For each class or object, the Interface Manager displays all built-in methods, including those available in the instance of the class, as well as any custom methods you have added to the object.

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

For each method in the class or object, the Interface Manager displays the method name, its parameters, return types, and a description, if any are present.

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

To view the Interface Manager

1. Right-click on the class in the Browser and select Interface Manager from the context menu

or from the method editor

1. Open the method editor for the class
2. Select View>>Interface Manager from the method editor menubar

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

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

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

### Dragging methods from the Interface Manager

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