ESI Technology Corporation

Meeting the healthcare challenge in a world of complexity!

Tools Reference Guide

EsiObjects V4.1

ESI Technology Corporation
5 Commonwealth Road
Natick, MA. 01760
www.esitechnology.com

Table of Contents

Introduction

This guide covers all of the EsiObjects development tools needed to develop an application. These tools are an integral part of the Class Development Environment (CDE).

Document Conventions

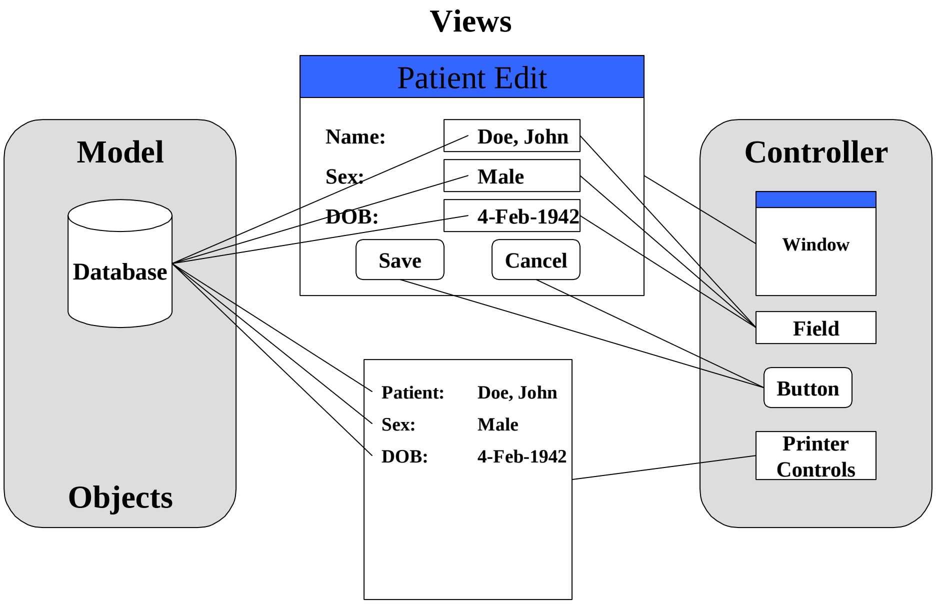
EsiObjects documentation uses the following typographical conventions:

For more information on this subject please refer to the BREAK Command section of this manual.	Underlined text is used to highlight a reference to another section of this manual or another manual.
Property	In text, italicized words indicate defined terms that are usually used for the first time. Words are also italicized for emphasis.
CREATE	Words in bold and capitalized are EsiObjects commands or keywords.
`Set T%Test=I%Pat.Name`	This font is used for code examples.

Overview of EsiObjects

Model-View-Controller

The Model-View-Controller concept is a common design pattern used to implement modern applications. EsiObjects itself is implemented using this pattern. As an EsiObjects programmer, you should strive to implement you applications using this pattern.



The diagram above illustrates the pattern when applied to a client server database application. Simply put, this concept enforces a separation of data (the model), from the interface primitives (the controller). These two sides are brought together into views. These views reflect different ways of looking at the same data.

In the example above, displaying the Name, Sex and DOB of a patient creates a Window view. The window components are a part of the controller. The patient data is extracted from the database and displayed using the controller components. Another view is created on paper of the same model side data. In this case the Printer Controls are used to create the view.

The EsiObjects development environment contains tools for creating and maintaining all class (model side) structures and code including workflow tools that increase your productivity.

EsiObjects Client Environment Overview

For a top-level overview of EsiObjects, see the EsiObjects Overview section of the EsiObjects Programmer Reference Guide.

The **EsiObjects Class Development Environment** (CDE) contains all the tools needed to develop the definitional components of an object oriented database application. These tools can be grouped functionally into the following categories:

- Workflow tools that increase your productivity.
- Class development tools that create all the definitional components of an object.
- Tools needed to test and debug your application.
- Import and export tools needed to transport the application components to external systems for sharing or backup.

In terms of the Model-View-Controller paradigm described in the previous section, the CDE provides the tools needed to develop the model side of an application.

The client side of the Class Development Environment is based on Microsoft Windows™. Fundamental to the EsiObjects environment is the Main Window. It contains a menu that lets you access the components listed below. It also gives you access to printing and help services. The Main Window contains several child windows that support the CDE set of tools or any other tool set that may be developed in the future. These windows are:

- **Session Browser –** This window contains tab sheets that contain graphical representations of library and folder structures associated with a particular session connection. Library and folder structures are visually displayed and provide a point and click approach to migrating to the desired services. Additionally, tools exist to automatically go any class in the hierarchy if you know its name or, if you are working in a folder, you can go directly to an object whose pointer is stored in the folder.
- **Output –** This window contains three tab sheets. The *Build* sheet is used by the system to display information about compiles, syntax checks, etc. The *Debug* sheet is used by the system to display system information. The *Output* sheet is available to you for displaying application level information. It provides a convenient way to track execution when testing your application among other things.
- **Documentation –** This window is used by the system to display documentation for the currently selected object. The text is stored in Rich Text Format to be compatible with other Microsoft tools such as Word™.
- **Session Control –** This window allows you to establish connections to one or more EsiObjects servers.
- **User Options –** This window contains three tab sheets that allow you customize your development environment.

EsiObjects Server Environment

When the EsiObjects client is started up on a PC and a session connection is made to a process that is started up on the server by the TCP redirector, an environment object is created in the server process. It is an instance of the class ESI$WindowsEnvironment. The client owns and is connected to the environment object on the server. The environment is always active as long as a session connection is maintained and can be referenced via the **$ENVIRONMENT** special variable.

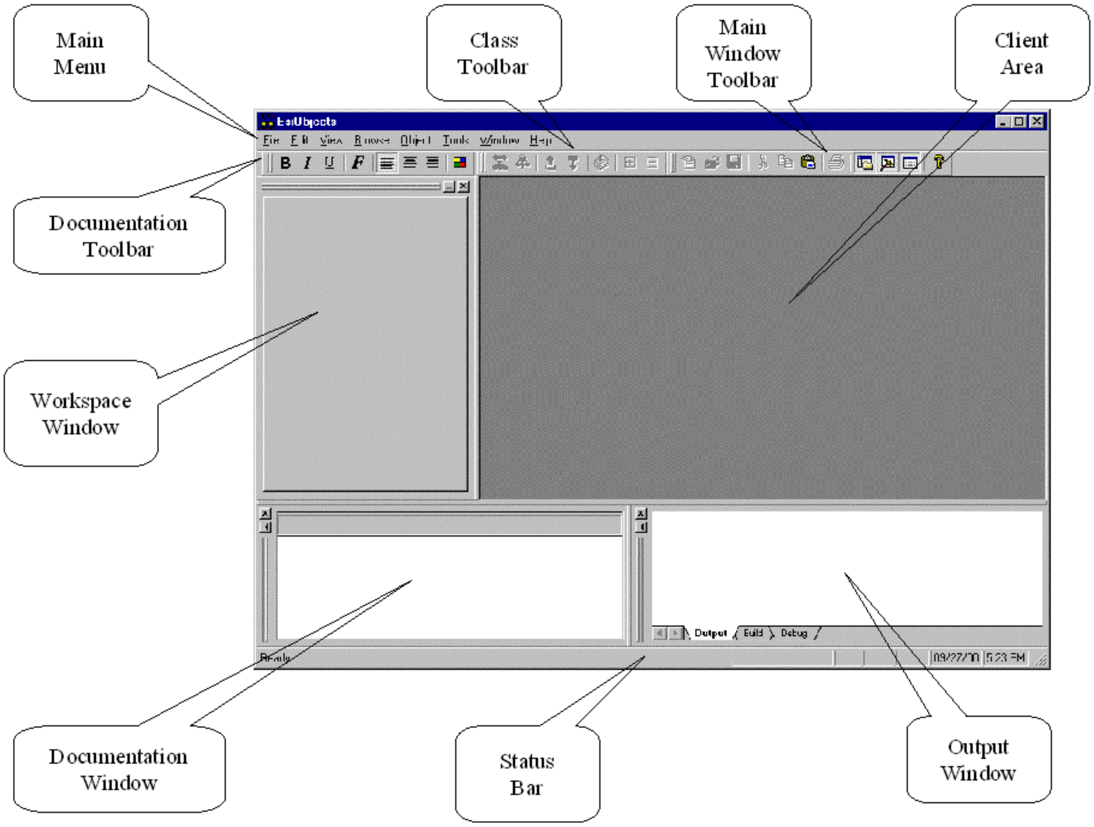
The environment object is always instantiated as part of the EsiObjects session connection. There is only one environment object associated with a session. EsiObjects supports multiple development sessions. That is, a programmer can have multiple sessions defined and have an EsiObjects environment associated with each session. This provides a flexible approach to development since browsers and editors can be brought up simultaneously while attached to separate sessions.

Class Development Environment Overview

Main Window and Components

Main Window Explained

This is the main window of the EsiObjects environment. It appears whenever the system is launched, and serves as the primary EsiObjects work area.



The **Main Menu** contains cascading menus of all the functions available in EsiObjects. The menus will always look the same to the programmer. Only those commands that are applicable to the currently selected item will be highlighted and activated. All others will be grayed out. The Main Menu contains all commands that are applicable to a selected object where popup menus generally contain only the most frequently used commands.

The window's **Client Area** is where EsiObjects places all visual objects launched from a browser or menu such as the method editor, property editor, etc.

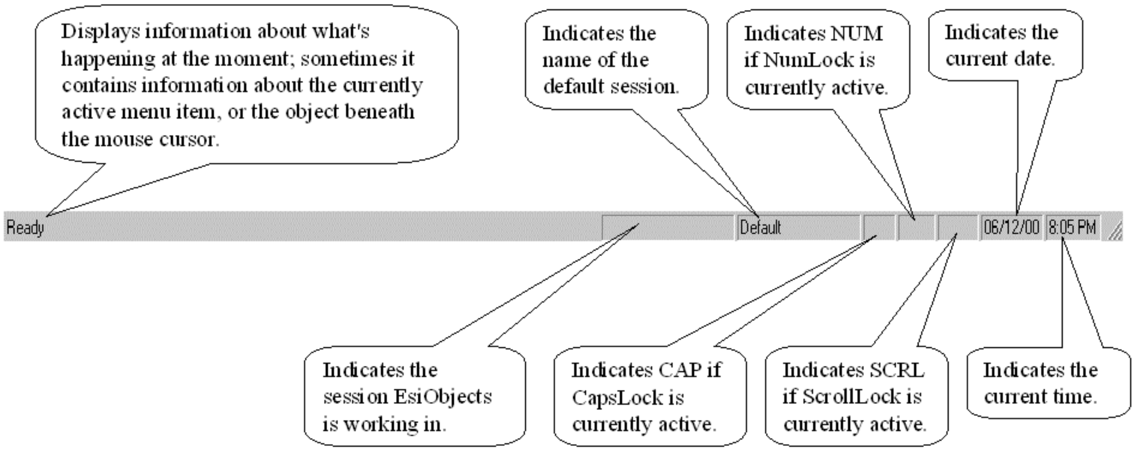
The **Session Browser** contains tab sheets that display library and folder structures for the currently connected sessions. The Session Browser and all of its functionality are used to migrate through all the available library and folder structures.

The **Main Window Toolbar** provides quick access to a variety of important functions that also appear on the Main Menu.

The **Documentation Window** always contains the documentation for the item selected in the Client Area.

The **Output Window** contains the Output, Debug and Build tab sheets. The Output sheet is available to you as an area to output information via the $ENV.Ouptut message. The Debug area is used by EsiObjects to display errors and the Build sheet is used by EsiObjects to display compile and syntax checking information.

The **Status Bar** on the bottom of the main window provides information on the current status of EsiObjects. It can be displayed toggling the main window's **View|Status Bar** command.



Displays information about what's happening at the moment; sometimes it contains information about the currently active menu item, or the object beneath the mouse cursor.

Indicates the name of the default session.

Indicates NUM if NumLock is currently active.

Indicates the current date.

Indicates the session EsiObjects is working in.

Indicates CAP if CapsLock is currently active.

Indicates SCRL if ScrollLock is currently active.

Indicates the current time.

Using the Main Window

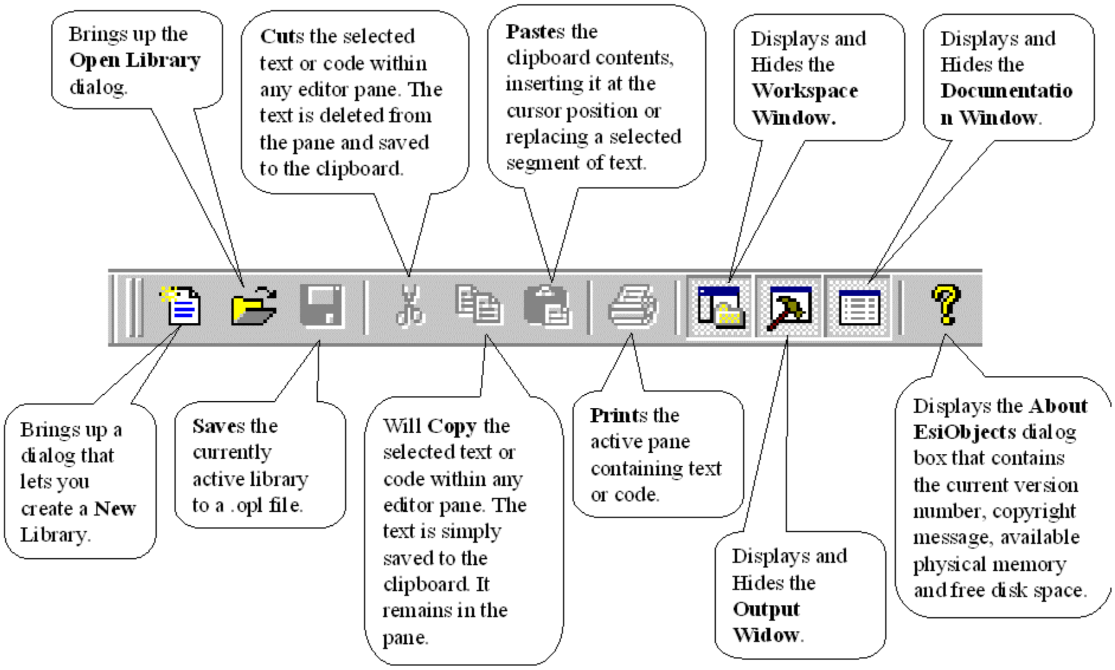
Keyboard Shortcuts

Certain Main Window operations can be accessed via accelerator keys. They are:

- **Alt+F4** shuts down EsiObjects, and closes the main window.
- **Ctrl+F4** closes any currently active child window.

Main Window Toolbar

The **Main Window Toolbar** provides quick access to a variety of important functions. Most of these functions can be performed in other ways; the toolbar is one of the most *convenient* ways to get at them. It can be shown or hidden from the Main Window's **View** menu, by selecting the **Toolbar** item.


Brings up the **Open Library** dialog.
Cuts the selected text or code within any editor pane. The text is deleted from the pane and saved to the clipboard.
Pastes the clipboard contents, inserting it at the cursor position or replacing a selected segment of text.
Displays and Hides the **Workspace Window.**
Displays and Hides the **Documentation Window**.
Brings up a dialog that lets you create a **New** Library.
Saves the currently active library to a .opl file.
Will **Copy** the selected text or code within any editor pane. The text is simply saved to the clipboard. It remains in the pane.
Prints the active pane containing text or code.
Displays and Hides the **Output Widow**.
Displays the **About EsiObjects** dialog box that contains the current version number, copyright message, available physical memory and free disk space.

To see the names of each button, position the mouse pointer over the button and wait about one second. A tool tip will appear with the name of the button.

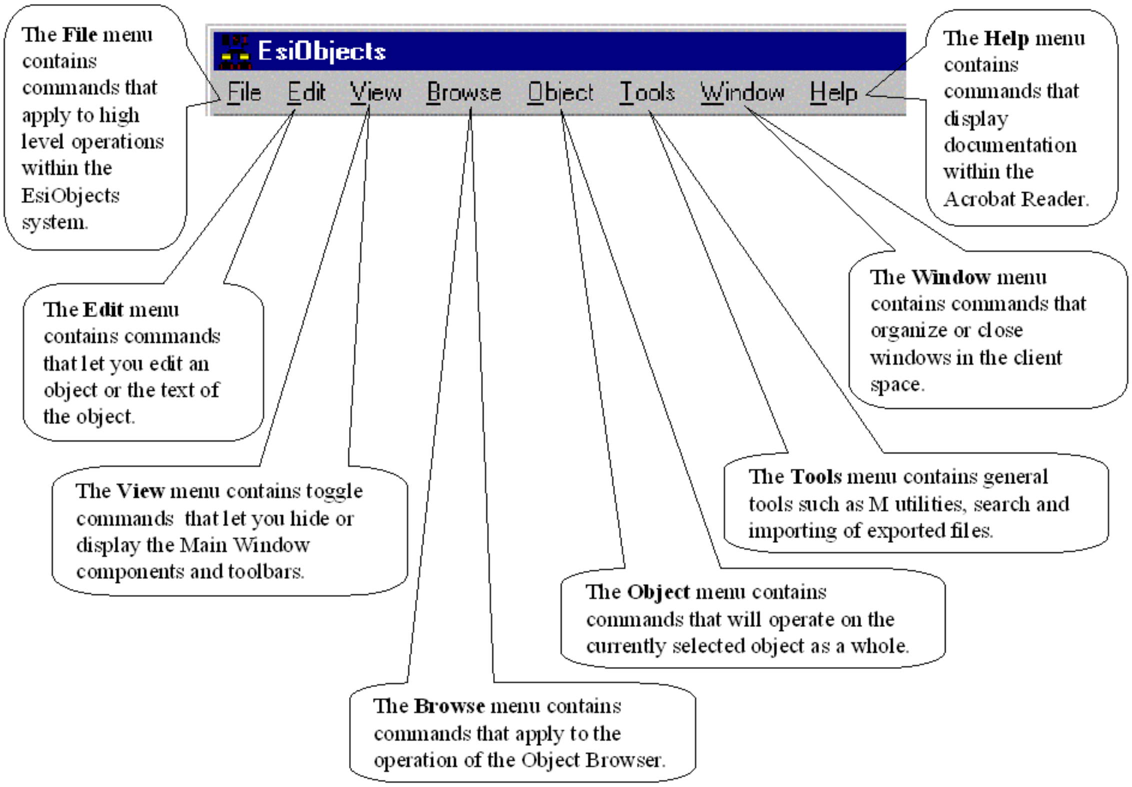
The toolbar is **detachable**, and can be *torn off* simply by **dragging** its margin area (without pressing one of the buttons). It can then be *docked* back to one of the four edges of the main window by dragging it over to the desired edge. Note that its orientation is horizontal when it is docked to the top or bottom of the window, and vertical when it is at the sides. While dragging, you can toggle its *orientation* by holding down the **Shift** key. You can also *prevent* it from docking at the edges of the main window by holding down the **Ctrl** key when moving it. When it is free-standing, it is possible to move the toolbar *outside* the EsiObjects main window.

Main Window Menu

Main Menu Commands

The Main Menu of EsiObjects is designed to display all functions available to EsiObjects no matter what object is currently selected. In other words, the Main Menu will always look the same and have the same commands on them. Only those commands that are active for the selected object will be highlighted. All other commands will be grayed. Good object oriented systems present a common view to the user, hiding the complexity from them.

The following diagram illustrates and explains the general orientation of each main menu item.



The **File** menu contains commands that apply to high level operations within the EsiObjects system.

The **Help** menu contains commands that display documentation within the Acrobat Reader.

The **Edit** menu contains commands that let you edit an object or the text of the object.

The **Window** menu contains commands that organize or close windows in the client space.

The **View** menu contains toggle commands that let you hide or display the Main Window components and toolbars.

The **Tools** menu contains general tools such as M utilities, search and importing of exported files.

The **Object** menu contains commands that will operate on the currently selected object as a whole.

The **Browse** menu contains commands that apply to the operation of the Object Browser.

All commands on the Main Menu will be explained in general at this point. The descriptions will apply to the object selected. Within this guide, each section that describes functional areas of EsiObjects will contain a list of active commands.

Main File Menu

Menu	Command	Description
File	**New|Session**	Invokes the New Session form that prompts for the name of the session and the type of connection: TCP or COM. A new session will be created if you choose to continue.
	New|Routine	Creates a new M routine.
	New|Library	Invokes the Create Library form that prompts for the required information needed to create a new library. If you proceed, the library will be created and its icon will be displayed in the **Session Browser**.
	New|Class	Invokes the Create Class form that prompts for the required information needed to create a new class within the selected library. If you proceed, the class will be created below the selected library or class icon.
	New|Interface	Creates a new interface within the selected class named Interface *n* where *n* is a sequential number.
	New|Method	Invokes the Add to Interface form that prompts for the service name and lets you select the type of service: Method, Property, Relationship or Event.
	New|Property	Invokes the Add to Interface form that prompts for the service name and lets you select the type of service: Method, Property, Relationship or Event.
	New|Relationship	Invokes the Add to Interface form that prompts for the service name and lets you select the type of service: Method, Property, Relationship or Event.
	New|Event	Invokes the Add to Interface form that prompts for the service name and lets you select the type of service: Method, Property, Relationship or Event.
	New|Instance Variable	Invokes the Add Variable form that prompts for the variable name and lets you select the type of variable: Instance or Class.
	New|Class Variable	Invokes the Add Variable form that prompts for the variable name and lets you select the type of variable: Instance or Class
	New|Folder	Adds a new folder to the session. If a folder is selected, the new folder will be made a child of the selected folder. If any part of a library structure is selected, it will create a root folder. Folders can be dragged to other folders.

	New|Version	When a code body has been selected (or any object that supports versioning), executing this command will create and save a new version of it.
	New|Object…	Not Implemented Yet.
	Save	Saves the selected object to persistent storage.
	Revert	When several edits have been made to the text or code of the object, executing this command will restore the object to its original state unless you have saved it in the interim.
	Rename	Lets you rename a object. The selected object is put into edit mode and all the text is selected. Any edit operation at this point will delete the selected name, replacing it with the typed or pasted characters. If you want to simply modify the existing name, move your cursor to the position and perform the edit. Clicking at the point of edit with your mouse will also put you in insert mode.
	Delete	Deletes the selected object or the selected text if in edit mode. When deleting an object you will usually be prompted to continue or not.
	Print...	Invokes the print dialog and then prints the selected object or text to the selected printer if you choose to proceed.
	Print Setup...	Invokes the Printer Setup form and lets you change the printer setup parameters. If you choose to proceed, the printer and printing options will be changed.
	Connections|Show	Displays the Session Control Window. Sessions can be created, deleted, modified, connected and disconnected from this window.
	Connections|Connect	Connects the selected session in the Session Control Window.
	Connections|Disconnect	Disconnects the selected session in the Session Control Window.
	Exit	Prompts to continue and then shuts the EsiObjects system down if you answer in the affirmative.

Main Edit Menu

<u>**Menu**</u>	<u>**Command**</u>	<u>**Description**</u>
Edit		
	Undo	Executing this command will undo the last operations performed against a selected object.
	Redo	Executing this command will redo the last undo operation.

	Cut	This command will remove the selected item or text and place it on the clipboard for future use.
	Copy	This command will copy (not remove) the selected item or text and place it on the clipboard for future use.
	Paste	Executing this command will cause the contents of the clipboard to be inserted at the cursor position if in text editing mode or replace a selected region.
	Delete	Executing this command will delete the selected text or object. Generally, when the delete operation is performed on an object, you will be queried as to whether you want to proceed.
	Select All	This command will select all objects or text.
	Find	Invokes the Find dialog box to appear. After specifying search criteria, it will search for that criteria and stop for you to perform edit or replace operations.
	Find Next	This command will continue the Find operation, continuing the search for the criteria established in the original search.
	Replace	This command operates in conjunction with the Find and Find Next operations. Executing it will replace the found instance with a specified instance.

Main View Menu

Menu	**Command**	**Description**
View		
	Toolbars|Main	This command is a toggle command that toggles the Main EsiObjects toolbar off and on. When the toolbar is toggled on, a √ will appear before command. When toggled off, the toolbar will not be displayed. When toggles on, it will be displayed either docked or not.
	Toolbars|Class	This command is a toggle command that toggles the Class toolbar off and on. When the toolbar is toggled on, a √ will appear before command. When toggled off, the toolbar will not be displayed. When toggles on, it will be displayed either docked or not.
	Toolbars|Browser Actions	When the Object Browser is active in the Client Area, this command will toggle the Browser Actions toolbar in the browser off and on.
	Toolbars|Symbol Types	When the Object Browser is active in the Client Area, this command will toggle the Symbol Types toolbar in the browser off and on.

Toolbars|Documentation This command is a toggle command that toggles the Documentation toolbar off and on. When the toolbar is toggled on, a √ will appear before command. When toggled off, the toolbar will not be displayed. When toggles on, it will be displayed either docked or not.

Find in Tree This command is used to find a service that is actively being edited within the library or folder tree structures. It is used to synchronize the Session Browser tree selection with the current service being edited.

Debugger This command will activate the Debugger window. The Debugger window is a separate window that is used to display the code, stack and symbols of the object being debugged. It has its own menu and is independent of the Main Window.

Documentation This command is a toggle command that toggles the Documentation Window between a hide and display state. When the command is toggled on, a √ will appear before command. When toggled off, the Documentation Window will not be displayed. When toggled on, it will be displayed as docked.

Output This command is a toggle command that toggles the Output Window between a hide and display state. When the command is toggled on, a √ will appear before command. When toggled off, the Output Window will not be displayed. When toggled on, it will be displayed as docked.

Session Browser This command is a toggle command that toggles the Session Browser between a hide and display state. When the command is toggled on, a √ will appear before command. When toggled off, the Session Browser will not be displayed. When toggled on, it will be displayed as docked.

Status Bar This command is a toggle command that toggles the Main Window Status Bar between a hide and display state. When the command is toggled on, a √ will appear before command. When toggled off, the Main Window Status Bar will not be displayed. When toggled on, it will be displayed.

Main Browse Menu

Menu	Command	Description
Browse		
	Look Into	Within the context of the Object Browser, if a variable is selected that has a OID associated with it, executing this command will force the Object Browser to migrate that link and display the context of the object pointed to by the subscript OID.
	Look In Subscript	Within the context of the Object Browser, if a variable is selected that has a OID as a subscript, executing this command will force the Object Browser to migrate that link and display the context of the object pointed to by the subscript OID.
	Pull Out	Executing this command will force the Object Browser to return to the object it came from and redisplay its context.
	Watch	Not Implemented Yet.
	Show Descendants	Not Implemented Yet.
	Refresh	This command, when executed, will totally refresh the Object Browsers display of an object state (variables and values).
	Show History	The Object Browser keeps track of the objects it migrates through. Executing this command will force a List History list box to appear, displaying the migration history.
	Edit Value	When you have selected a variable within the Object Browser that has a string value bound to it, executing this command put you into edit mode. The value of the variable can then be modified.
	Goto Definition	Not Implemented Yet
	Class	Not Implemented Yet
	Evaluation	Not Implemented Yet
	Recycle	You have control over whether a completely new Object Browser is instantiated every time you migrate to a new object. The Recycle button on the browser's toolbar controls this. If the button is depressed, that means that only one instance of the browser will exist for all migrations. This command will indicate that by a √ in front of it. Executing this command toggles the Recycle button between the recycle and no recycle states.

	Auto Refresh	If you are changing the state of an object using the Object Browsers embedded Xecute Shell, you can use this toggle command to turn auto-refresh on and off. When on (indicated by a √ in front of the command), changing the state of the object being browsed will automatically cause the display to refresh. Toggling the Auto Refresh command causes the equivalent Auto Refresh button on the Object Browsers toolbar to pop in and out.

Main Object Menu

Menu	**Command**	**Description**
Object		
	Properties…	Invokes the property sheet for the currently selected object. The property sheet will display all the public properties of the object and their values.
	Edit	Invokes the editor for the selected object if it has one. For example, if you have a variable selected in the Session Browser, the Variable Editor will be brought up in the client area.
	Import	Invokes a common file dialog that lets you select a file that is the same type as the object selected in the Session Browser. This command is used to directly import the exported object into the selected object; therefore, the objects must be the same type.
	Export	This command will export the selected object to an external file on the EsiObjects workstation client. The common file dialog will let you select the file and directory to save the file in. It will also let you store all object relationships if applicable. The contents of the selected object will be stored in ASCII format that is readable.
	Override	When an object is selected in a subclass such as an interface, service or variable and the object is inherited from a superclass, this command will copy the actual object into the currently selected object. It is through this mechanism that you can specialize the behavior of an object.
	Promote	When an object is selected in a subclass such as an interface, service or variable and the object exists at the selected level and not at the superclass level, executing this command will copy the selected object to the superclass. It is through this mechanism that you can generalize an interface, service or variable, making it available to all subclasses of the superclass.

Goto Ancestor
When you have a service selected in a subclass and it exists there as well as in a superclass, executing this command will prompt EsiObjects to transfer control to that superclass service.

Compile|Release
This command will compile the selected item or all items for a release execution. The release compile is what normally runs when the code is executed. The compilation results are displayed in the Build tab sheet of the Output Window.

Compile|Debug
This command will compile the selected item or all items for debug execution. This is the code that the Debugger uses in a debugging session. The compilation results are displayed in the Build tab sheet of the Output Window.

Compile|Both
This command will compile the selected item or all items for release and debug execution. The compilation results are displayed in the Build tab sheet of the Output Window.

Compile|Advanced
This command is only active when a class is selected within the Session Browser. A dialog will display that lets you include all subclasses and/or nested classes in the compile range as well as the type of compile (Release or Debug).

Compile|Syntax Check
This command will syntax check the selected method or property accessor. The results of the check are put out to the Output Window.

Purge
Executing this command will cause a dialog box to appear requesting the number of code body versions you want to keep. After specifying the number to retain, the EsiObjects structure from the selected point down will be iterated and all lower versions exceeding this number will be deleted, leaving the highest numbered versions.

Remove Debug
This command causes the EsiObjects structure to be iterated from the selected point down. All debug compiles associated with encountered code bodies will be deleted.

Unlink
This command applies to classes. When a class is selected and it has superclasses (single or multiple inheritance), the selected class will be unlinked from its superclass. If the selected class multiply inherits from two or more superclasses, a dialog will appear, letting you make a choice as to which class to unlink from.

Link
This command applies to classes. When executed it causes the Link to Superclasses dialog to appear. It queries for a class to be linked to. A full Library$Class reference should be given.

	Xecute Shell	This command invokes the Xecute Shell in the context of the selected object. That is, if the Xecute Shell is invoked in the context of a selected method, you will have access the that objects internals (state).

Main Tools Menu

All the commands listed in *italics* are Add-in commands. A programmer who understands how to create an Add-in .dll file can add commands. See the next section for a more detailed explanation of the Add-in concept.

Menu	**Command**	**Description**
Tools		
	Global|Directory	*Invokes the Global selector letting you select a range of globals for display. Only the global name is displayed.*
	Global|Save	*Invokes the Global selector letting you select a range of globals for export to an external file.*
	Global|Restore	*Invokes the Global file selector letting you import a set of globals from an external file.*
	Routine|Directory	*Invokes the Routine selector letting you select a range of routines for display. Only the routine name is displayed.*
	Routine|Save	*Invokes the Routine selector letting you select a range of routines for export to an external file.*
	Routine|Restore	*Invokes the Routine file selector letting you import a set of routines from an external file.*
	Routine|Selective Restore	*Invokes the Routine file selector letting you import a selected set of routines from an external file.*
	Routine|Editor	*Invokes the Routine selector letting you select a routine to edit.*
	Search|All	Invokes the Search dialog that queries for search criteria. Once you provide the information, the search engine will be invoked. It will search all levels within the EsiObjects library structure for the criteria specified. When it gets a hit, it stores the reference to the object in a workbox. You can tear the workbox off and use this information to perform operations on.
	Search|Selected	Invokes the Search dialog that queries for search criteria. Once you provide the information, the search engine will be invoked. It will search the selected structure object and all lower levels for the criteria specified. When it gets a hit, it stores the reference to the object in a workbox. You can tear the workbox off and use this information to perform operations on.

	Goto Class	Invokes a dialog that queries for the name (or partial name using wildcard characters * and ?) of a class. If it finds multiple classes matching the search criteria, they are displayed in a list for you to select from. Double clicking on the desired class reference will prompt the system to transfer control to that class in the Session Browser. The class will be opened up, exposing the supported interfaces, nested classes and subclasses if they exist.
	Generic Import	Invokes a common file selection dialog that lets you select an export file that contains a supported files extension. The generic import will automatically import the contents of the file to the correct object. For example, if a method was exported into a file having a .opm file extension, the Generic import would import the contents of that file into the correct method.
	Options…	Invokes the EsiObjects Options dialog. This dialog contains User, Format and Preferences tab sheets. The User tab sheet lets you enter your name and initials to be used to identify code bodies. Additionally, macrocode can be entered that will expand when a new code or documentation body is created.

Main Windows Menu

Menu	Command	Description
Windows		
	Close	When invoked the selected window in the client area of the main window will be closed.
	Close All	When invoked, all windows in the client area will be closed.
	Next	Selects the next window in the client area.
	Previous	Selects the previous window in the client area.
	Cascade	Arranges all windows in the client area into cascading order.
	Tile Horizontally	Arranges all windows in the client area horizontally.
	Tile Vertically	Arranges all windows in the client area vertically.

Main Help Menu

Menu	Command	Description
Help		

Getting Started	Activates the Acrobat Reader and displays the Getting Started Tutorial. This tutorial is designed to teach you some fundamental object oriented concepts. It is primarily designed to teach you how to use the EsiObjects tool set.
Administrator's Guide	Activates the Acrobat Reader and displays the Administrator's Guide. This guide contains all the information needed to start and shutdown the EsiObjects system as well as how to install and set up the servers for the supported M systems.
Language Reference Guide	Activates the Acrobat Reader and displays the Language Reference Guide. This guide contains all the information you will need to use the EsiObjects language. Each language element is explained in detail.
Programmer's Reference Guide	Activates the Acrobat Reader and displays the Programmer's Reference Guide. This guide contains all the information you will need to know about objects and how to use them within your application.
Tools Guide	Activates the Acrobat Reader and displays the Tools Guide. This guide contains extensive information about the EsiObjects tool set. Each GUI object is described in detail along with instructions on how to use it.
About EsiObjects	Invokes a dialog that displays current status information about the EsiObjects Class Development Environment.

Main Menu Add-in Programs

Add-ins are supplemental programs that extend the capabilities of EsiObjects by adding custom commands and specialized features.

You can write your own Add-in programs. Writing your own is outside the scope of this guide.

To use an add-in, you must install the add-in program in the EsiObjects root directory and then register it. If the program has been implemented properly, it will appear as a command in the Tools menu.

To take it off the menu, all you have to do is unregister it.

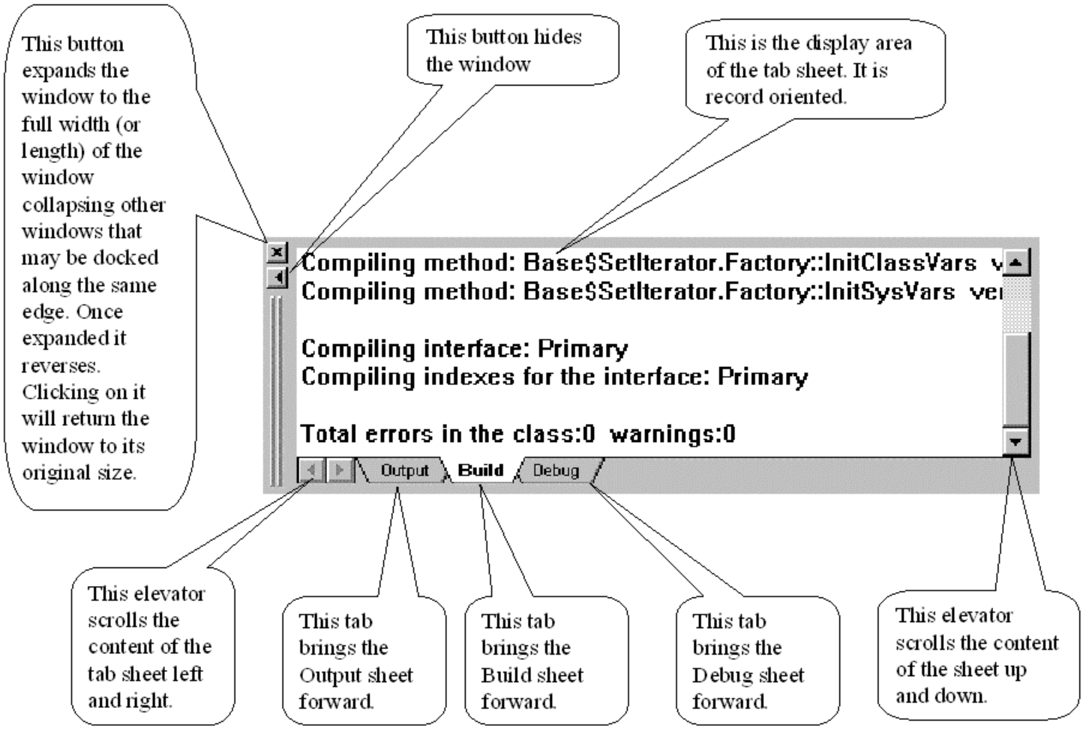
In the previous section, all the traditional MUMPS utilities are defined as an Add-in program.

Output Window

Output Window Explained

Components of the Output Window

The output window appears as a resizable window. It can be resized and docked to any of the four edges of the Main Window.



```
Compiling method: Base$SetIterator.Factory::InitClassVars v
Compiling method: Base$SetIterator.Factory::InitSysVars ve

Compiling interface: Primary
Compiling indexes for the interface: Primary

Total errors in the class:0  warnings:0
```

This button expands the window to the full width (or length) of the window collapsing other windows that may be docked along the same edge. Once expanded it reverses. Clicking on it will return the window to its original size.

This button hides the window

This is the display area of the tab sheet. It is record oriented.

This elevator scrolls the content of the tab sheet left and right.

This tab brings the Output sheet forward.

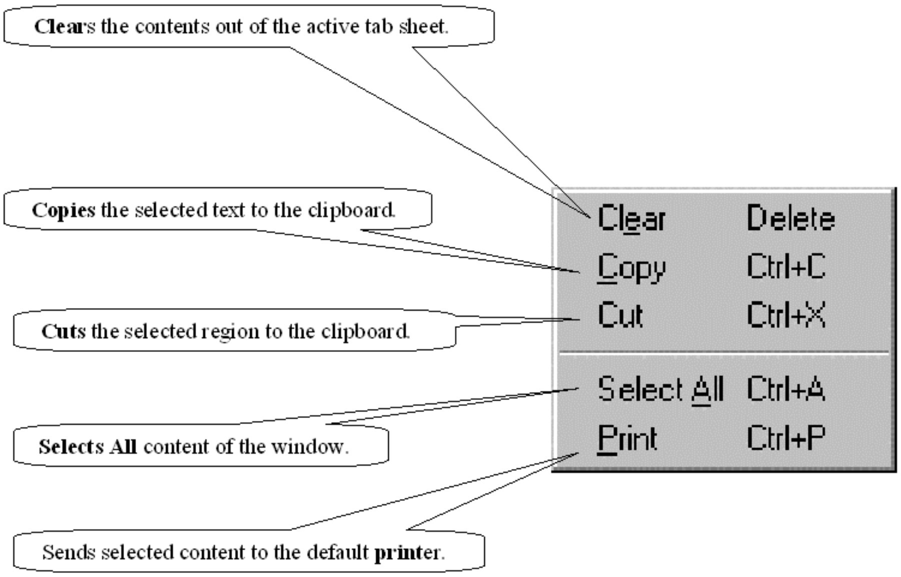
This tab brings the Build sheet forward.

This tab brings the Debug sheet forward.

This elevator scrolls the content of the sheet up and down.

Output Window Popup Menu

The Output Window popup menu is invoked by right clicking anywhere on the output window.



Clears the contents out of the active tab sheet.

Copies the selected text to the clipboard.

Cuts the selected region to the clipboard.

Selects All content of the window.

Sends selected content to the default **printer**.

Docking the Output Window

The output window may be docked to any of the four edges of the main window. The Output Window can only be docked - it cannot be undocked. However, you can freely reposition it along any of the four edges of the Main Window. When you are first getting used to the output window, it is easy to make the mistake of *accidentally* docking it while repositioning. By holding down the **Ctrl** key while repositioning the output window prevents it from docking, thereby avoiding this problem.

To reposition the Output Window, simply place the mouse pointer on the grab bar and press and hold the left mouse button down. Now drag the Output Window to any edge of the main Window and drop it. The window should dock to the edge. At this point you may want to resize the window. Simply grab the edge and expand it.

Using the Output Window

Debug and Build Tab Sheets

The **Debug** and **Build** tabs sheet are used by the EsiObjects system. All tab sheets in the Output Window are record oriented and output only. You can perform standard cut and paste operations individual or multiple records. A popup menu invoked by right clicking on the tab or on a selected item provides these operations as well as the main Edit menu.

Output Tab Sheet

The **Output** tab sheet, however, is available to you for use. It can be used to display status information and is also convenient for displaying trace information when debugging your application.

If you are not familiar with the concept behind the EsiObjects special variable **$ENVIRONMENT**, refer to the EsiObjects Server Environment section of this guide.

The EsiObjects environment object, accessed via the $ENVIRONMENT variable, contains a method called Output. It takes one parameter - a string. The following example illustrates who to put a line of text out to the Output Window.

```
Do $Env.Output(“This is a line of text”)
```

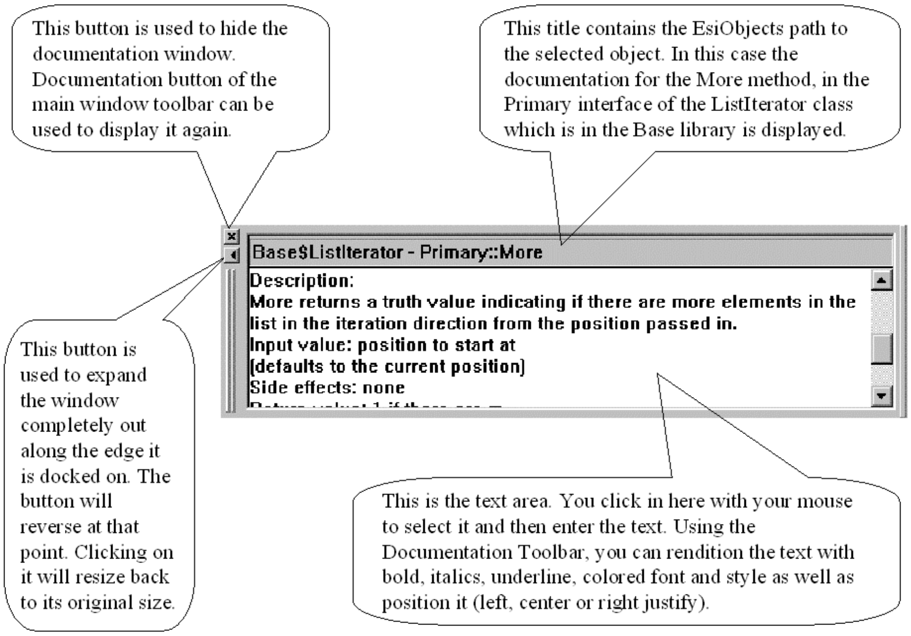
Documentation Window

Documentation Window Explained

Components of the Documentation Window

The Documentation Window is where all documentation is entered and displayed for the object currently selected. For example, if a method is selected in the Session Browser, the documentation for that method will automatically be displayed in that window. All text-editing operations are common to operations commonly found in a word processor. The text is stored in Rich Text Format (RTF). Associated with the window is a Toolbar and Popup Menu that let you perform fundamental operations.

Documentation Pane



Base$ListIterator - Primary::More

Description:
More returns a truth value indicating if there are more elements in the list in the iteration direction from the position passed in.
Input value: position to start at
(defaults to the current position)
Side effects: none

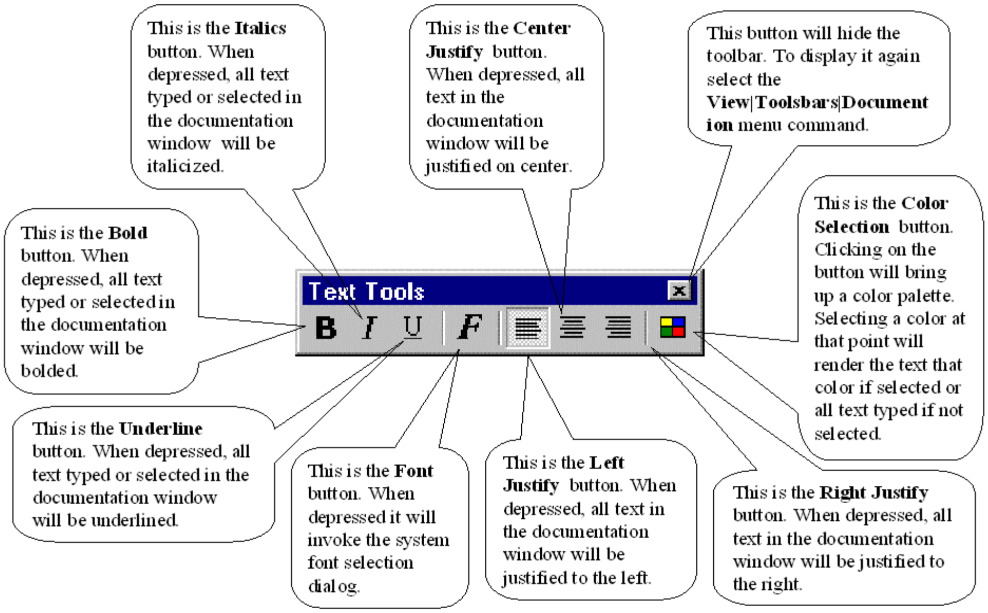
This button is used to hide the documentation window. Documentation button of the main window toolbar can be used to display it again.

This title contains the EsiObjects path to the selected object. In this case the documentation for the More method, in the Primary interface of the ListIterator class which is in the Base library is displayed.

This button is used to expand the window completely out along the edge it is docked on. The button will reverse at that point. Clicking on it will resize back to its original size.

This is the text area. You click in here with your mouse to select it and then enter the text. Using the Documentation Toolbar, you can rendition the text with bold, italics, underline, colored font and style as well as position it (left, center or right justify).

Tool Bar



This is the **Italics** button. When depressed, all text typed or selected in the documentation window will be italicized.

This is the **Center Justify** button. When depressed, all text in the documentation window will be justified on center.

This button will hide the toolbar. To display it again select the **View|Toolsbars|Documention** menu command.

This is the **Bold** button. When depressed, all text typed or selected in the documentation window will be bolded.

This is the **Color Selection** button. Clicking on the button will bring up a color palette. Selecting a color at that point will render the text that color if selected or all text typed if not selected.

This is the **Underline** button. When depressed, all text typed or selected in the the documentation window will be underlined.

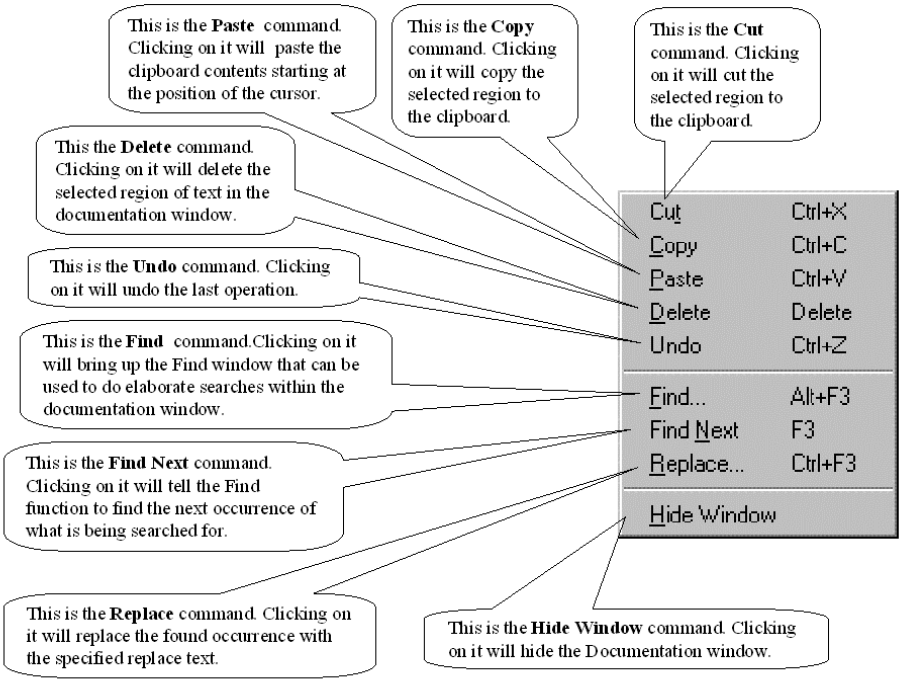
This is the **Font** button. When depressed it will invoke the system font selection dialog.

This is the **Left Justify** button. When depressed, all text in the documentation window will be justified to the left.

This is the **Right Justify** button. When depressed, all text in the documentation window will be justified to the right.

Popup Menu

The Documentation popup menu is invoked by right clicking in any documentation pane. Each command of the popup menu is explained in the illustration below.



This is the **Paste** command. Clicking on it will paste the clipboard contents starting at the position of the cursor.

This is the **Copy** command. Clicking on it will copy the selected region to the clipboard.

This is the **Cut** command. Clicking on it will cut the selected region to the clipboard.

This the **Delete** command. Clicking on it will delete the selected region of text in the documentation window.

This is the **Undo** command. Clicking on it will undo the last operation.

This is the **Find** command.Clicking on it will bring up the Find window that can be used to do elaborate searches within the documentation window.

This is the **Find Next** command. Clicking on it will tell the Find function to find the next occurrence of what is being searched for.

This is the **Replace** command. Clicking on it will replace the found occurrence with the specified replace text.

This is the **Hide Window** command. Clicking on it will hide the Documentation window.

Docking the Documentation Window

The Documentation Window can be docked to any of the four edges of the main window. The Documentation Window can only be docked - it cannot be undocked. However, you can freely reposition it along any of the four edges of the Main Window. When you are first getting used to the output window, it is easy to make the mistake of *accidentally* docking it while repositioning. By holding down the **Ctrl** key while repositioning the output window prevents it from docking, thereby avoiding this problem.

To reposition the Documentation Window, simply place the mouse pointer on the grab bar and press and hold the left mouse button down. Now drag the Documentation Window to any edge of the main Window and drop it. The window should dock to the edge. At this point you may want to resize the window. Simply grab the edge and expand it.

Using the Documentation Window

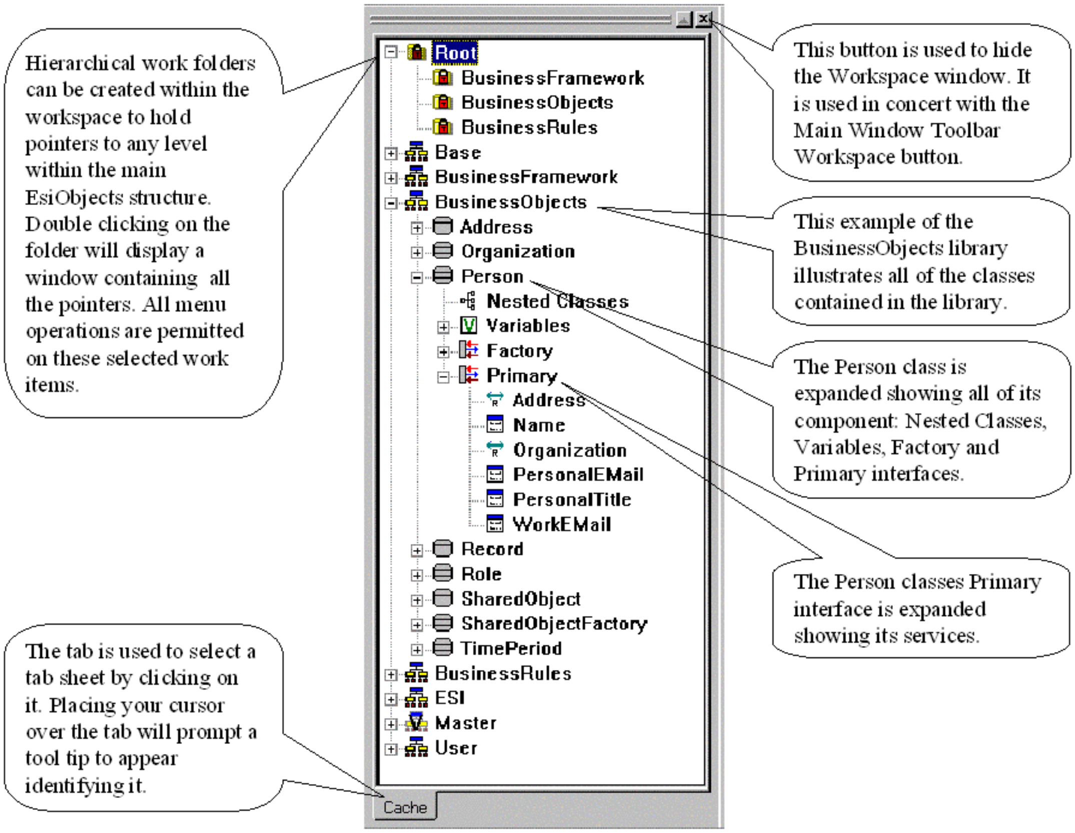
Using the documentation window is comparable to using a word processor such as Word™ or WordPad™. Associated with the Documentation Window is a tool bar (described above) that can be used to format text.

The Session Browser

The Session Browser Explained

The EsiObjects development environment is designed to provide all the necessary tools needed to develop an object oriented application.

The **Session Browser** component is a window that contains tab sheets. Each tab sheet may contain a graphical representation of EsiObjects library and folder structures. Browse operations are permitted on each structure. These operations let you find objects quickly and easily. The following is a picture of the EsiObjects **Session Browser**.



Hierarchical work folders can be created within the workspace to hold pointers to any level within the main EsiObjects structure. Double clicking on the folder will display a window containing all the pointers. All menu operations are permitted on these selected work items.

This button is used to hide the Workspace window. It is used in concert with the Main Window Toolbar Workspace button.

This example of the BusinessObjects library illustrates all of the classes contained in the library.

The Person class is expanded showing all of its component: Nested Classes, Variables, Factory and Primary interfaces.

The Person classes Primary interface is expanded showing its services.

The tab is used to select a tab sheet by clicking on it. Placing your cursor over the tab will prompt a tool tip to appear identifying it.

Using the Session Browser

Two operations are fundamental to the Session Browser. They are:

1) Docking

2) Displaying and hiding.

Docking the Session Browser

The Session Browser window must be **docked** to any of the four edges of the main window. The Session Browser can only be docked - it cannot be undocked.

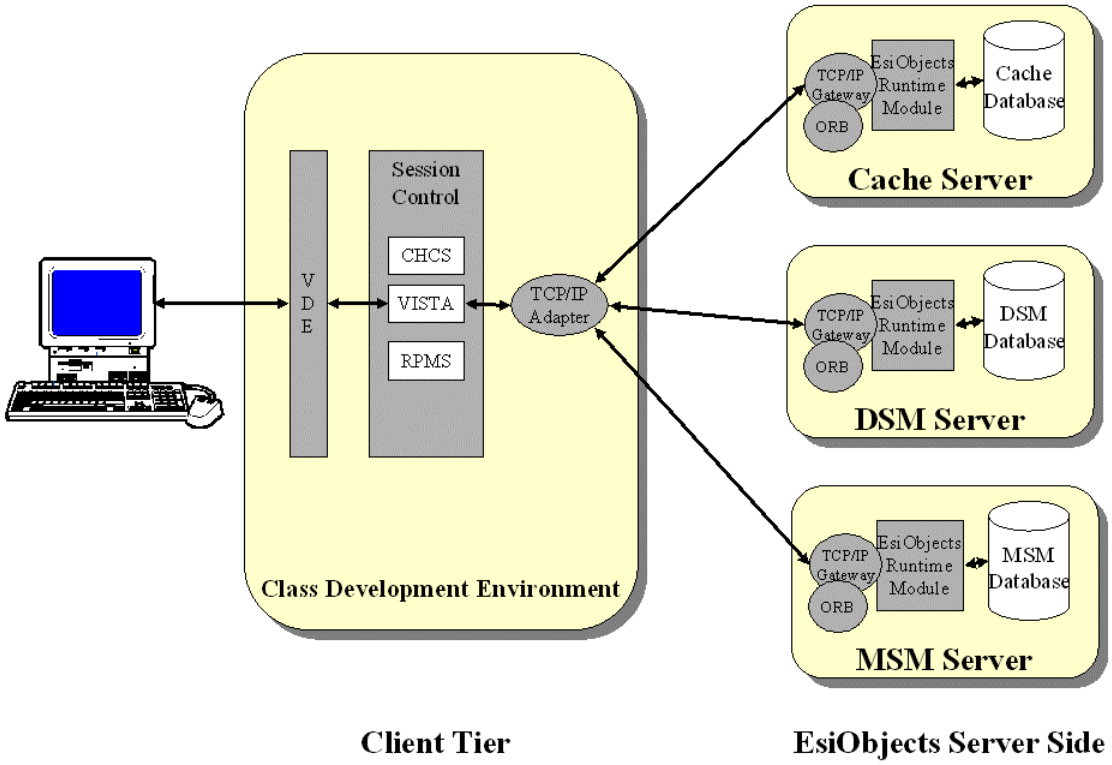
Displaying and Hiding the Session Browser

The Session Browser window may be hidden or displayed at your convenience. The **Hide or Display** the window, pop out or depress the Session Browser button on the main window toolbar with your mouse pointer. The Session Browser will appear where it was when it was hidden. You may also **hide** the window by clicking on the hide button ☒ in the upper right hand corner of the window.

Session Control

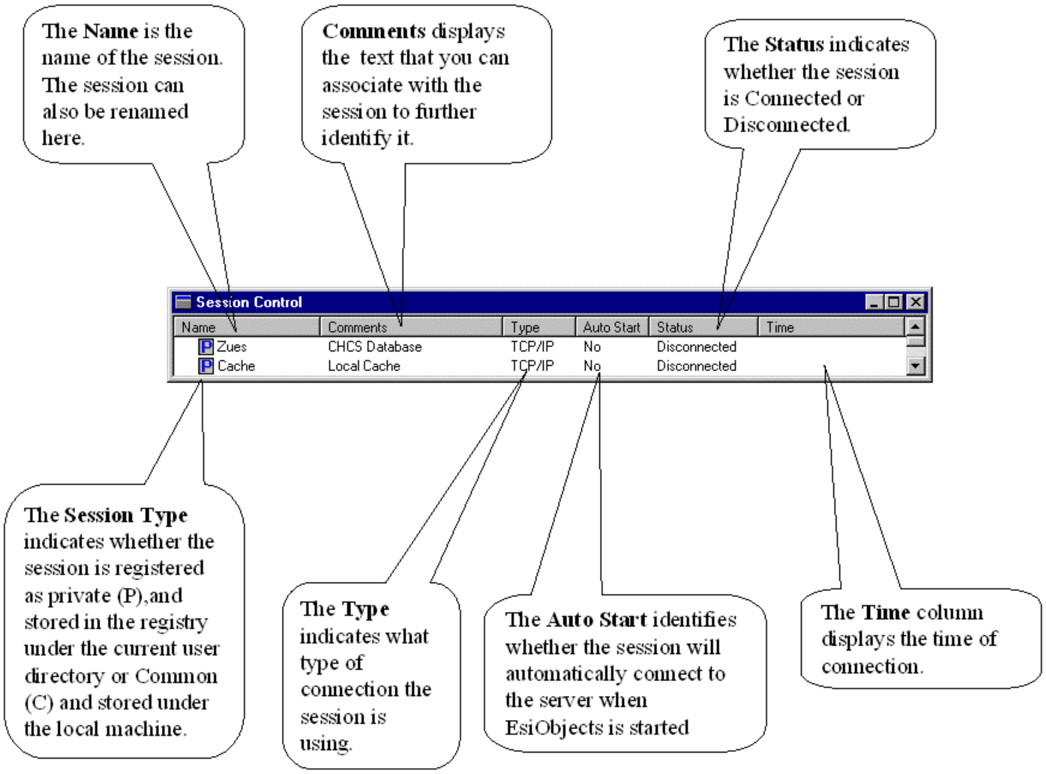
Session Control Explained

As illustrated below, Session Control allows the EsiObjects client to connect to a supported M database via a **TCP** bridge connection. Access to multiple databases on different servers through different sessions can be established on one client.



Session Control lets you maintain connections to multiple servers. When you connect to a session, all library and folder structures available in that session are displayed on a tab sheet within the Session Browser. The session name is displayed on the tab. The tab sheet that has focus identifies the session that is active. Clicking on a session tab lets you quickly switch between sessions.

Session Control Window



Name	Comments	Type	Auto Start	Status	Time
P Zues	CHCS Database	TCP/IP	No	Disconnected	
P Cache	Local Cache	TCP/IP	No	Disconnected	

The **Name** is the name of the session. The session can also be renamed here.

Comments displays the text that you can associate with the session to further identify it.

The **Status** indicates whether the session is Connected or Disconnected.

The **Session Type** indicates whether the session is registered as private (P),and stored in the registry under the current user directory or Common (C) and stored under the local machine.

The **Type** indicates what type of connection the session is using.

The **Auto Start** identifies whether the session will automatically connect to the server when EsiObjects is started

The **Time** column displays the time of connection.

Session Control Menus

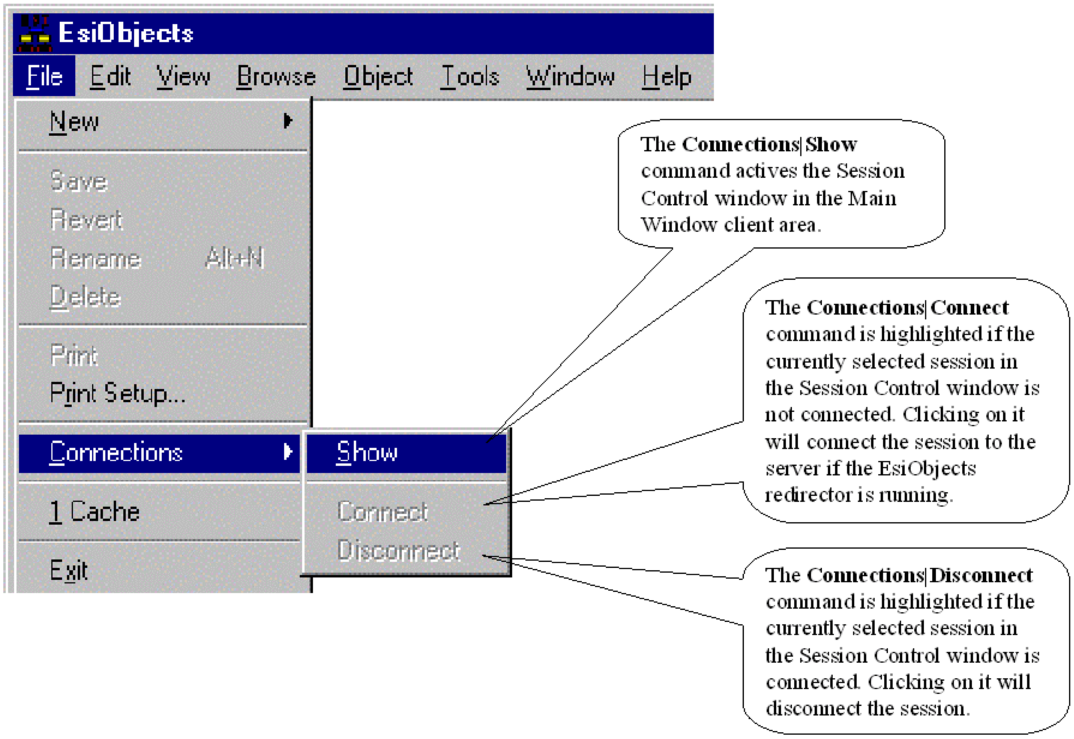
There are two menus that are available for executing session control commands: the Main Menu and the Session Control Window Popup Menu.

Session Control Main Menu

When the Session Control is displayed in the client area of the Main Window and is selected, the Main Menu will highlight those session control commands that are active. Each active command is described below.

The File|Connections Commands

The **File|Connections** commands are used to display the session control window as well as connect and disconnect to the EsiObjects server.



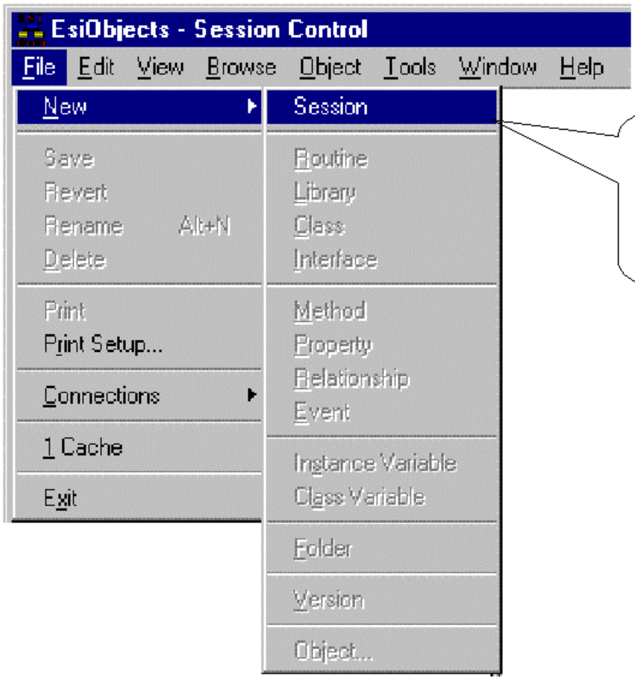
The **Connections|Show** command actives the Session Control window in the Main Window client area.

The **Connections|Connect** command is highlighted if the currently selected session in the Session Control window is not connected. Clicking on it will connect the session to the server if the EsiObjects redirector is running.

The **Connections|Disconnect** command is highlighted if the currently selected session in the Session Control window is connected. Clicking on it will disconnect the session.

The File|New|Session Command

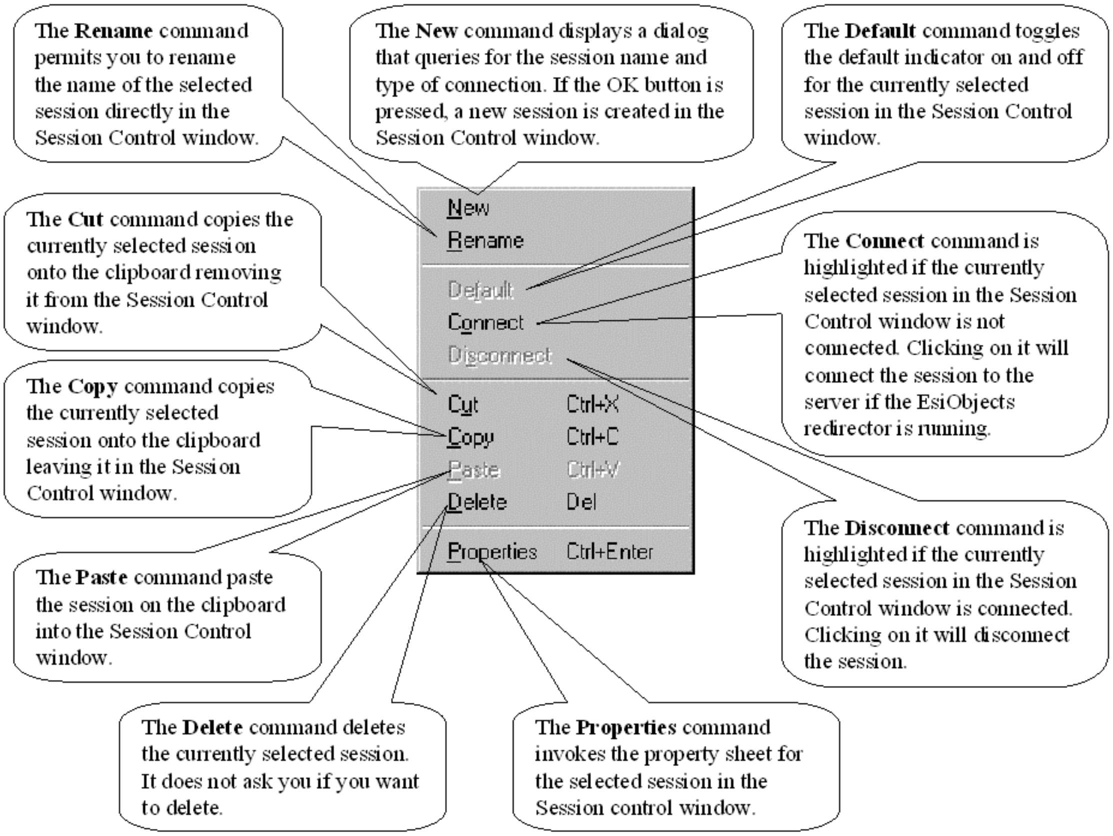
The **File|New|Session** command is used to create a new session.



The **New|Session** command displays a dialog that queries for the session name and type of connection. If the OK button is pressed, a new session is created in the Session Control window.

Session Window Popup Menu

Right clicking on the session icon will display the session control popup menu. Each command is described below.



The **Rename** command permits you to rename the name of the selected session directly in the Session Control window.

The **New** command displays a dialog that queries for the session name and type of connection. If the OK button is pressed, a new session is created in the Session Control window.

The **Default** command toggles the default indicator on and off for the currently selected session in the Session Control window.

The **Cut** command copies the currently selected session onto the clipboard removing it from the Session Control window.

The **Connect** command is highlighted if the currently selected session in the Session Control window is not connected. Clicking on it will connect the session to the server if the EsiObjects redirector is running.

The **Copy** command copies the currently selected session onto the clipboard leaving it in the Session Control window.

The **Disconnect** command is highlighted if the currently selected session in the Session Control window is connected. Clicking on it will disconnect the session.

The **Paste** command paste the session on the clipboard into the Session Control window.

The **Delete** command deletes the currently selected session. It does not ask you if you want to delete.

The **Properties** command invokes the property sheet for the selected session in the Session control window.

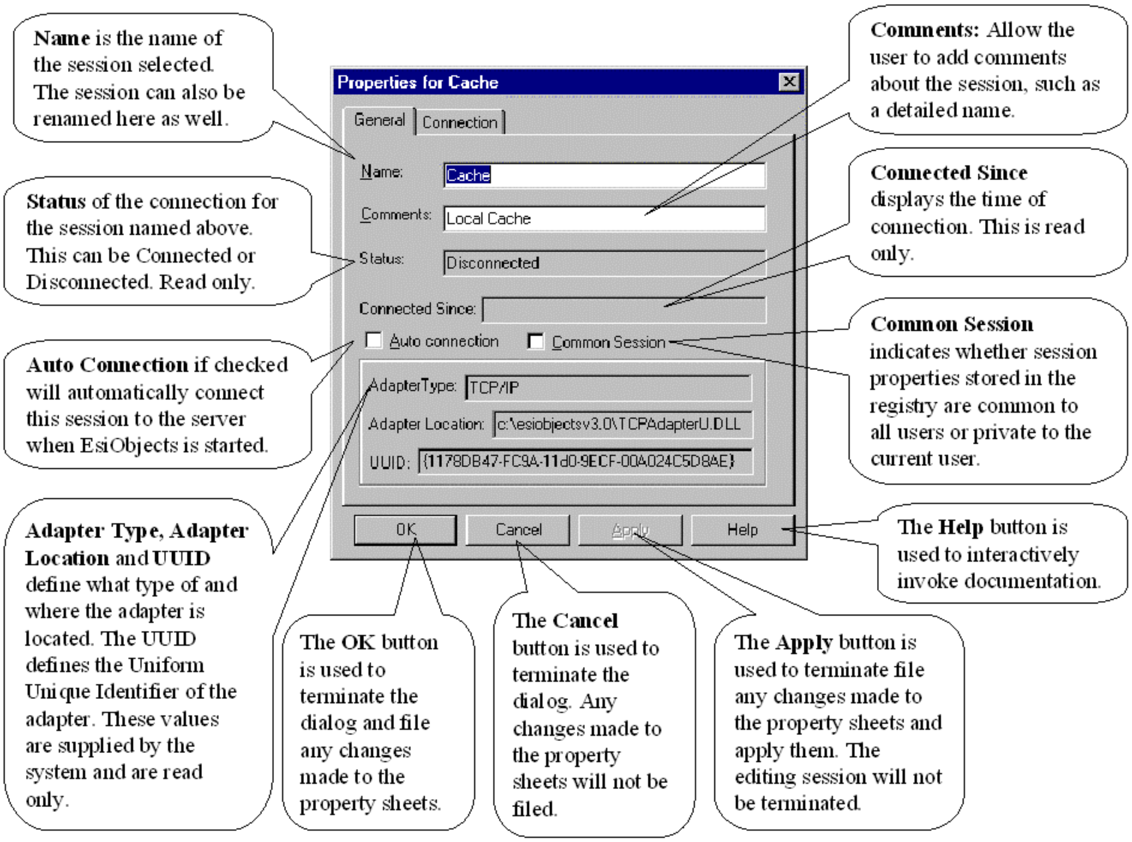
Session Control Properties

See the EsiObjects Overview section in the EsiObjects Programmer’s Reference Guide for an overview of Session Control in EsiObjects.

A sessions properties can be edited via the a property sheet that is brought up via the **File|Object|Properties** command of the main menu or the **Properties** command on the session window popup menu.

General Tab Sheet

The General tab sheet shown below is general to all connection types.



Name: Cache

Comments: Local Cache

Status: Disconnected

Connected Since:

□ Auto connection □ Common Session

AdapterType: TCP/IP

Adapter Location: c:\esiobjectsv3.0\TCPAdapterU.DLL

UUID: {1178DB47-FC9A-11d0-9ECF-00A024C5D8AE}

Name is the name of the session selected. The session can also be renamed here as well.

Status of the connection for the session named above. This can be Connected or Disconnected. Read only.

Auto Connection if checked will automatically connect this session to the server when EsiObjects is started.

Adapter Type, Adapter Location and **UUID** define what type of and where the adapter is located. The UUID defines the Uniform Unique Identifier of the adapter. These values are supplied by the system and are read only.

Comments: Allow the user to add comments about the session, such as a detailed name.

Connected Since displays the time of connection. This is read only.

Common Session indicates whether session properties stored in the registry are common to all users or private to the current user.

The **Help** button is used to interactively invoke documentation.

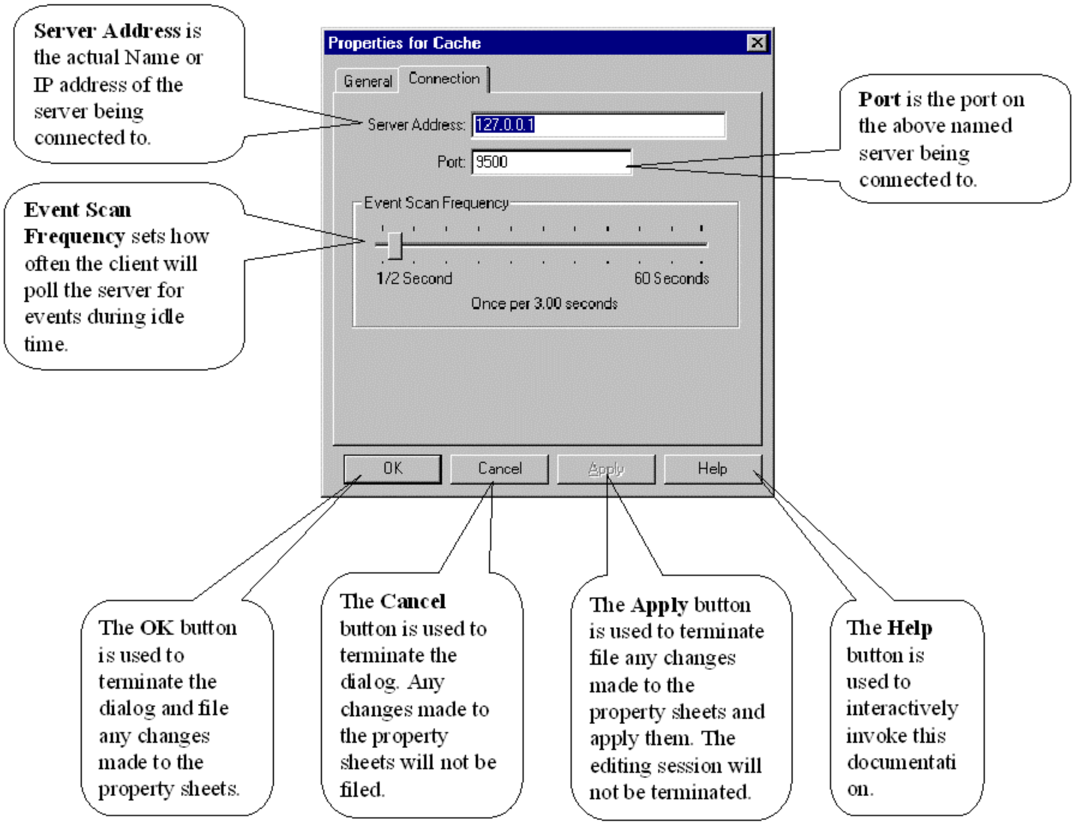
The **OK** button is used to terminate the dialog and file any changes made to the property sheets.

The **Cancel** button is used to terminate the dialog. Any changes made to the property sheets will not be filed.

The **Apply** button is used to terminate file any changes made to the property sheets and apply them. The editing session will not be terminated.

Connection Tab Sheet

The Connection tab sheet shown below is specific to a TCP/IP connection.



Server Address: 127.0.0.1

Port: 9500

Once per 3.00 seconds

Server Address is the actual Name or IP address of the server being connected to.

Port is the port on the above named server being connected to.

Event Scan Frequency sets how often the client will poll the server for events during idle time.

The **OK** button is used to terminate the dialog and file any changes made to the property sheets.

The **Cancel** button is used to terminate the dialog. Any changes made to the property sheets will not be filed.

The **Apply** button is used to terminate file any changes made to the property sheets and apply them. The editing session will not be terminated.

The **Help** button is used to interactively invoke this documentation.

Using Session Control

When the EsiObjects client is first brought up, it may or may not automatically connect to a session. If it is the first time you started EsiObjects it will not start a session. You must create a session. However, this means that EsiObjects must reside on the server you are connecting to. To find out how to install EsiObjects on the server, please refer to the appropriate read me file for the M system you will be installing it on. .

Creating a New Session

If the Session Control window is not open, use the **File|Session|Show** menu option to display it in the client area.

Once the session control window is open, right click in the open window region to open the popup menu.

Select **New** to create a session. This will invoke a New Session dialog.

Enter the **Name** of the session to be created. The name is only used to reference the session and does not require any special naming conventions.

Select the connection **Type** from the pull down menu list. Currently the only supported connection types are TCP/IP.

Click the **OK** button to save or the **CANCEL** button to exit the dialog without saving.

Now that a new session has been created and the type of connection indicated, it is necessary to define the connectivity information.

Select the session just created by left clicking on the item once and then right click to call the popup menu. Select **Properties** from the menu.

A dialog appears with the tabs **General** and **Connection**. (See the Session Control Properties under the Session Control Explained section above for explanations of the property sheets).

Once all the appropriate information has been entered in the **Properties** dialog, right click on the session. Select **Connect** from the popup menu (or main menu) to start the session. (Note that if AutoConnect is enabled, the session will connect when EsiObjects is started without this step.)

The status of the session will change from Disconnected to Connected and all libraries and folders available within that session will be available on a tab sheet of the Session Browser.

In the event that a connection cannot be established refer to the section on Troubleshooting Session Control below.

The Class Development Environment (CDE) cannot be accessed on a server until a session has been connected. It should be noted that more than one session can be opened at the same time to different servers and databases using different connections.

Create a Session by Copying and Renaming

A convenient way to create a new session is to Copy an existing session and rename. Right click on the session to be copied and select Copy on the pop-up menu.

Next, move the mouse pointer over the blank window and right click. Select Paste from the pop up menu. Notice a “copy” of the session appears next to the original session. This copy has all of the exact characteristics and settings of the copied session and may need to be edited to meet the users needs.

Right click on the new session copy and select Rename from the pop up menu. Rename the session.

Disconnecting a Session

To disconnect an EsiObjects session, right click on the session and select Disconnect from the pop-up menu. The status of the session in the session control window will change to Disconnected and it's tab sheet in the Session Browser window will disappear.

Additionally, when the EsiObjects client is shut down, accordingly, the session(s) are disconnected.

Deleting a Session

Select the session from the Session Control window that you want to delete.

Right click on the session name to bring up the popup menu.

Select the Delete command. The Session will be removed from the Session Control window without confirmation.

Troubleshooting Session Control

If a user has trouble connecting to a server with session control, there are possible steps that can be taken to isolate and correct the problem.

The Properties of the session being used to connect to a server should be reviewed first, specifically the Connection tab settings.

1. Make sure the IP Address and Port name are correctly entered and correct.
2. Verify the server being connected to is started and that the TCP/IP listener is running at this address and port.

User Options

User Options Explained

EsiObjects offers you numerous alternatives to customizing your development environment.

You can personalize your environment by storing your user name and initials. Your initials are associated with all code bodies you create or modify. Your initials can be one of the key search strings used by the EsiObjects Search engine.

You can store macro templates that can stamp out new source code objects (methods, properties and events) with text and code that is specific to the code body. These templates can save a great deal of time.

You can also control the font and size of all text and code displayed in each of the documentation and source code editing windows.

Finally, numerous options can be checked on or off fit your workflow patterns.

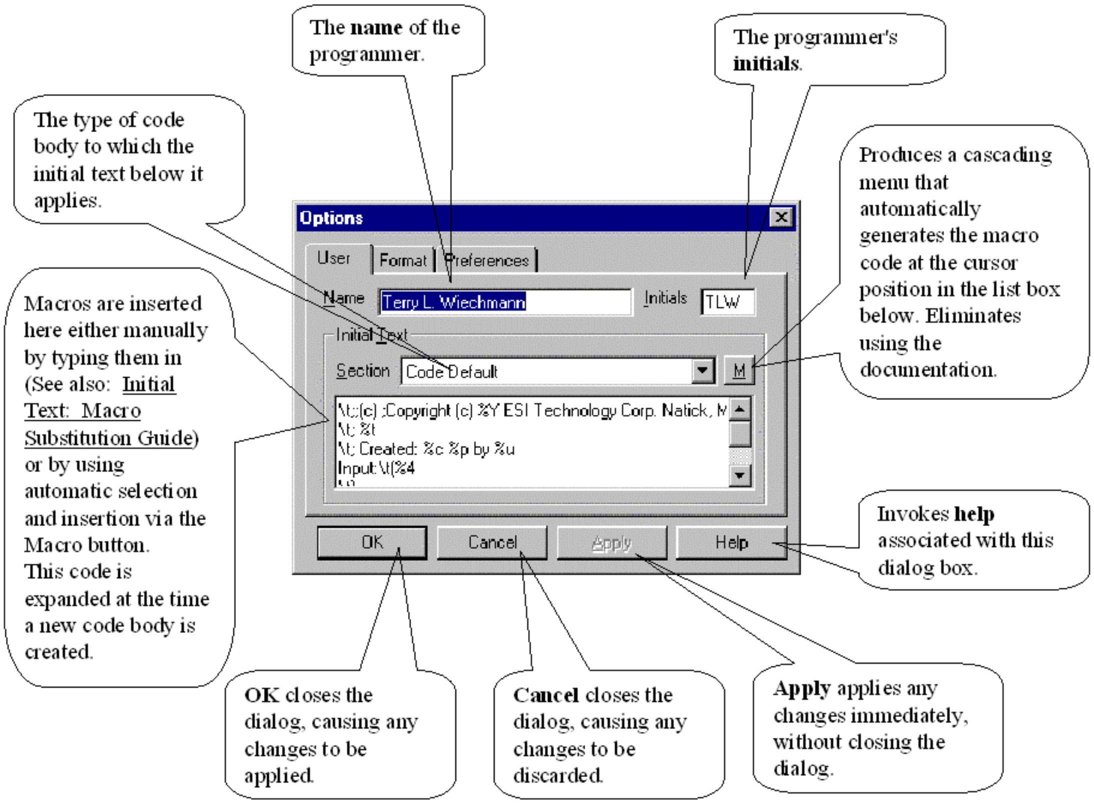
Using the User Options

From the **Tools** menu, choose the **Options** item. A dialog box appears that contains three tab sheets: User, Format and Preferences. The User sheet lets you create template information for documentation and source code associated with each method, property or event. The Format sheet lets you select how the text is rendered to you, that is, the font and size of the text. The Preferences sheet lets you tailor your EsiObjects operating environment to your needs.

User Tab Sheet

The User sheet is invoked by clicking on the **User** tab. The illustration below describes all components of the User tab of the Options property sheet. The **Name** and **Initials** fields identify you as a user. These fields are important in that they are used to stamp objects created by you. Also, the Search engine uses your initials as one type of search criteria.

The Initial Text section lets you create macrocode for each source code and documentation objects in the system (methods, property accessors, events, routines). The macrocode is expanded whenever you create a new object. You can use the macrocode to set up default code and documentation templates for your project. For more information on the available macros, see the section Macro Substitution Token List.



Name: Terry L. Wiechmann

Initials: TLW

Section: Code Default

```
\t;;(c) ;Copyright (c) %Y ESI Technology Corp. Natick, M
\t; %t
\t; Created: %c %p by %u
Input:\t(%4
```

The **name** of the programmer.

The programmer's **initials**.

The type of code body to which the initial text below it applies.

Produces a cascading menu that automatically generates the macro code at the cursor position in the list box below. Eliminates using the documentation.

Macros are inserted here either manually by typing them in (See also: Initial Text: Macro Substitution Guide) or by using automatic selection and insertion via the Macro button. This code is expanded at the time a new code body is created.

Invokes **help** associated with this dialog box.

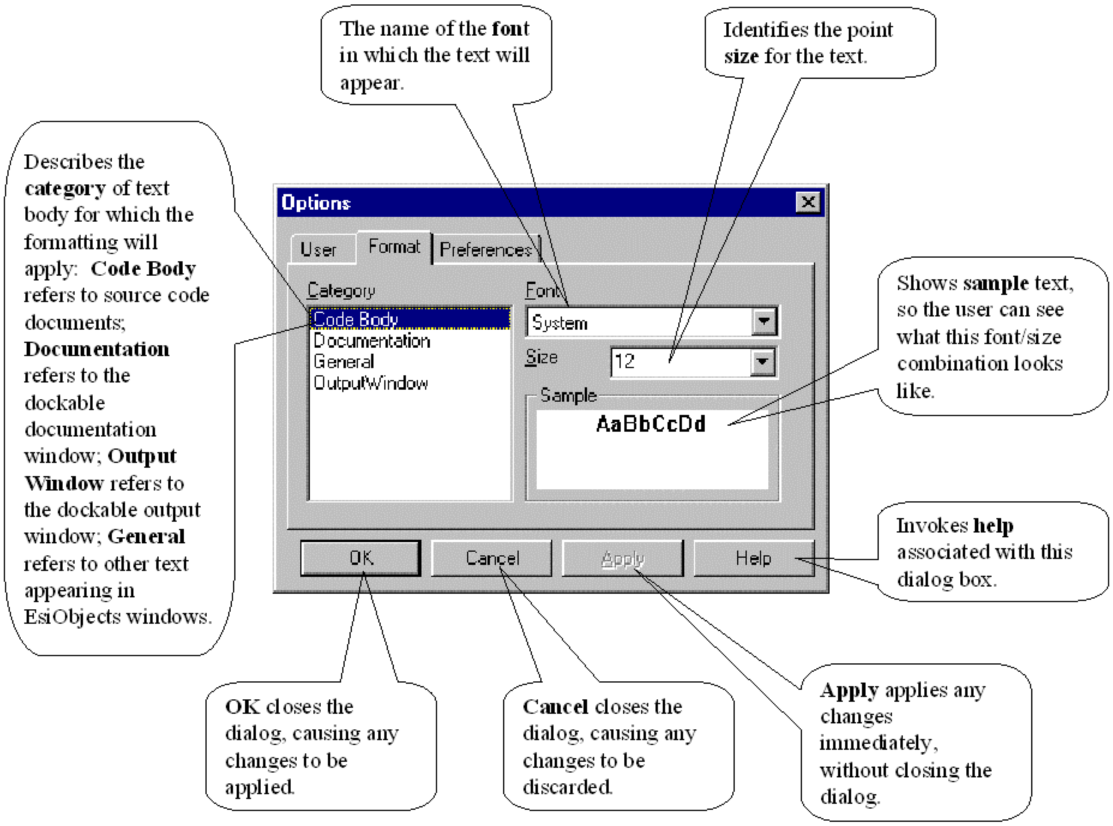
OK closes the dialog, causing any changes to be applied.

Cancel closes the dialog, causing any changes to be discarded.

Apply applies any changes immediately, without closing the dialog.

Format Tab Sheet

The Format sheet is invoked by clicking on the **Format** tab. Illustrated below is the Options property sheet Format. The Format sheet allows you to alter the text font and size for the various text components of EsiObjects.



Font: System

Size: 12

The name of the **font** in which the text will appear.

Identifies the point **size** for the text.

Describes the **category** of text body for which the formatting will apply: **Code Body** refers to source code documents; **Documentation** refers to the dockable documentation window; **Output Window** refers to the dockable output window; **General** refers to other text appearing in EsiObjects windows.

Shows **sample** text, so the user can see what this font/size combination looks like.

Invokes **help** associated with this dialog box.

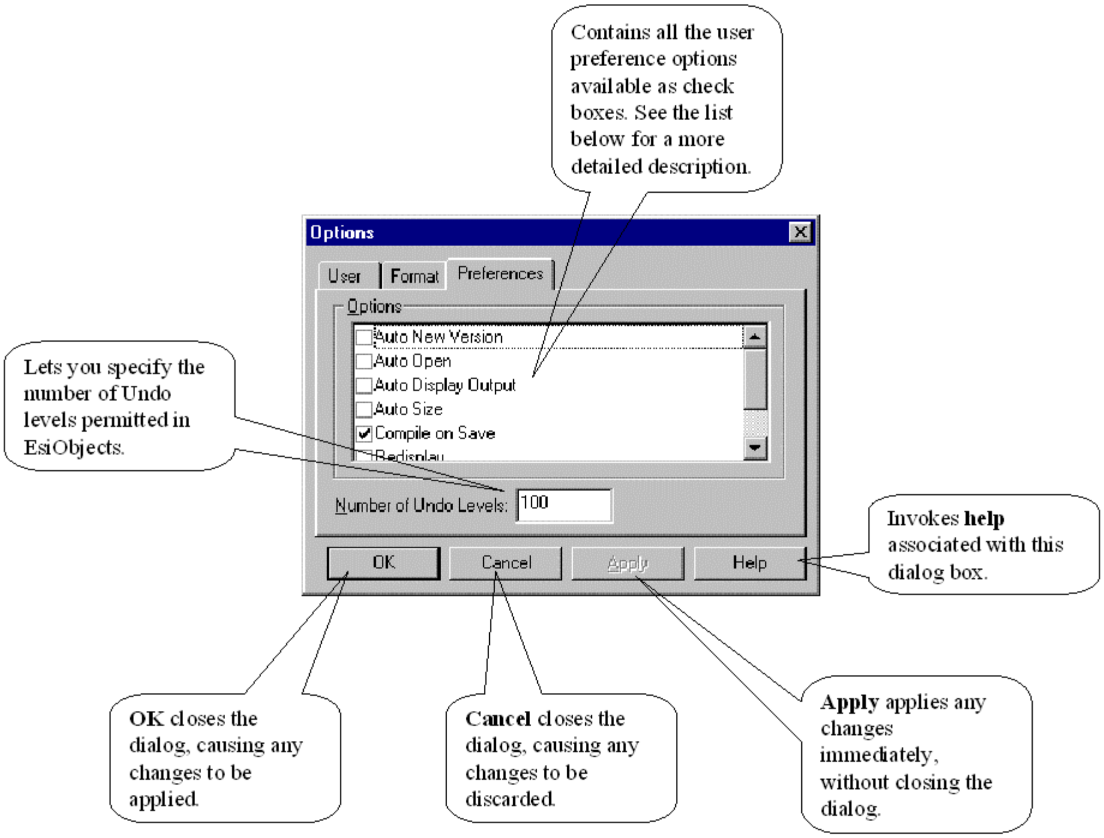
OK closes the dialog, causing any changes to be applied.

Cancel closes the dialog, causing any changes to be discarded.

Apply applies any changes immediately, without closing the dialog.

Preferences Tab Sheet

The Preferences sheet is invoked by clicking on the **Preferences** tab. This sheet allows specific options to be activated by checking the appropriate check box. Options that are checked are enabled. Options that are not checked are disabled.



Options

☐ Auto New Version
☐ Auto Open
☐ Auto Display Output
☐ Auto Size
☑ Compile on Save
☐ Redisplay

Number of Undo Levels: 100

Contains all the user preference options available as check boxes. See the list below for a more detailed description.

Lets you specify the number of Undo levels permitted in EsiObjects.

Invokes **help** associated with this dialog box.

OK closes the dialog, causing any changes to be applied.

Cancel closes the dialog, causing any changes to be discarded.

Apply applies any changes immediately, without closing the dialog.

The following table contains a description of each option available on the Preferences sheet.

Item	Description
Auto Display Output	If checked, the Output window will automatically appear if hidden, when the results of a compile are sent to the Build tab sheet.
Allow Delete of Folders	If checked, the Delete command will appear in the folder popup menus (and the main menu **Edit|Delete** command). Since all actions within the Folder are directed to the object in the library structure, unintended use of the delete command can have damaging consequences. Therefore, use of it is left up to the programmer as a personal preference.
Auto New Version	When prompted to save source code, the **New Version** check box on the Save dialog will default to what is set here. When checked, the default action on saving source code will be to create a new version of the source code on every save.

Auto Open If checked, adding a new method, property, event or relationship in the Session Browser will automatically open the appropriate editor for that object.

Compile On Save When prompted to save source code, the **Compile** check box on the Save dialog will default to what is set here. When checked, the default action on saving source code will be to compile the source code after saving.

Redisplay When checked, EsiObjects will search for an open editor or search result window to reuse and bring forward, rather than always creating a new window. This applies to the method, property, event editors as well as the search results windows.

Reuse Search Window When a search is done, a search dialog box is displayed that lets you identify search criteria and range. Checking this box will ensure that it is reused if you specify another search. Unchecked means a new dialog will be used every time a search is done.

Verify Application Close If checked, EsiObjects will prompt you for verification of shut down. If not checked, EsiObjects will simply shut down without verification. It is advised to leave this checked all the time.

Macro Substitution Token List

The following special formatting tokens are available for use in the Initial Text field of the User Options dialog. They can be typed directly into the Initial Text list box or they can by automatically entered via the Macro button on the User tab sheet.

Using the Macro button and cascading menus lets you quickly create a template without the burden of knowing what each token stands for.

Token	Meaning
%1	Properties – Input Specification
%2	Properties – Body
%a	Abbreviated weekday name.
%A	Full weekday name.
%b	Abbreviated month name.
%B	Full month name.
%c	Date and time representation appropriate for locale.
%#c	Long date and time representation, appropriate for locale. For example, "Wednesday, January 17, 1996, 12:34:56".
%d	Day of month as decimal number, leading zeros included (01-31).
%#d	Day of month as decimal number, no leading zeros (1-31).
%H	Hour in 24-hour format, leading zeros included (01-24).
%#H	Hour in 24-hour format, no leading zeros (1-24).
%I	Hour in 12-hour format, leading zeros included (01-12).
%#I	Hour in 12-hour format, no leading zeros (1-12).
%j	Day of year as decimal number, leading zeros included (001-366).
%#j	Day of year as decimal number, no leading zeros (1-366).
%m	Month as decimal number, leading zeros included (01-12).
%#m	Month as decimal number, no leading zeros (1-12).
%M	Minute as decimal number, leading zeros included (00-59).
%#M	Minute as decimal number, no leading zeros (0-59).
%p	Current locale's AM/PM indicator for 12-hour clock.
%S	Second as decimal number, leading zeros included (00-59).

%#S	Second as decimal number, no leading zeros (0-59).
%U	Week of year as decimal number, with Sunday as first day of week, leading zeros included (00-51).
%#U	Week of year as decimal number, with Sunday as first day of week , no leading zeros (0-51).
%w	Weekday as decimal number with Sunday as 0 (0-6).
%W	Week of year as decimal number with Monday as first day of week, leading zeros included (00-51).
%#W	Week of year as decimal number with Monday as first day of week, no leading zeros (00-51).
%x	Date representation for current locale.
%#x	Long date representation appropriate for current locale, e.g. "Wednesday, January 17, 1995"
%X	Time representation appropriate to current locale.
%y	Year without century, as decimal number, leading zeros included (00-99).
%#Y	Year without century, as decimal number, no leading zeros (0-99).
%z	Time zone name or abbreviation, if known - No characters, if not known.
%%	Percent sign (%) appears in target text.
%u	User name, from Options dialog.
%i	User initials, from options dialog.
%n	Name of entity to which the text applies.
%t	Full title of entity to which the text applies.
\t	Inserts a tab character (ASCII 9).
\n	Inserts a newline character (ASCII 10). These generally don't appear in source text, except in a CR+LF combination at the end of each line.
\r	Inserts a carriage return character (ASCII 13). These generally don't appear in source text, except in a CR+LF combination at the end of each line.
Ctrl+Enter	Inserts carriage return, line feed combination. Same as **\r\n**, but easier to see visually.
\	Inserts a backslash (****) in the target text.

Setting The Initial Text

To set the initial text for any code body, follow the step outlined below.

1. Select the **Options…** option from the **Tools** menu.

3) In the **Section** combo box, select the type of code body you wish to enter initial text for. **Code Default** will apply to all method types. In the example below, **Method** was selected.

2. Place the cursor where you want the macro to expand. Enter the macro and/or text you wish to have expand in a new code body. Alternatively, use the Macro button to popup a cascading menu to insert the macro automatically. Existing methods are not affected by any changes made here. You can enter text or any of the macro substitutions described in the table above.

See the following topic for an example of setting initial text using macro substitution.

Initial Text Example

The following example illustrates how to set initial text for methods. It also shows how macro substitution works, by showing the initial text as it appears before macro substitution, and as the actual text might appear in a given example.

(Note that these examples can be cut out of this document and pasted into the Initial Text list box.)

Initial Text

The returns in the following text were achieved by pressing **Ctrl+Enter** in the Initial Text field of the User Options dialog. A less visually ambiguous alternative would be to insert the text **\r\n** for each return; however, this is less visually appealing.

```
\t;;(c) ;Copyright (c) 1997-%Y ESI Technology Corp. Natick, MA

\t; %t

\t; Created: %c %p by %u

Input:\t(%4

\t)

\t; %3 method code begins here.

\tQUIT
```

Default Source Text

The following default source code text is produced for a method, when the preceding macro text is used.

```
    ;;(c) ;Copyright (c) 1997-1998 ESI Technology Corp. Natick, MA

    ; Framework$ErrorBroker - Primary::GetError

    ; Created: 12/19/98 09:59:21 AM by Terry L. Wiechmann

Input:     (

           )

    ;  method code begins here.

    QUIT
```

Help Documentation

Help Documentation Explained

The EsiObjects Help documentation is available as Acrobat .pdf files. There are 4 guides and one tutorial that are designed to cover all aspects of using the EsiObjects system. The guides and tutorial are described in the following sections.

Administrator’s Guide

Assumptions about reader:

- Have administrative skills at the OS and M levels.
- Have PC user skills.

Goals of this guide are:

- Explain the EsiObjects Client Server Environment.
- Describe how to use MSM’s RVG capability.
- Describe how to setup and use EsiObjects for each of the supported M systems: MSM, DSM, GT.M and Cache.

Language Reference Guide

Assumptions about reader:

- Knows object oriented concepts.
- Has programmed in other languages.

Goals of this guide are:

- Explain all code body structures.
- Describe the syntax of the language.
- List and describe all Commands.
- List and describe all Functions.
- List and describe all Special Variables.
- List and describe all Operators.

Programmer’s Reference Guide

Assumptions about reader:

- Knows object oriented concepts.
- Knows the MUMPS language plus the EsiObjects extensions.

Goals of this guide are:

- Give an overview of EsiObjects.
- Describe how to use Objects.
- Describe how to Integrating Objects.
- Outline the Guidelines for using Objects.
- Describe Client Server Programming using EsiObjects.

Tools Guide

Assumptions about reader:

- Knows object oriented concepts.
- Knows how to use PC applications.

Goals of this guide are:

- Give an overview of EsiObjects Environment.
- Describe each component of the EsiObjects Environment and how to use them.
- Describe each component of the Class Development Environment and how to use each one.
- Describe the Transport tools and how to use each one.

Getting Started Tutorial

This tutorial is delivered with EsiObjects. It covers OO concepts and teaches you how to use the EsiObjects tools by actually constructing a small application.

Using the Help Documentation

EsiObjects documents are accessible in three ways:

1. Through the EsiObjects Help menu. Simply click on the desired guide or tutorial and the Acrobat Reader will be launched and the document displayed.
2. Via the EsiObjects installation subdirectory Help. All EsiObjects help documents reside in the Help subdirectory. Access this directory and simply double click on the desired document. Acrobat Reader will be launched and the document displayed.
3. Launch the Reader directly and open the desired file.

Class Development Environment Tools

Tools Overview

The EsiObjects object model is based on the classification system. The CDE environment contains all the tools needed to develop and test these classes. Class development tools available in the CDE fall into the following 5 categories:

- **Browser** tools designed to let you migrate all components within an EsiObjects session. You use browsers to find the definitional object you want to work on. EsiObjects provides multiple views into definitional structures, either directly through the library structure or indirectly through folders. Folders can be private to your session or shared with other programmers connected to the same session.
- **Editors** are tools that let you modify an object found through the Browser.
- **Search** tools are used to search across a range of objects in the EsiObjects library structure for specified criteria. The results of the search can be used to activate the associated editor, wizard or property sheet for the object double clicked on.
- **Debugger** is an interactive tool that lets you step through the execution sequence of your application. It automatically displays the stack and symbol table states after each step.
- **Transport** tools are used to transfer definitional level and instance level data and code between environments via external files. The transport tools can be used to share your work within a project or to back up it up for safekeeping.

Browsers

The EsiObjects CDE contains **Browser** tools designed to migrate to all objects within an EsiObjects library. You can use browsers to find the object you want to work on. EsiObjects contains the following browsers:

- Session Browser
- Object Browser

The ability to browse objects is a major part of the EsiObjects CDE. A browser is a development tool that provides a common way to migrate through relationships between definitional objects for the purpose of interrogating or modifying the internals of the object.

Everything in EsiObjects is an object including EsiObjects itself. The only difference is the type. Therefore, the browsers can migrate and display any object in the system simply because they are structurally identical although their content may be different.

Session Browser

A **Session Browser** displays and provides migration services for all structures supported within a session. Currently two structures are supported:

1) Library
2) Folder

The Session Browser is used to migrate only; it does not expose the internals of the objects.

Object Browser

The **Object Browser** is used to browse any type of object. It can be used to browse classes as well as instances of classes. It can browse static objects or objects that are being modified by a running application. It displays the internal state of the object and permits migration back and forth along object links. It has an integrated Xecute Shell that can be used to evaluate EsiObjects expressions or execute a line of EsiObjects code.

Editors and Property Sheets

The ability to edit definitional objects is a major part of the CDE. Definitional objects are based on the classification system and are called classes. Classes contain all information needed to create an instance of the class commonly referred to as an object. Instances always know who is responsible for making them. The parent class contains *methods* that give the object behavior, *properties* to expose the object's state, *relationships* that link the object to other objects and *event templates* needed to respond to unsolicited events.

An **editor** is a development tool that provides a common way to create, edit, delete, or inquire into an object, whether it is a variable, method, property, relationship or event of a class.

A **property sheet** typically presents itself as one or more tab sheets in a window that contains fields that you may modify. Often the fields are read only, that is, only available as information and not modifiable.

Search Tools

Browsers are used to directly access objects. The Session Browser can be used to access a method, property, relationship or event within an interface of a specific class if you know where it resides. Browsers are used when looking for a component that is in a known location. You can migrate the library tree with the Session Browser or, if you know its name, you can use the **GoTo Class** dialog to access it quickly.

Often, however, you will have a need to find a specific occurrence of an object that is a part of the library structure, some attribute of an object or string within an object. You generally do not know where it is or how many occurrences exist. In these cases, a search tool is needed.

The EsiObjects **Search Tool** lets you search through the library, class and interface levels within a library for specific criteria. It records the object paths where a criteria match is found. These paths are displayed in the search window. The programmer can then double click on any one of the paths to launch the appropriate editor for the object found. If the path of a property is selected, the property editor will launch. If the path of a method is selected, the method editor will be launched. Regardless of the editor, each occurrence of the search criteria will be highlighted within the edit pane.

Debugger Tool

The EsiObjects CDE contains an **interactive debugger** that is used to debug your code whether they be methods, properties, relationships or event handlers. The debugger is designed to give you control over the execution sequence. Additionally, it displays the state of the object after each step is executed. All variables accessible to the execution context are displayed as well as the execution stack. A tab sheet exists within the debugger GUI that lets you change the state of an object at any step within the debugging process.

Transport Tools

Object Transport Tools

The EsiObjects **Transport Tools** are used to package your application's definitional components for backup or transfer. EsiObjects libraries contain class hierarchies. These hierarchies contain interfaces and they contain services such as methods, properties, etc. Using the EsiObjects Transport Tools you can choose to start exporting components at any level, transferring that level and sublevels to a flat file. These files can be used as backup as well transfer work between systems.

Multiple components of the same level can be exported in the same file due to multiple selections permitted by the export utility.

The Object Transport Tools only transfer the definitional components of your application. They do not transfer any M level routines or globals. The next section addresses that capability.

There are three transport tools available:

1. **Export** is used to export definitional components to flat files within the Windows environment. The files are typed according to the level the transfer started with by using a unique file extension. For example, if you started transferring at a class level, the file extension would be .opc.
2. **Import** is used to import an exported file into the same type of component that it was exported from. That is, if it were exported as a Class, it must be imported into a class. File extensions are used to identify the file type.

3. **Generic Import** is used to import a component or components into the same component it was exported from. Generic Import knows how to import the components

Traditional M Transport Tools

EsiObjects runs within any supported ANSI Standard M system. The model side (server) of the system is written in M. It consists of M routines and globals. It will coexist with any application that does not conflict with its namespace rules. All EsiObjects routines and globals names begin with VES. The Veterans Administration has assigned this namespace.

All definitional components (libraries, classes, methods, properties, …) created by EsiObjects are mapped to routine and global namespaces when you create them. EsiObjects gives you control over the mapping. It also gives you control over the mapping of instances of those classes via the CREATE command. The Transport Tools described above let you export and import all classes and their components. However, it is often necessary to export and import object instances. Additionally, you may want to import and export traditional M application routines and globals. The Traditional M Transport Tools are designed for this purpose.

Session Browser Explained

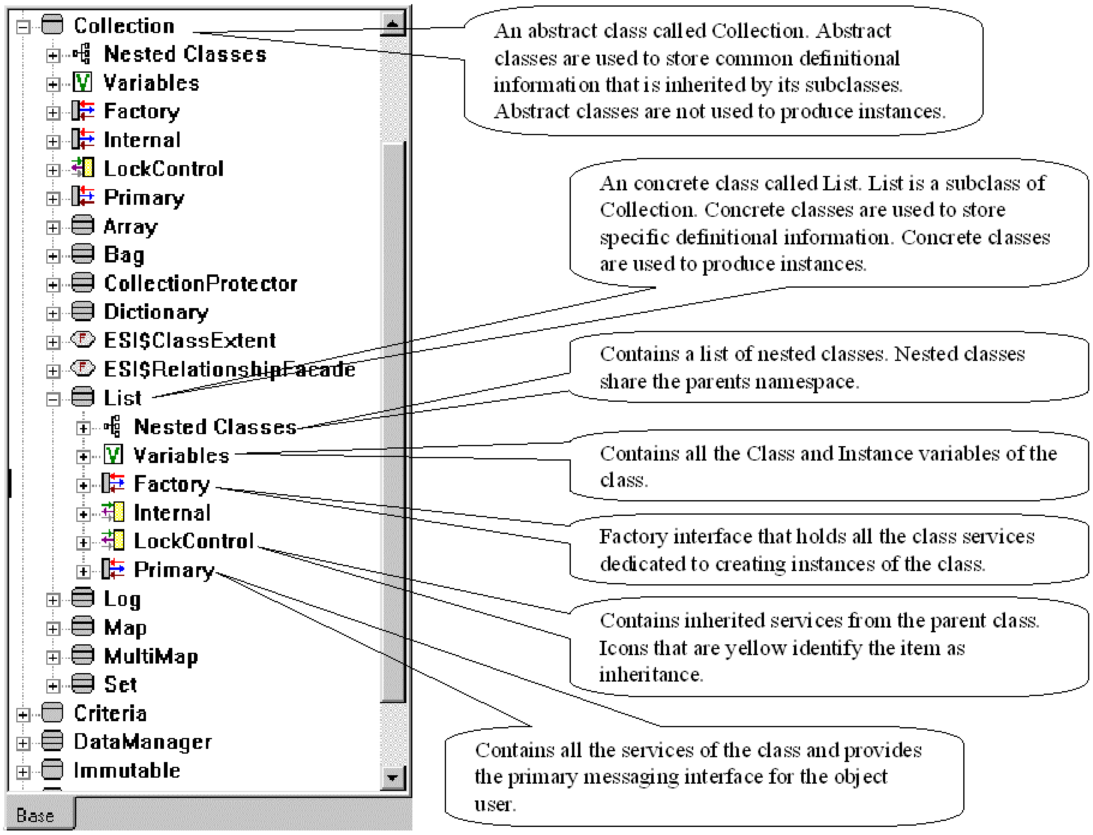
The **Session Browser** is the tool used to browse through a specific class library or folder. This tool is central to developing classes and folders.

You can add and remove classes, arrange the class hierarchy and build the class definitions. Using this browser, you add methods, properties, events, relationships and variables to define a class, and then launch the specific editors or wizards to define each of the services.

The **Session Browser** also supports folders. You can add and remove folders, arrange the folder hierarchy as well as populate the folders with objects that exist in the library structure. When the folder is opened, all of its contents are displayed in a window. From this folder you can access all the objects within it just as if you were working with them directly.

The **Session Browser** appears as a consequence of connecting to a server session. It contains all libraries and folders available in that session.

The picture below illustrates a Session Browser that contains a portion of the Base library. The major components of the class hierarchy are generally described. Each component in the hierarchy will be explained in detail in subsequent sections.



An abstract class called Collection. Abstract classes are used to store common definitional information that is inherited by its subclasses. Abstract classes are not used to produce instances.

An concrete class called List. List is a subclass of Collection. Concrete classes are used to store specific definitional information. Concrete classes are used to produce instances.

Contains a list of nested classes. Nested classes share the parents namespace.

Contains all the Class and Instance variables of the class.

Factory interface that holds all the class services dedicated to creating instances of the class.

Contains inherited services from the parent class. Icons that are yellow identify the item as inheritance.

Contains all the services of the class and provides the primary messaging interface for the object user.

Session Structures

There are two structures supported in EsiObjects: the library and folder structures. These structures are made available in the Session Browser window at the time a session is connected.

Library Structures

The **Session Browser** is used to migrate through and perform operations on class components of the EsiObjects system.

The general class structure is a hierarchy as follows:

- A **Library** contains one or more classes.
- **Classes** contain, Interfaces, Variables, Nested Classes and Subclasses.
- **Interfaces** can be added as needed and contain the services of the Class, that is, methods, properties, events and relationships.
- The **Variables** interface contains all the variables needed to define the internal state of an object (instance). Two types of variables can be defined: Instance and Class. Instance variables live within the actual object (instance) and Class variables live within the Class object.

- **Nested Classes** are classes that share the same namespace as their parents. They are an integral part of their parent's namespace. They do not inherit any of their parent's services.
- **Subclasses** are classes that do not share their parent's namespace. They do inherit their parent's variables, interfaces and services.

The Session Browser is used to browse classes within a particular library. It is used to traverse the class hierarchy, display the contents of the class interfaces and acts as a launch pad for the method, property, relationship, event and variable editors. It offers extensive functionality for the maintenance of the class hierarchy through popup and main menu commands. Classes can be added, modified and deleted from the browser.

The Session Browser presents itself to you as a tab sheet at the time a session is connected to.

Libraries Explained

EsiObjects supports Class Libraries. Libraries are used to group classes by some artificial criteria that are usually based on application or organizational requirements. Libraries provide a firewall between groups that prevent inadvertent damage to protected classes.

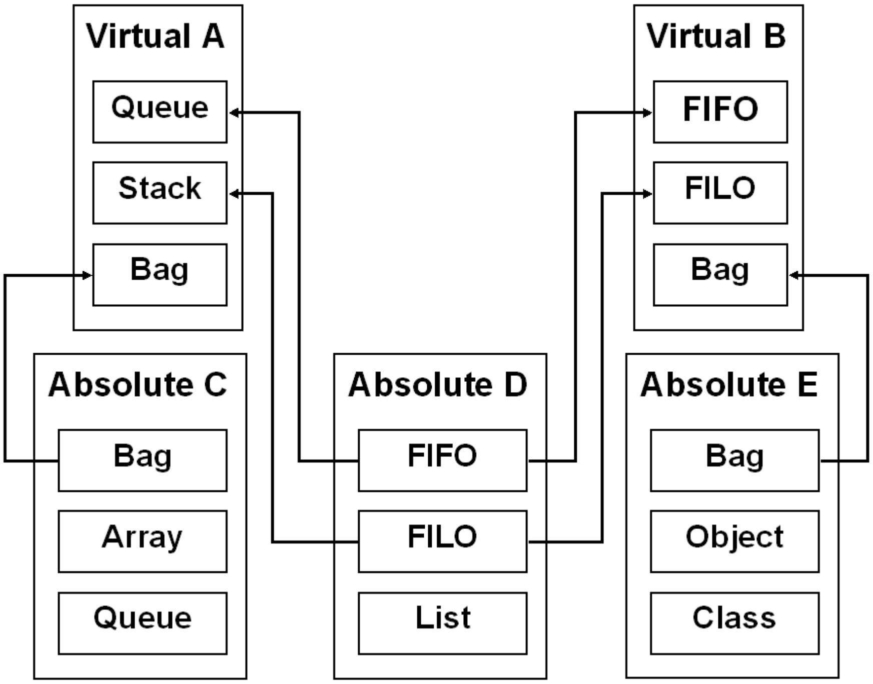
Two types of libraries are supported:

- Absolute
- Virtual

Absolute libraries physically contain classes. **Virtual** libraries do not physically contain classes; instead, they can integrate classes contained in one or more absolute libraries. A virtual library can also view some classes in absolute libraries by alternate names called Aliases. It can even have multiple entries for a single class, each under a different name.

The relationship between virtual and absolute libraries is hierarchical, and never more than one level deep. A virtual library imports classes from at least one absolute library, but there is no restriction on the number of classes it can import, and the number of absolute libraries from which they can come. Some or all of the classes in an absolute library may be exported to any virtual library. An absolute library can export its class names to many different virtual libraries. A virtual library is so flexible that it can view any combination of classes in any absolute libraries by any valid names.

The following diagram illustrates these concepts.



In this figure, there are two virtual and three absolute libraries. ***Absolute Library C*** exports its **Bag** class to ***Virtual Library A***, while ***Absolute Library E*** exports its **Bag** class to ***Virtual Library B***. This causes no conflict, but if ***Virtual Library A*** wants to import ***Absolute Library E***'s copy of **Bag**, it will have to do so under a different name.

Absolute Library D exports its **FIFO** and **FILO** classes to ***Virtual Library B*** under the same names. It also exports them to ***Virtual Library A*** under different names: **FIFO** as **Queue**, and **FILO** as **Stack**. The same class **FILO** can be viewed by one virtual library as **FILO** and by another as **Stack**. Any number of virtual libraries under any combination of names can view it.

To syntactically reference a class within a library, precede the class name with the library name separated by a $. For example, FileManager$FMRoot would access the FMRoot class within the FileManager library.

The Class Hierarchy Explained

In the Session Browser you can create new classes and link those classes in a hierarchy that takes advantage of inheritance and/or private name spacing, providing for code reuse. Additionally you can link these classes to classes in other libraries.

EsiObjects supports multiple inheritance. Therefore a class can inherit from any number of superclasses. Though multiple inheritance can add complexity to your system and should be used carefully, the benefits and flexibility provided can be worthwhile.

Additionally, EsiObjects supports nested classes. By default, classes in EsiObjects are uniquely named within the library that contains them. A class is identified by a full name of the form library$class. Nested classes provide additional flexibility in naming classes.

EsiObjects allows a class to be nested within a containing class. The name of the nested class is relative to the containing class. For example, if a class named Inner is defined as a nested class within the containing class User$Outer, then the full name of the Inner class is User$Outer>Inner. Nesting may be continued to any arbitrary depth. For example, a class named Deeper can be defined within the container class User$Outer>Inner, and its full name is User$Outer>Inner>Deeper.

The basic rule concerning the names of nested classes is that classes nested within a given containing class must have different names. In this regard, the containing class takes on some of the functionality of a library as being a scope for class names.

It is important to observe that nesting a class within a containing class does not make it a subclass. A nested class does not inherit from the containing class unless it is explicitly linked to it as a subclass.

A nested class always appears in the Session Browser as a subnode of the Nested Classes node under its containing class. This is it's principal node in the Session Browser. If a nested class is linked as a subclass of another class, then it also appears in the Session Browser under that other class, along with the subclasses of that class. But its node there is identified by the special nested class icon . This nested class node cannot be expanded, but double-clicking on it moves the Session Browser pointer to its principal node, which can be expanded.

When a class is defined as a subclass of a nested class, it is not automatically nested within the same container. To define a subclass that is also nested, first create the new nested class and then link it to the desired superclass.

Reusable Libraries Explained

EsiObjects is delivered with a comprehensive set of pre-defined classes. These classes are organized into libraries. Three libraries are delivered with EsiObjects, they are:

1. **Master** - a virtual library that contains a master list of classes in other libraries.
2. **ESI** - which contains system level support classes. Most classes are of little interest to the application programmer. However, classes like TransportType are of great interest since they provide the basic object transport capabilities needed for maximum performance in a Client Server or Internet environment. They reside in the ESI library because they are of great use to the EsiObjects system itself.
3. **Base** - which contains all the foundation classes supported by EsiObjects. They represent extensive reusability. Some classes that are delivered with the EsiObjects system are:
    - Collections, a superset of the ODMG's Collections with associated iterator classes.
    - Immutables that support date and time stamping and ranges.
    - Mix-In classes that, when dropped onto another class, provide its services through multiple inheritance.

- Name Pools that form instance hierarchies with full instance inheritance.
- Data Manager class that lets the programmer define nested Dictionaries for maintaining instance indices based on the property of a class. The Data Manager automatically maintains these instances using EsiObjects event processing capabilities.
- Criteria classes that offer filter and range criteria functionality for searching across instance ranges.

Folder Structures

Within a session, you can create hierarchical folders. These folders, when double clicked on, will display a window. Within that window, you can drag any object displayed in the Session Browser window including another folder.

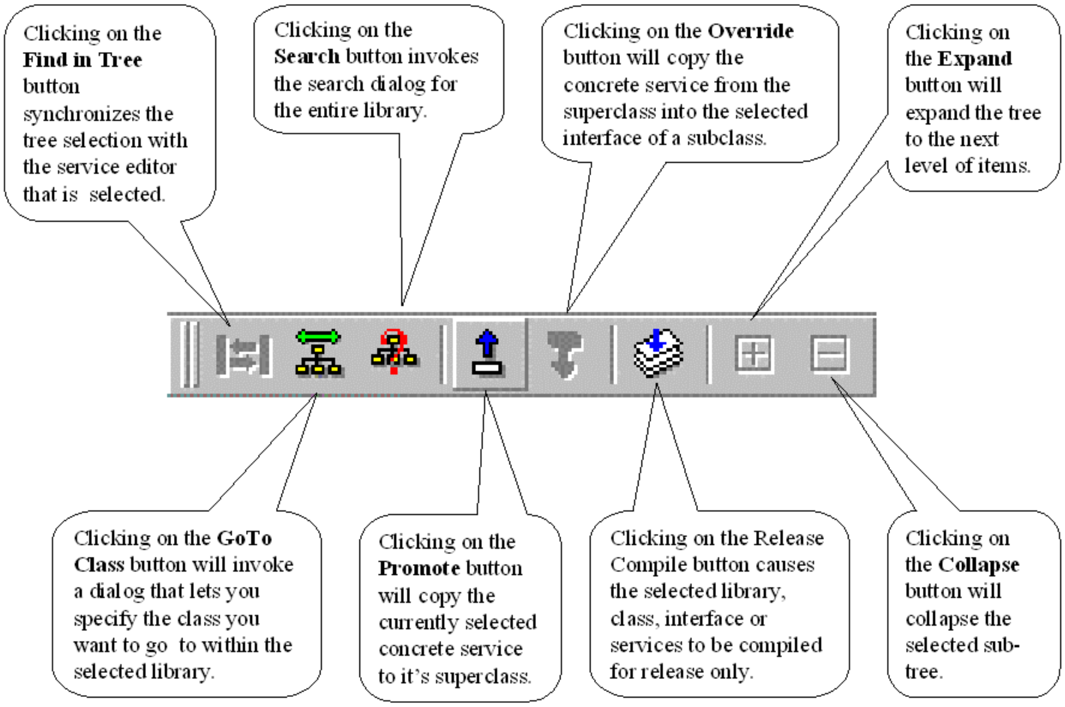
Folders are useful when working on numerous objects that are disseminated throughout the structures. They provide a way to keep your work concentrated, eliminating excessive searching and migration to work on each component. It is a workflow enhancement that increases your productivity.

Please note: Selecting an item within the Folder has the same affect as directly selecting the item in library or folder structure. All menu operations will be performed indirectly on the object selected, not directly on the item in the work window (The Remove command is an exception to this rule).

Session Browser Tools

Class Toolbar

Directly associated with the Session Browser is the Class Toolbar. If displayed, contains buttons that let you perform class operations quickly based on the selected object within a class. The toolbar is described below.



Clicking on the **Find in Tree** button synchronizes the tree selection with the service editor that is selected.

Clicking on the **Search** button invokes the search dialog for the entire library.

Clicking on the **Override** button will copy the concrete service from the superclass into the selected interface of a subclass.

Clicking on the **Expand** button will expand the tree to the next level of items.

Clicking on the **GoTo Class** button will invoke a dialog that lets you specify the class you want to go to within the selected library.

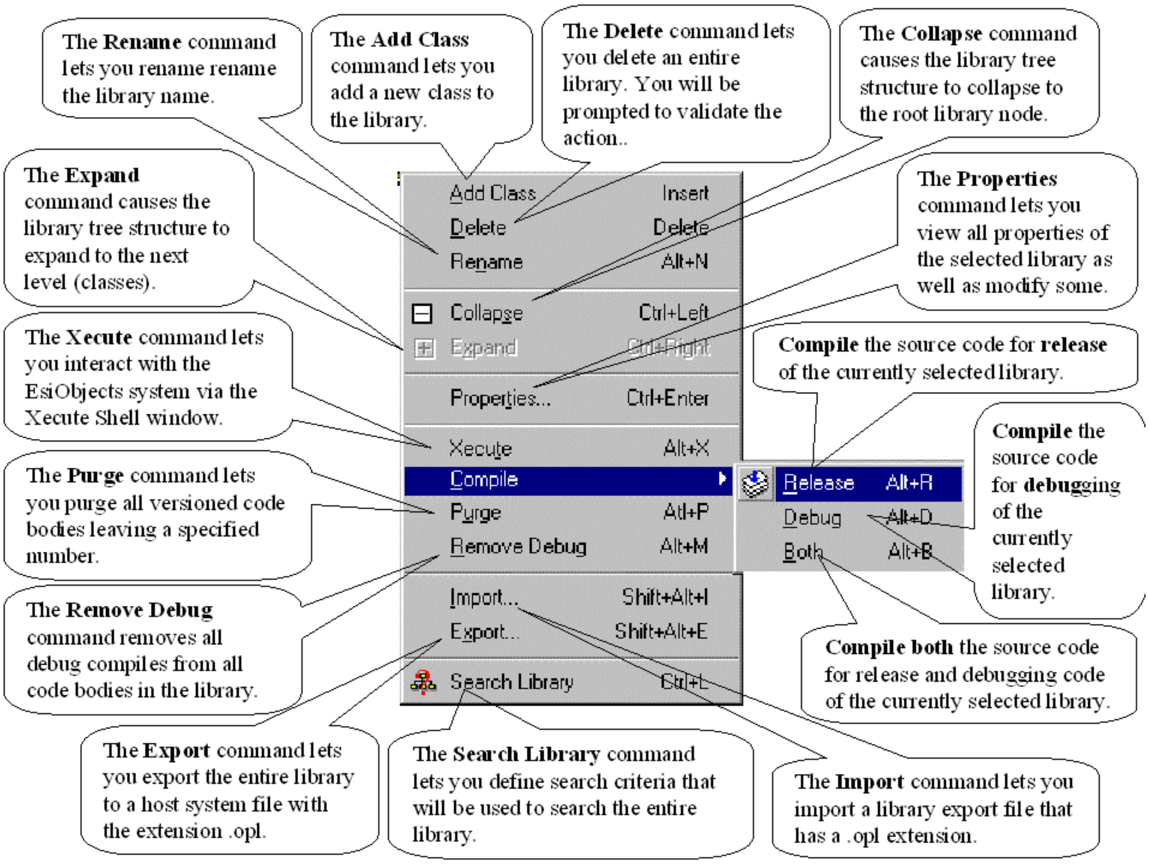
Clicking on the **Promote** button will copy the currently selected concrete service to it's superclass.

Clicking on the Release Compile button causes the selected library, class, interface or services to be compiled for release only.

Clicking on the **Collapse** button will collapse the selected sub-tree.

Library Popup Menu

The Library popup menu is invoked inside the Session Browser's pane right clicking on the library name or icon or by pressing **Shift+F10** key combination or.



The **Rename** command lets you rename rename the library name.

The **Add Class** command lets you add a new class to the library.

The **Delete** command lets you delete an entire library. You will be prompted to validate the action..

The **Collapse** command causes the library tree structure to collapse to the root library node.

The **Expand** command causes the library tree structure to expand to the next level (classes).

The **Properties** command lets you view all properties of the selected library as well as modify some.

The **Xecute** command lets you interact with the EsiObjects system via the Xecute Shell window.

Compile the source code for **release** of the currently selected library.

Compile the source code for **debugging** of the currently selected library.

The **Purge** command lets you purge all versioned code bodies leaving a specified number.

The **Remove Debug** command removes all debug compiles from all code bodies in the library.

Compile both the source code for release and debugging code of the currently selected library.

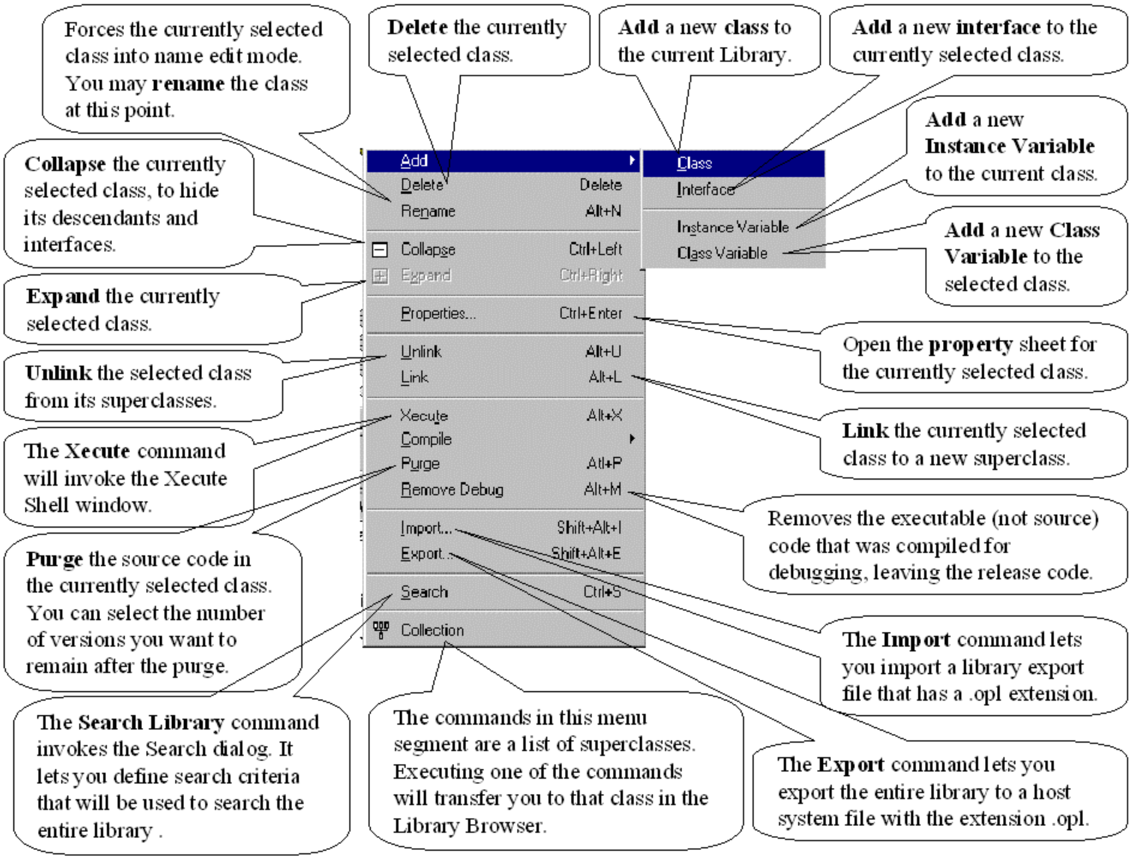
The **Export** command lets you export the entire library to a host system file with the extension .opl.

The **Search Library** command lets you define search criteria that will be used to search the entire library.

The **Import** command lets you import a library export file that has a .opl extension.

Class Popup Menu

The Class popup menu is invoked inside the Session Browser's pane by right clicking on a class or pressing **Shift+F10** with the class selected. The pictures below illustrate the functionality available on this popup menu.



Forces the currently selected class into name edit mode. You may **rename** the class at this point.

Delete the currently selected class.

Add a new **class** to the current Library.

Add a new **interface** to the currently selected class.

Add a new **Instance Variable** to the current class.

Add a new **Class Variable** to the selected class.

Collapse the currently selected class, to hide its descendants and interfaces.

Expand the currently selected class.

Open the **property** sheet for the currently selected class.

Unlink the selected class from its superclasses.

Link the currently selected class to a new superclass.

The **Xecute** command will invoke the Xecute Shell window.

Removes the executable (not source) code that was compiled for debugging, leaving the release code.

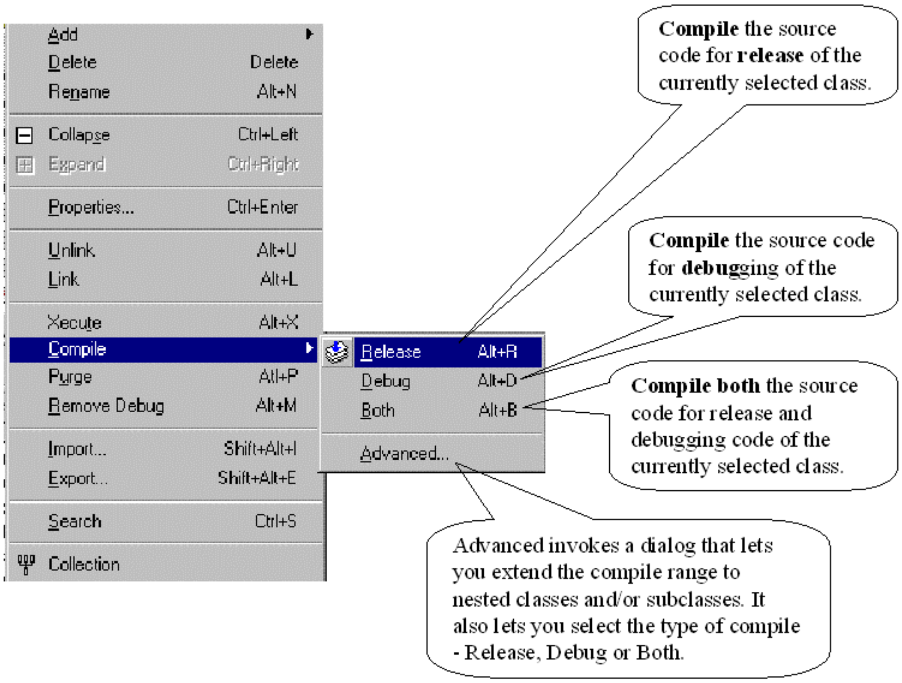
Purge the source code in the currently selected class. You can select the number of versions you want to remain after the purge.

The **Import** command lets you import a library export file that has a .opl extension.

The **Search Library** command invokes the Search dialog. It lets you define search criteria that will be used to search the entire library .

The commands in this menu segment are a list of superclasses. Executing one of the commands will transfer you to that class in the Library Browser.

The **Export** command lets you export the entire library to a host system file with the extension .opl.



Compile the source code for **release** of the currently selected class.

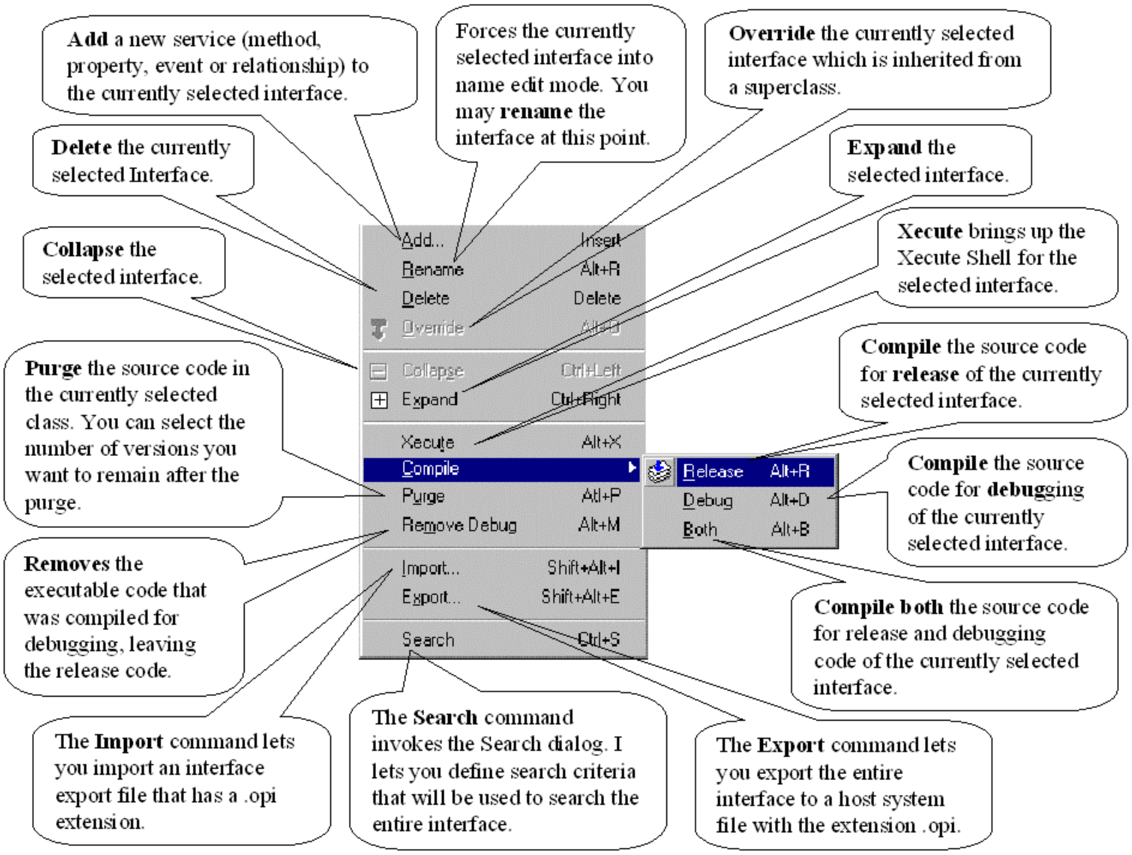
Compile the source code for **debugging** of the currently selected class.

Compile both the source code for release and debugging code of the currently selected class.

Advanced invokes a dialog that lets you extend the compile range to nested classes and/or subclasses. It also lets you select the type of compile - Release, Debug or Both.

Interface Popup Menu

The Interface popup menu is invoked inside the Session Browser's pane by right clicking on an interface or selecting the interface and pressing **Shift+F10**.



Add a new service (method, property, event or relationship) to the currently selected interface.

Forces the currently selected interface into name edit mode. You may **rename** the interface at this point.

Override the currently selected interface which is inherited from a superclass.

Delete the currently selected Interface.

Expand the selected interface.

Collapse the selected interface.

Xecute brings up the Xecute Shell for the selected interface.

Purge the source code in the currently selected class. You can select the number of versions you want to remain after the purge.

Compile the source code for **release** of the currently selected interface.

Compile the source code for **debugging** of the currently selected interface.

Removes the executable code that was compiled for debugging, leaving the release code.

Compile both the source code for release and debugging code of the currently selected interface.

The **Import** command lets you import an interface export file that has a .opi extension.

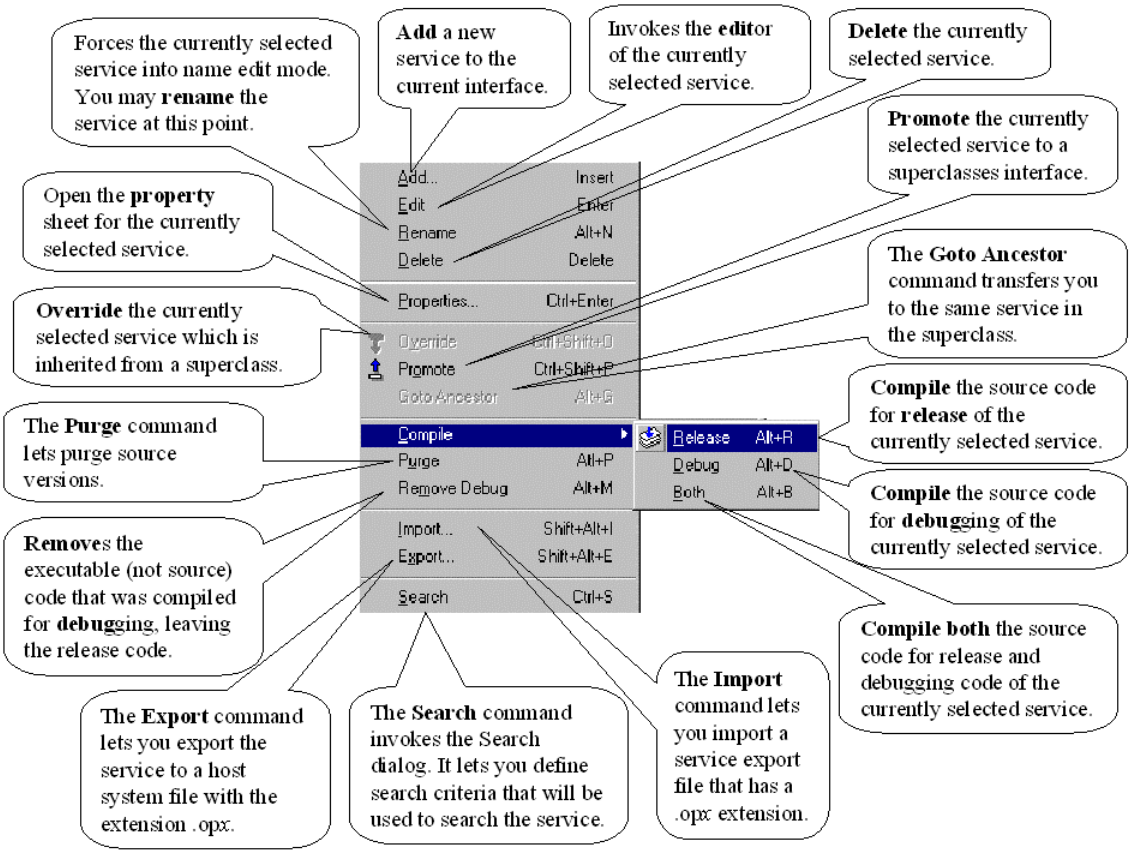
The **Search** command invokes the Search dialog. I lets you define search criteria that will be used to search the entire interface.

The **Export** command lets you export the entire interface to a host system file with the extension .opi.

Service Popup Menu

The Service popup menu is invoked inside the Session Browser's tree by right clicking on any property, method, event or relationship in the interface or pressing **Shift+F10** with the service selected.

Based on what service is selected, certain menu commands will be highlighted and others will be grayed. However, all the functionality is the same no matter what type of object is selected.



Forces the currently selected service into name edit mode. You may **rename** the service at this point.

Add a new service to the current interface.

Invokes the **editor** of the currently selected service.

Delete the currently selected service.

Promote the currently selected service to a superclasses interface.

Open the **property** sheet for the currently selected service.

Override the currently selected service which is inherited from a superclass.

The **Goto Ancestor** command transfers you to the same service in the superclass.

Compile the source code for **release** of the currently selected service.

The **Purge** command lets purge source versions.

Compile the source code for **debugging** of the currently selected service.

Removes the executable (not source) code that was compiled for **debugging**, leaving the release code.

Compile both the source code for release and debugging code of the currently selected service.

The **Export** command lets you export the service to a host system file with the extension .opx.

The **Search** command invokes the Search dialog. It lets you define search criteria that will be used to search the service.

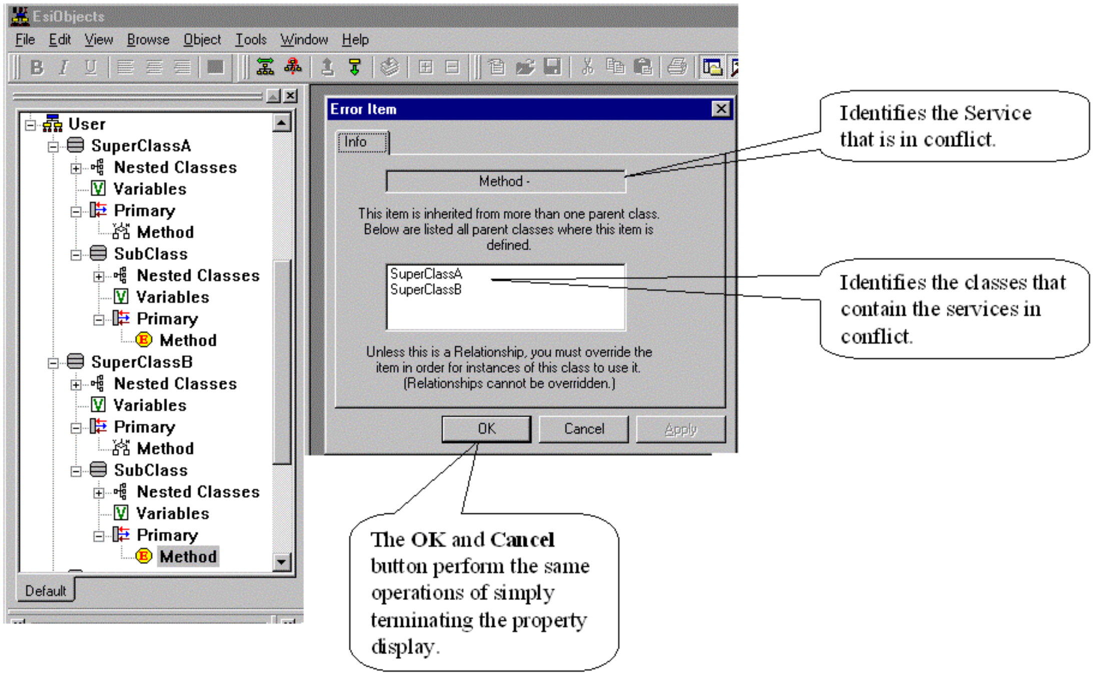
The **Import** command lets you import a service export file that has a .opx extension.

Multiple Inheritance Conflict

In the Session Browser, when a subclass inherits the same service from two separate classes, this is known as a multiple inheritance conflict. An error results from this and is indicated by an “E” with a circle around it (See the picture below).

If the user right clicks on the service and selects Properties from the pop up menu, a window appears describing the problem or conflict. The window includes the service in conflict, a description of the problem and the involved classes.

To rectify the situation, you can override the service that is in conflict.



This item is inherited from more than one parent class. Below are listed all parent classes where this item is defined.

Unless this is a Relationship, you must override the item in order for instances of this class to use it. (Relationships cannot be overridden.)

Identifies the Service that is in conflict.

Identifies the classes that contain the services in conflict.

The **OK** and **Cancel** button perform the same operations of simply terminating the property display.

Using the Session Browser

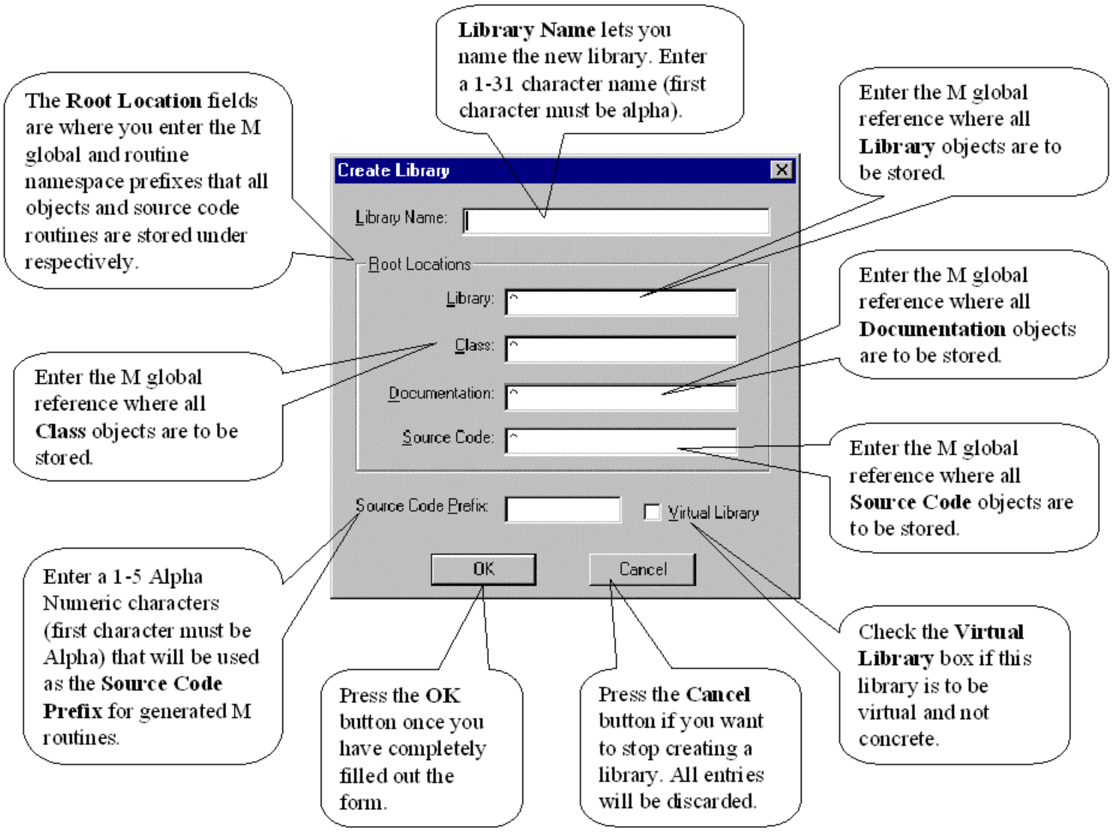
In addition to migrating a library or folder structure, the Session Browser lets you perform numerous operations within the structure. The operations you can perform are defined by the popup of main menu for the object selected within the structure. The following sections will describe the operations you can perform on these two structures.

Library Operations

Creating a New Library

This task is performed from the EsiObjects CDE Main Window.

1. Execute the **File|New|Library** command. The following dialog appears:



The **Root Location** fields are where you enter the M global and routine namespace prefixes that all objects and source code routines are stored under respectively.

Library Name lets you name the new library. Enter a 1-31 character name (first character must be alpha).

Enter the M global reference where all **Library** objects are to be stored.

Enter the M global reference where all **Documentation** objects are to be stored.

Enter the M global reference where all **Class** objects are to be stored.

Enter the M global reference where all **Source Code** objects are to be stored.

Enter a 1-5 Alpha Numeric characters (first character must be Alpha) that will be used as the **Source Code Prefix** for generated M routines.

Press the **OK** button once you have completely filled out the form.

Press the **Cancel** button if you want to stop creating a library. All entries will be discarded.

Check the **Virtual Library** box if this library is to be virtual and not concrete.

2. Enter the **Name of the Library**. Library names, like many other EsiObjects names, are from 1 to 31 characters long. The characters may be any combination of letters and numbers, except that the first character must be a letter.

3. Enter an M global location for the **Library**’s objects. This should be distinct from the other global locations in the dialog. Global locations can contain literal subscripts. For performance reasons, it is recommended that all locations are stored within the same global name, using distinct first level subscript name. For example ^CUST("Library"), ^CUST("Doc"), etc.

4. Enter an M global location for the **Classes** in the Library. This should be distinct from the other global locations in the dialog.

5. Enter an M global location for the Library's **Documentation** objects. This should be distinct from the other global locations in the dialog.

6. Enter an M global location for the Library's **Source Code**. This should be distinct from the other global locations in the dialog.

7. Enter a **Source Code Prefix** to be used in naming all the M routines compiled down from EsiObjects Source Code in the Library.

8. Press the **OK** button to create the library or press the **Cancel** button to discard all entries and avoid creating a new library.

9. EsiObjects then opens an empty Library Browser containing the name of the new library in its title bar.

Source Code Prefixes—How Long?

The Source Code Prefix is a **routine name** prefix for the automatically generated M routines in the library. ANSI Standard M routine names can be up to **8** characters long, and there are **62** legal characters for each position (0-9, A-Z, a-z) after the first character. The following table summarizes the number of possible routines, based on the number of characters in the prefix you specify:

Chars	Routines Possible	
1	3,521,614,606,208	trillions
2	56,800,235,584	billions
3	916,132,832	
4	14,776,336	millions
5	238,328	hundreds of thousands
6	3,844	
7	62	

Note: If you have two projects whose source code prefixes accidentally overlap - it will NOT cause problems. EsiObjects checks to see whether a routine name has been used, before using it for a generated routine name. If the routine name has already been used, then EsiObjects simply tries the next possible name. But source code prefixes should be helpful, making it obvious which routines belong to a given library.

Source Code Prefixes should be *long enough* to reliably prevent conflicts between the Library's automatically generated routines and all other routines on the system. However, they should be *short enough* to guarantee a sufficient number of routine names. A **four** or **five** character name, if it is unlikely to be used anywhere else, is usually safest.

Note: ESI Technology Corporation reserves the source code prefix VES. Do not use this prefix for your source code. Likewise, do not use a one-character prefix of V or a two-character prefix of VE since these prefixes may also generate conflicting routine names.

Same or Different Globals?

There is no restriction on overlapping global locations: it's possible to store all the source code and documentation objects in the same global, for example. However, you may experience reduced runtime performance when Class/Library information gets mixed in with Source/Documentation objects.

One approach that makes sense is to assign different subscript locations within the same global to the various components of a project.

Deleting a Library

When deleting a library you need to be concerned about types of objects:

- Definitional
- Instance

A library object knows where its **definitional** objects are stored. This is the information you give it when you create a new library. Every time you create a new class, interface, service of a class or any documentation objects associated with these components, they are automatically stored at the global roots specified at library creation time. Any M routines generated as a result of compiling a code body are also mapped to the namespace specified at library creation time.

However, **instance** objects are another matter. Instances will either be created as shared or not shared (persistent or non-persistent). Non-persistent objects only have a lifetime as long as the EsiObjects environment they were created in, consequently, we are not concerned with them in the context of this discussion. Persistent objects, however, live until explicitly deleted. These objects can be mapped to any location at creation time. The EsiObjects **CREATE** command contains a **Base** keyword that lets you map instances of a class to any M global root location (See the CREATE command in the EsiObjects Language Reference Guide for a detailed description of the command). If you do not specify a base location, EsiObjects will default the object to **^VESoshob**.

Mapping instances to M global locations is totally up to you, consequently, you must keep a record of what M global root locations these objects are stored under. It is up to you to maintain the instance global nodes and their subscripts.

Follow the instructions below to delete a library.

1. Connect to the session that contains the library you want to delete.
2. Select the library you want to delete by clicking on its name or icon. (If the Session Browser is not visible, make it visible by clicking on the Session Browser Tool Bar button).
3. Execute the **Edit|Delete** command. You will be prompted with a warning. If you want to proceed click on the **Yes** button.

At this point the library will be deleted. You will have to delete the instances manually by killing the globals they live in. This can be done through the Xecute Shell.

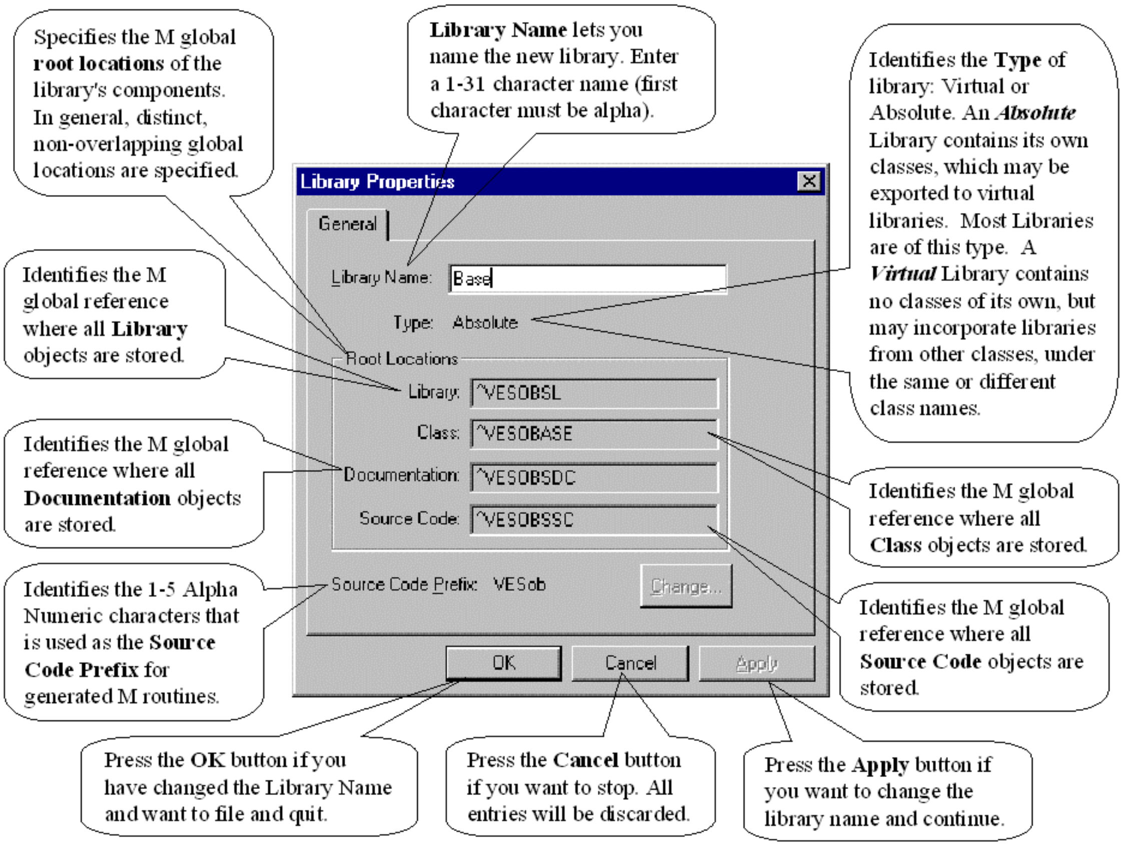
Examining Library Properties

To invoke the Library Property Sheet, follow these instructions:

1) Select the library by clicking on the library name or icon in the Library Browser.

2) From the **Object** menu, select **Properties**.

The Library Property Sheet dialog contains only one tab, for the Library's General properties.



Library Name: Base

Type: Absolute

Library: ^VESOBSL

Class: ^VESOBASE

Documentation: ^VESOBSDC

Source Code: ^VESOBSSC

Source Code Prefix: VESob

Specifies the M global **root locations** of the library's components. In general, distinct, non-overlapping global locations are specified.

Library Name lets you name the new library. Enter a 1-31 character name (first character must be alpha).

Identifies the **Type** of library: Virtual or Absolute. An ***Absolute*** Library contains its own classes, which may be exported to virtual libraries. Most Libraries are of this type. A ***Virtual*** Library contains no classes of its own, but may incorporate libraries from other classes, under the same or different class names.

Identifies the M global reference where all **Library** objects are stored.

Identifies the M global reference where all **Documentation** objects are stored.

Identifies the M global reference where all **Class** objects are stored.

Identifies the 1-5 Alpha Numeric characters that is used as the **Source Code Prefix** for generated M routines.

Identifies the M global reference where all **Source Code** objects are stored.

Press the **OK** button if you have changed the Library Name and want to file and quit.

Press the **Cancel** button if you want to stop. All entries will be discarded.

Press the **Apply** button if you want to change the library name and continue.

Class Operations

Creating Classes

Before you can create a new class, do the following:

1. Connect to the session that contains the library you want to add a class to.
2. Keep in mind that the Session Browser may not be visible. If this is the case, click on the Session Browser button of the System Toolbar to make it visible.

Creating a Root Class

If you wish to add a new class to the library, without linking that class to any other class, follow the steps below.

1. Either right click on the library name or icon and select the **Add|Class** command from the popup menu or select the library and execute the **File|New|Class** command of the Main Menu (or strike the Insert key and choose from the dialog).
2. The Create Class dialog appears. Enter the name of the class. Valid class names are alphanumeric, with the first character being an alpha only. Class names can be up to 32 characters in length and must be unique within the library.
3. Select the class type: Concrete, Abstract, or Mix-in. The choice here does not affect the behavior of the class, only the icon used to display the class in the tree panel.

4. Select **OK** to create the class. The new class is displayed in the tree panel as a *top-level class* directly below the library icon, in other words, with no super or subclasses.

Creating a Subclass

If you wish to add a new class as a subclass, follow these instructions

1. Either right click on the parent class name or icon and select the **Add|Class** command from the popup menu or select the parent class and execute the **File|New|Class** command of the Main Menu (or strike the Insert key and choose from the dialog).
2. The Create Class dialog appears. Enter the name of the class. Valid class names are alphanumeric, with the first character being an alpha only. Class names can be up to 32 characters in length and must be unique within the library.
3. Select the class type either Concrete or Abstract depending on where the class is in the hierarchy. If it is a terminal class (bottom of the hierarchy) it is Concrete. If it resides above the bottom class, it is Abstract. The choice here does not affect the behavior of the class, only the icon used to display the class in the tree panel.
4. Select **OK** to create the class. The new class is displayed in the tree panel directly below the selected parent class.

Creating a Nested Class

If you want to create a Nested Class, follow the steps below. Nested Classes are namespaced within a parent class. The parent class can be an Abstract, Concrete or Mix-in class.

1. Either right click on the Nested Classes name or icon and select the **Add|Class** command from the popup menu or select the Nested Classes by clicking on its icon and execute the **File|New|Class** command of the Main Menu.
2. The Create Nested Class dialog appears. Enter the name of the class. Valid class names are alphanumeric, with the first character being an alpha only. Class names can be up to 32 characters in length and must be unique within the library.
3. Select the class type either Concrete or Abstract depending on where the class is in the hierarchy. If it is a terminal class (bottom of the nested hierarchy) it is Concrete. If it resides above the bottom nested class, it is Abstract. The choice here does not affect the behavior of the class, only the icon used to display the class in the tree panel.
4. Select **OK** to create the nested class. The new nested class is displayed in the tree panel directly below the selected parent class.

Editing Class Properties

Editing class properties applies to any type of class, whether it be a root, subclass or nested class.

Invoking the Class Property Sheet

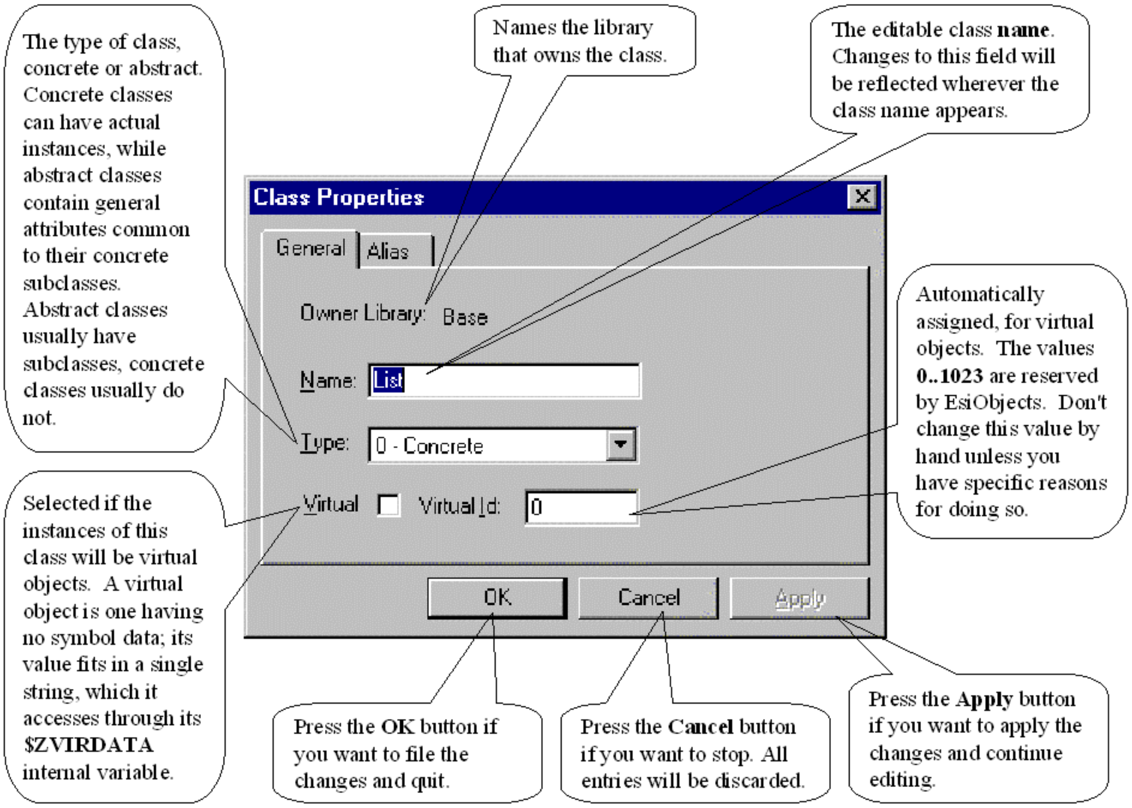
There are several ways to invoke the class property sheet:

- From the Session Browser, right click on a class and select the **Properties** command from the popup menu.

- From the Session Browser, select a class and press **Ctrl+Enter**.
- Select the class and execute the **Object|Properties** command of the main menu.

General

Class General Properties are invoked by clicking on the **General** tab of the Class Property Sheet. The General properties are presented when the property sheet dialog is initially opened.



Owner Library: Base

Name: List

Type: 0 - Concrete

Virtual □ Virtual Id: 0

The type of class, concrete or abstract. Concrete classes can have actual instances, while abstract classes contain general attributes common to their concrete subclasses. Abstract classes usually have subclasses, concrete classes usually do not.

Names the library that owns the class.

The editable class **name**. Changes to this field will be reflected wherever the class name appears.

Automatically assigned, for virtual objects. The values **0..1023** are reserved by EsiObjects. Don't change this value by hand unless you have specific reasons for doing so.

Selected if the instances of this class will be virtual objects. A virtual object is one having no symbol data; its value fits in a single string, which it accesses through its **$ZVIRDATA** internal variable.

Press the **OK** button if you want to file the changes and quit.

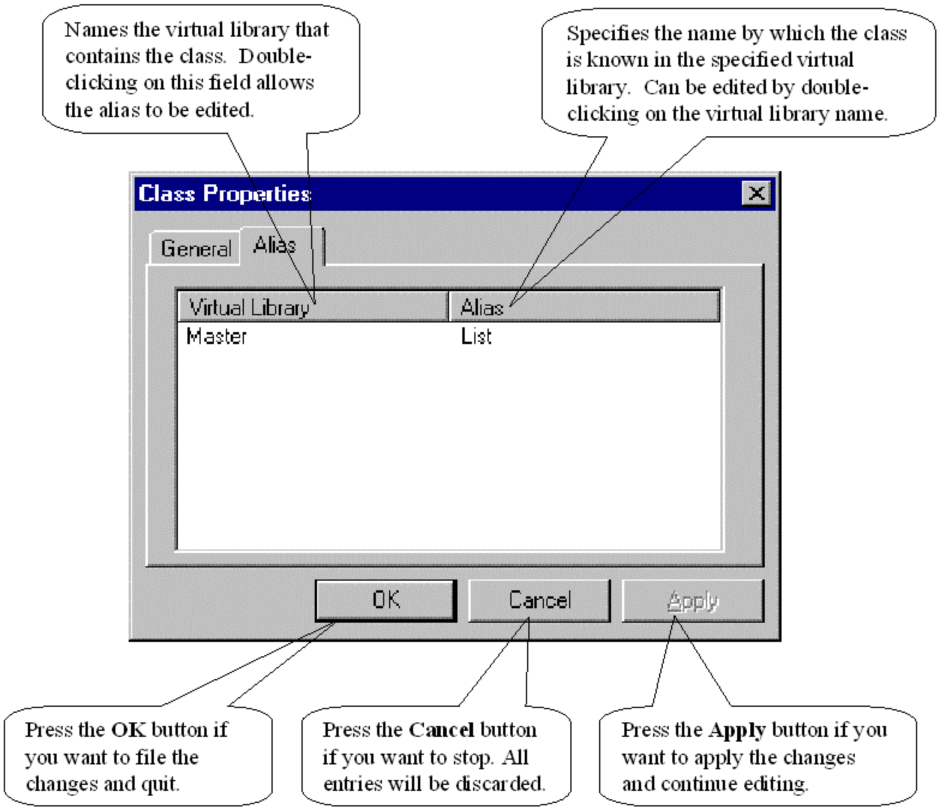
Press the **Cancel** button if you want to stop. All entries will be discarded.

Press the **Apply** button if you want to apply the changes and continue editing.

Alias

Aliases are used to map concrete class names to a virtual library under a different name. For example, assume you have a virtual library where you want to map two classes that have the same name from different libraries. Obviously you must change the name of one of them to prevent a namespace conflict within the virtual library. This can be accomplished by giving one of that classes an alias. Aliases are assigned to a virtual library through the class's property sheet.

Class Alias Properties are invoked by clicking on the **Alias** tab of the Class Property Sheet.



Virtual Library	Alias
Master	List

Names the virtual library that contains the class. Double-clicking on this field allows the alias to be edited.

Specifies the name by which the class is known in the specified virtual library. Can be edited by double-clicking on the virtual library name.

Press the **OK** button if you want to file the changes and quit.

Press the **Cancel** button if you want to stop. All entries will be discarded.

Press the **Apply** button if you want to apply the changes and continue editing.

Linking Classes

If the class does not exist, go to the Creating Classes section and follow the instructions for Creating a Subclass. If you wish to have an existing class be a subclass of another, follow the steps below.

Linking Classes

To link two existing classes to each other in a super-subclass relationship, follow the steps below.

1) In the Session Browser, select the class that is going to be the subclass in the relationship.
2) Right click on the class name or icon and execute the **Link** command from the popup menu or execute the **Object|Link** command from the main menu. You can also select the class and press the **Alt+L** keys.
3) The Link to Superclass dialog appears.
4) Enter the name of the class to link as a superclass. If the class to link to is in another library, specify the class name in the full library format: **Libraryname$Classname**. For example, if you are linking to the class Array in the Base library, specify the name as follows: Base$Array. If you are linking a nested class to another nested class, you must use the special syntax **Libraryname$Classname>NestedClassname>…**.

5) Select the **OK** button to link the classes. The selected class becomes a subclass of the class specified in Step 4.

Breaking Class *Relationships*

If you wish to break the linkage between two classes or nested classes, follow the steps below. These instructions apply to normal and nested classes.

1) In the Library Browser, select the subclass (normal or nested) you wish to remove from the relationship.
2) Right click on the class name or icon and execute the **Unlink** command in the popup menu or execute the **Object|Unlink** command in the main menu. You can also select the class and press the **Alt+U** keys.
3) If there are multiple superclasses you are prompted for which superclass to unlink from. Select the superclass from the dialog.
4) The selected class and its descendants are automatically unlinked from the relationship and are no longer subclasses to the parent. Note that the selected class may disappear from the tree view if it was unlinked from another class in the same library and is now a top-level class.

Using Drag-and-Drop to Build the Hierarchy

You can build the class hierarchy using drag-and-drop. This allows for a much easier way to link classes than using menu options.

- In the Session Browser, select a class you wish to link to another either as a subclass or a superclass.
- Holding the left mouse button down on the class name or icon, drag the mouse cursor to the class name you wish to link too.
- If you need to scroll down or up to get to the class name to link to, drag the cursor just below or above the tree panel. Doing so will cause the tree to scroll in that direction.
- If you need to navigate down to a subclass, drag the cursor over the superclass and press the **Shift** key. The display of the class will be expanded to show the subclasses.
- Once you have dragged the cursor over the desired class, lift the left mouse button to drop the class. You will be prompted to link the dropped class as subclass or a superclass.
- Selecting **Superclass** will make the dragged class the superclass. Selecting **Subclass** will make the dragged class the subclass.

Promotion and Generalization

Promotion and **Generalization** are two related class-restructuring operations that cause a service (event, relationship, method or property) to be moved up "higher" in the class structure. These operations apply to normal and nested classes.

- In **Promotion**, the service is simply moved up to the superclass without changes. Generally (but not always) you should delete from the subclass unless you want to override the service. The **Promote** command on the service popup menu, the Main Menu **Object|Promote** command or the Class Toolbar button ⬆ moves a service up to the superclass.
- In **Generalization**, the service is moved up to the superclass and modified (or rewritten entirely) in a ***more general*** way to accommodate a variety of possible subclasses. Generally it is ***not*** deleted from the subclass. It is merely moved up the class tree to the proper level of abstraction so that other subclasses can inherit the service.

All services can also be moved up to the superclass by using drag-and-drop.

If you need to delete the subclass service, then simply select it and press the **Delete** key (or use the **Delete** command on the popup menu).

Demotion and Specialization

Demotion and Specialization are two related class-restructuring operations that cause a service to be moved down "lower" in the class structure. These operations apply to normal or nested classes. You can demote a service by

1. Executing the **Override** command in the service popup menu.
2. Executing the **Object|Override** command of the Main menu.
3. Clicking on the Override button of the Class Toolbar.

Executing any one of these commands will move a method, property, event or relationship down from the superclass from which it is inherited.

- In **Demotion**, the service is simply copied down to the subclass without changes. The service remains in the superclass. You may want to keep it there for further specialization (see next bullet) or you may want to delete it and use the inherited service.
- In **Specialization**, the service is moved down to the subclass and modified (or rewritten entirely) in a ***more specific*** way to accommodate the specific needs or features of the subclass. Generally it is ***not*** deleted from the subclass.

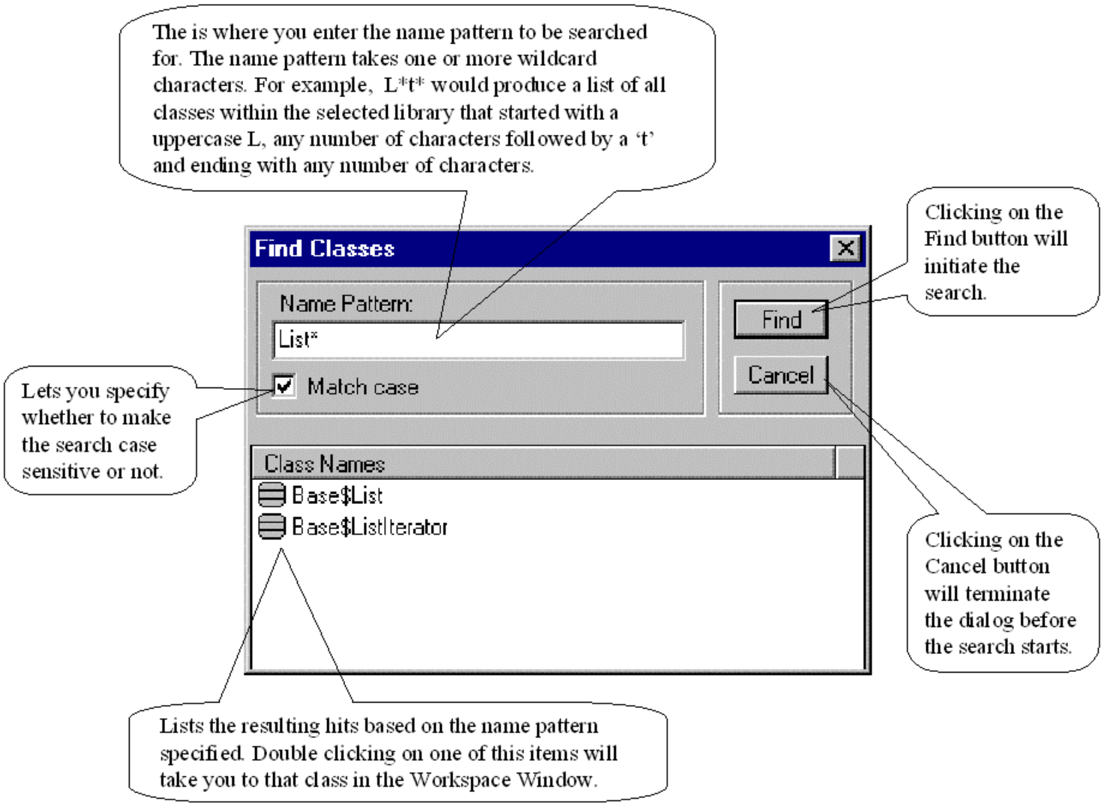
Services can also be moved down to the subclass by using drag-and-drop.

If you need to delete the superclass service, then simply select it and press **Delete** (or execute the Delete command in the popup or main menu.)

Finding a Class in the Hierarchy

Large application libraries as well as the ESI and Base libraries contain a large number of classes. Often you will know where the class resides in the library hierarchy. However, often you won't. To help you access a class quickly, EsiObjects provides a means by which you can go to a class directly if you know the full or partial name of the class. To find a class, follow the steps below.

1) To go to a class directly, click on the **GoTo Class** button that is on the Class Toolbar, press the **Ctrl G** key combination or execute the **Tools|Search|Goto Class** command on the main menu. A Find Class dialog box will display.
2) Specify a path name in the form of **Libraryname$Classname** or **Libraryname$Classname>NestedClassname>…** .
3) Click on the Find button to initiate the search. The dialog to expand if two or more hits are found. If only one hit is found, you will be transferred directly to that library class. If the Class Name list is displayed, you may select the class you want to transfer to by double clicking on the class name you want.



Name Pattern: List*

☑ Match case

Base$List
Base$ListIterator

The is where you enter the name pattern to be searched for. The name pattern takes one or more wildcard characters. For example, L*t* would produce a list of all classes within the selected library that started with a uppercase L, any number of characters followed by a 't' and ending with any number of characters.

Clicking on the Find button will initiate the search.

Lets you specify whether to make the search case sensitive or not.

Clicking on the Cancel button will terminate the dialog before the search starts.

Lists the resulting hits based on the name pattern specified. Double clicking on one of this items will take you to that class in the Workspace Window.

Please note that the search is confined to the library of the currently selected object in the Session Browser. That is, if you have the Collection class selected in the Base library, the search will be confined to the Base library only.

Deleting Classes

Deleting a class is a straightforward Session Browser operation. Follow the steps below:

1) Select the class you want to delete.
2) Right click on the selected class and execute the **Delete** command in the popup menu (or the **Edit|Delete** command in the main menu or press the **Del** key)
3) A warning dialog will appear. Click on the **Yes** button if you want to delete the class from the library. If you do not, click the **No** button.

If you chose **Yes**, the class and all its interfaces and services will be deleted. The Library structure within the Session Browser will be updated.

Please Note: If the class had subclasses, they will not be deleted. They will become root classes.

Interface Operations

Interfaces are a means of partitioning class services into logical groupings. The Primary interface is the default interface and does not need to be specified in the object message structure. Other interfaces can be created at will. One other interface that has significance besides Primary is Factory. Factory, if defined, is reserved and may contain constructor and destructor methods that are automatically executed at object instantiation time.

Some benefits of being able to partition class services into separate interface are:

- Minimized the number of services the programmer has to view to find the one of interest, increasing productivity.
- Minimizes the number of services the User Interface must display, increasing response time.
- Offers the capability of adding security to the interface in the future.

Creating Interfaces

If you wish to add a new interface to the class, follow the steps below.

1) Either right click on the class name or icon and select the **Add|Interface** command from the popup menu or select the class and execute the **File|New|Interface** command of the main menu (or strike the Insert key and click on the Interface radio button in the dialog).
2) EsiObjects will automatically create a new interface giving it a name that begins with **Interface** and ends with a number. To change its name:
   a) Select it by clicking on the name and either
      i) Right click and execute the Rename command on the popup menu.
      ii) or, execute the Rename command on the main menu.
   b) or, simply click the second time to go into Rename mode directly.

Valid interface names are alphanumeric, with the first character being an alpha only. Interface names can be up to 32 characters in length and must be unique within the class.

Deleting Interfaces

Deleting interfaces is a straightforward Session Browser operation. Follow the steps below:

4) Select the interface you want to delete.
5) Right click on the selected interface and execute the Delete command in the popup menu (or the Delete command in the main menu or press the **Del** key)
6) A warning dialog will appear. Click on the Yes button if you want to delete the interface from the class. If you do not, click the No button.

If you chose Yes, the interface and all it's services will be deleted. The class structure within the Session Browser will be updated.

Variable Operations

Creating Variables

If you wish to add a new variable to the class, follow the steps below.

1) Left click on the Variable interface.
2) There are three ways to create a variable:
   a) Right click on the Variable interface and select the **Add** command from the popup menu
   b) or execute the **File|New|Instance Variable** or **File|New|Class Variable** command of the main menu
   c) or strike the **Insert** key.
3) In all cases you will be presented with a dialog box requesting the variable name. Enter the name of the variable.
4) Only in case 1b will the correct type be selected. Use the pull down list box to select the type of variable. EsiObjects supports Instance and Class variables.

Valid variable names are alphanumeric, with the first character being an alpha only. Names can be up to 32 characters in length and must be unique within the variable interface.

Deleting Variables

Deleting interfaces is a straightforward Session Browser operation. Follow the steps below:

1) Select the variable you want to delete.
2) There are three ways to delete the variable:
   a) Right click on the selected variable and execute the Delete command in the popup menu
   b) or execute the **Object|Delete** command in the main menu
   c) or press the **Del** key.

3) A warning dialog will appear. Click on the **Yes** button if you want to delete the variable from the interface. If you do not, click the **No** button.

If you chose **Yes**, the variable will be deleted. The structure within the Session Browser will be updated.

Modifying a Variable Declaration

By creating a variable in the Variable interface of a class, you have declared it to the compiler. EsiObjects contains a Variable Definition Editor that lets you specialize the declaration. To learn more about specializing a variable, refer to the Variable Definition Editor section of this guide.

Service Operations

Creating Services

Within an Interface, you may create one or more object services. They are, Methods, Properties, Events and Relationships. Follow the steps outlines below to add a service.

1) Select the interface by left clicking on it in the Session Browser.
2) There are three ways to create an interface service:
   a) Right click on the interface and select the **Add** command from the popup menu
   b) or execute the **File|New|Method**, **Property**, **Event** or **Relationship** command on the main menu
   c) or select the interface and then strike the **Insert** key.
3) In all cases you will be presented with a dialog box requesting the service name. Enter the name of the service.
4) Only in case 1b will the correct type be selected. Use the pull down list box to select the type of service.

Valid service names are alphanumeric, with the first character being an alpha only. Names can be up to 32 characters in length and must be unique within the interface.

Deleting Services

Deleting a service is a straightforward Session Browser operation. Follow the steps below:

1) Select the service you want to delete.
2) There are three ways to delete the service:
   a) Right click on the selected service and execute the Delete command in the popup menu
   b) or execute the **Object|Delete** command in the main menu
   c) or press the **Del** key.

3) A warning dialog will appear. Click on the **Yes** button if you want to delete the service from the interface. If you do not, click the **No** button.

If you choose **Yes**, the service will be deleted. The structure within the Session Browser will be updated.

Modifying a Service

EsiObjects contains an editor for methods, properties and events. It contains a wizard for relationships. The description and use of these editors are described in the following sections:

- for Methods, refer to the Method Editor section.
- for Properties, refer to the Property Editor section.
- for Events, refer to the Event Template Editor section.
- and for Relationships, refer to the Relationship Wizard section.

Synchronizing the Tree Selection

Often when editing a service of a class, you will be browsing in the tree structure of one of the libraries that is a part of the current session. You will most likely want to synchronize the tree selection in the Session Browser with the currently active service editor. You can do this by using the **Find in Tree** function. To use the Find in Tree function, follow these steps:

1. Select the editor for the service you want to synchronize.
2. Execute the **View|Find in Tree** command or, if the Class Toolbar is visible, click on the **Find in Tree** button.

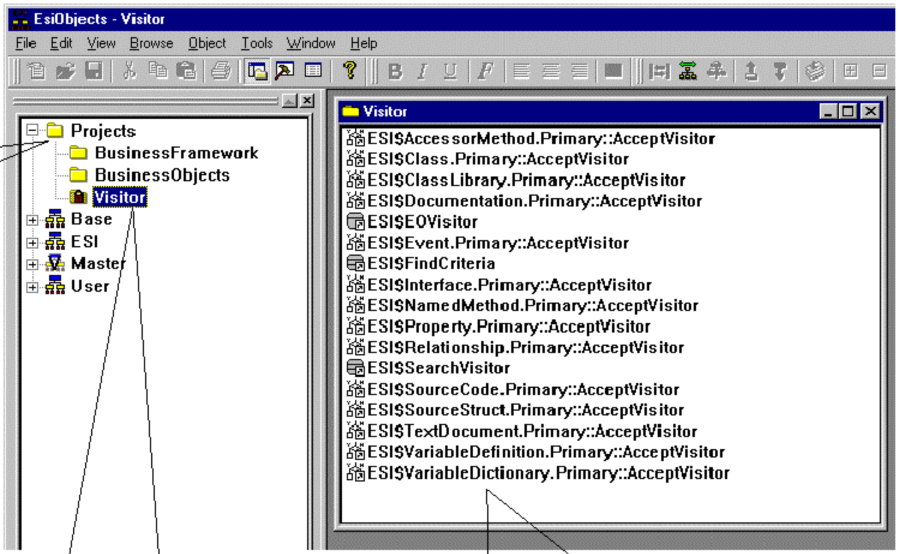
The tree in the Session Browser will readjust and expand if necessary. The service associated with the selected editor will be selected and highlighted.

Folder Operations

Folders are folders in the traditional Windows sense. Folders can be organized into hierarchical structures. Each folder may have subfolders. Within each folder, you can store library, class, interface or service objects that exist in any library structure. All menu operations that are normally available to the object in the library are available to you through the folders. Folders provide an indirect means of accessing all the objects in a library structure.

Folders are used to store objects that you are currently working on. They can also be used to store disparate objects that you want to export as a unit. Folders are general-purpose objects that can be used for a number of different reasons.

The picture below illustrates a typical folder structure in the Session Browser and it’s Folder Content window in the client area.



Projects
- BusinessFramework
- BusinessObjects
- Visitor

Base
ESI
Master
User

ESI$AccessorMethod.Primary::AcceptVisitor
ESI$Class.Primary::AcceptVisitor
ESI$ClassLibrary.Primary::AcceptVisitor
ESI$Documentation.Primary::AcceptVisitor
ESI$EOVisitor
ESI$Event.Primary::AcceptVisitor
ESI$FindCriteria
ESI$Interface.Primary::AcceptVisitor
ESI$NamedMethod.Primary::AcceptVisitor
ESI$Property.Primary::AcceptVisitor
ESI$Relationship.Primary::AcceptVisitor
ESI$SearchVisitor
ESI$SourceCode.Primary::AcceptVisitor
ESI$SourceStruct.Primary::AcceptVisitor
ESI$TextDocument.Primary::AcceptVisitor
ESI$VariableDefinition.Primary::AcceptVisitor
ESI$VariableDictionary.Primary::AcceptVisitor

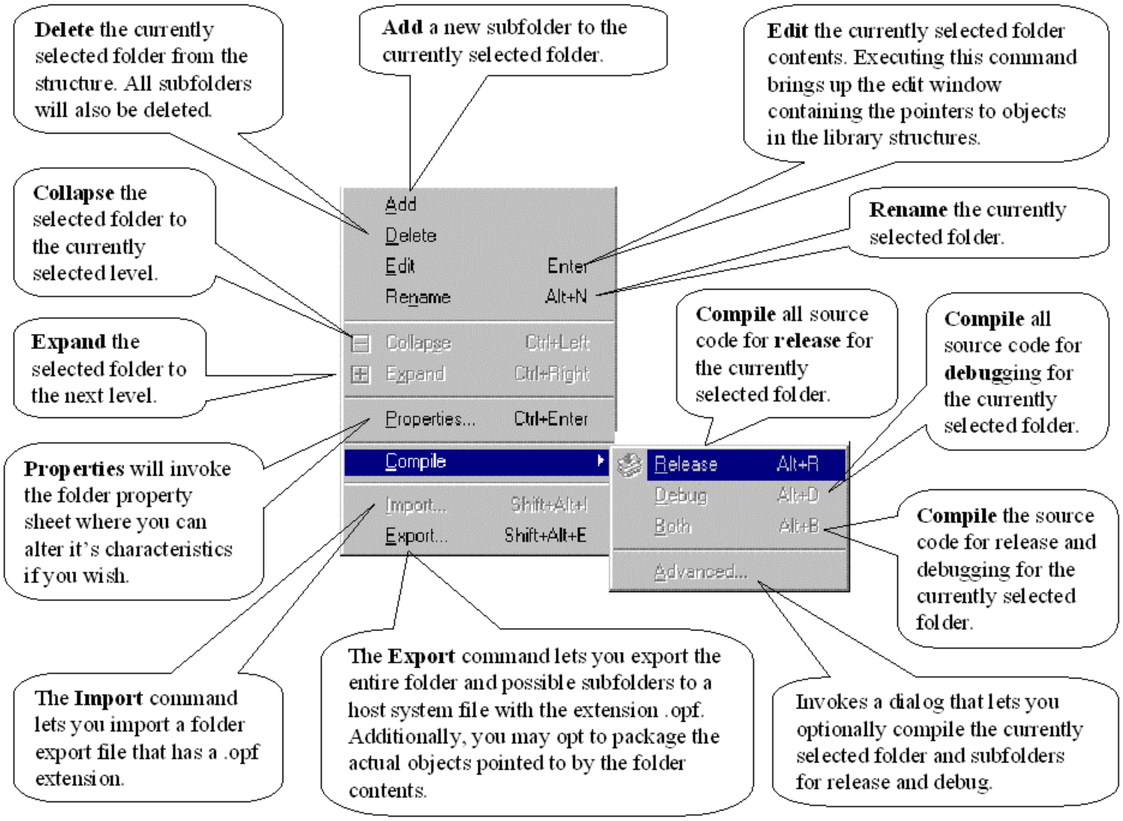
Folders can be organized into hierarchies.

Folders may contain any session object available in the library structure. Folders can be used to store objects being currently worked on or objects that you want to export. They can be used for any purpose.

Folders contain objects. These objects can be viewed via a window. All the objects outlines above are a part of a project within the ESI library to implement the Visitor pattern used by the EsiObjects Search Tool. Operations can be performed indirectly on each object.

Folder Structure Operations

The Session Browser is used to perform folder structure operations. The operations can be accessed through a popup menu. The following illustrates the popup menu that appears as a result of right clicking on a folder name or icon.



Delete the currently selected folder from the structure. All subfolders will also be deleted.

Add a new subfolder to the currently selected folder.

Edit the currently selected folder contents. Executing this command brings up the edit window containing the pointers to objects in the library structures.

Collapse the selected folder to the currently selected level.

Rename the currently selected folder.

Expand the selected folder to the next level.

Compile all source code for **release** for the currently selected folder.

Compile all source code for **debugging** for the currently selected folder.

Properties will invoke the folder property sheet where you can alter it's characteristics if you wish.

Compile the source code for release and debugging for the currently selected folder.

The **Import** command lets you import a folder export file that has a .opf extension.

The **Export** command lets you export the entire folder and possible subfolders to a host system file with the extension .opf. Additionally, you may opt to package the actual objects pointed to by the folder contents.

Invokes a dialog that lets you optionally compile the currently selected folder and subfolders for release and debug.

Most of the operations are the same operations that apply to the library structure. The Add, Delete and Edit operations will be explained here.

Creating a New Folder

If you wish to add a new folder to the class, follow the steps below.

1) First determine where you want the folder. Select an existing folder if you want the new folder to be a subfolder. If you want it to be a root (top level) folder, select any object in a library.
2) To invoke the New folder dialog, do one of the following:
    a) Right click on the selected item and execute the **Add** command.
    b) Execute the **File|New|Folder** command in the main menu.
3) Enter the name of the folder in the Create Folder Name field. Pull down the Type combo-box field and select the type of folder. Two types of folders are available: Common and Private. Common is shared among all programmers sighed into the session. Private is private to you

and not shared. Private is based on you current initials. The two types are available under the following conditions:

a) Common and Private only when the folder is a root or subfolder to another Common folder.

b) Private only when it is a subfolder to another Private folder.

Click on the **OK** button to create the folder.

Valid folder names are alphanumeric, with the first character being an alpha only. Folder names can be up to 32 characters in length.

Deleting a Folder from the Structure

Deleting a folder is a straightforward Session Browser operation. Follow the steps below:

1) Select the folder you want to delete.
2) Right click on the selected folder and execute the **Delete** command in the popup menu (or the **Edit|Delete** command in the main menu or press the **Del** key)
3) A warning dialog will appear. Click on the **Yes** button if you want to delete the folder. If you do not, click the **No** button.

If you chose **Yes**, the folder and all its contents will be deleted. The Folder structure within the Session Browser will be updated.

Note: If the folder has subfolders, they will be deleted also!

Moving a Folder and its Content to another Folder

By default, all drag and drop operations on a folder are deep copy operations. That is, if you drag a folder onto another folder and drop it, only the contents will be transferred. The actual folder structure will remain unaltered. All pointers from the folder being drug and dropped will be merged with the contents of the target folder. However, the source folder will be unaltered.

If you want to move the folder and its contents, there are two ways to do this.

1) Left click on the folder and drag it to the folder you want to move it to as a subfolder. Press the **Ctrl** key down and then drop the object. The folder will become a subfolder of folder you dropped it on and it will be removed from its original position in the tree.
2) Right click on the folder and drag it to the folder you want to move it to as a subfolder. Drop it. At this point a popup menu will appear giving you the option to **Copy**, **Move** or **Cancel**. Select the **Move**. The folder will be inserted as a subfolder and removed from its original location.

Copying a Folder’s Content to another Folder

As discussed under the Move operation, all drag and drop operations on a folder structure are by default deep copy operations. The copy operation only moves the content of the folder to another folder. There are two ways of copying the contents of a folder.

1) Left click on the folder and drag it to the folder you want to move it to as a subfolder. Drop it. The folder contents, if any, will be copied to the target folder. The source folder will remain in its original position and the contents will be unaltered.
2) Right click on the folder and drag it to the folder you want to move it to as a subfolder. Drop it. At this point a popup menu will appear giving you the option to **Copy**, **Move** or **Cancel**. Select the **Copy**. The source folder contents will be copied to the target folder and the original folder will remain unaltered.

Invoking the Folder’s Content Editor

Each folder in the structure may or may not have content. You can invoke the Folder Editor by simply double clicking on it or alternatively, performing the following steps:

1) Select the folder you want to edit.
2) Right click on the selected folder and execute the **Edit** command in the popup menu (or the **Object|Edit** command in the main menu or press the **Enter** key)
3) A folder content editor window will appear. It will display the pointers to the objects in the library structures if those objects were dragged into the edit window.

Refer to the Section Folder Content Editor for information on how to use the editor.

Finding Library Objects and Folders

As a programmer, you generally remember class names, or at least, part of the name. EsiObjects contains a structural search feature called **GoTo Class.** This feature is a part of the Session Browser window. This function can be invoked by first selecting the library (or an object in the library) and then pressing the **Ctrl+G** keys or executing the **View|Toolbars|Class** command of the Main Menu. A Find Class dialog will be displayed in the client area. You can enter a class name to search for. The name may contain the ‘*’ wildcard character. If it does not contain the character, the name will be searched for literally.

For example:

- If the class name ‘Collect’ is entered, the system will search specifically for a class called ‘Collect’.
- If you enter the class name ‘Collect*’, the system will look for a class name that begins with the characters ‘Collect’ and ends with any other valid name characters. If one class is found with a name fitting this pattern, the system will automatically go to that class in the selected Session Browser, opening up the tree structure at that point. However, if the system finds more than one class matching this pattern, you will be

presented with a list of the hits. Selecting the class you want will then prompt the system to go to that class and open up the library structure.

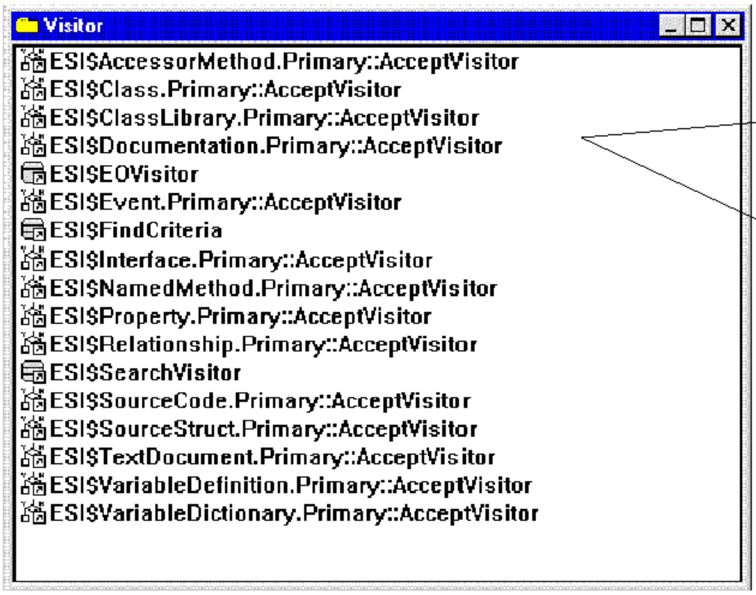
- If you enter a class name with more that one wildcard characters, the system will look for that pattern. If you enter a '*Collect*' for example, the system will search for class names that begin with any valid characters and end with any valid characters having the literal characters 'Collect' anywhere in between.

Folder Content Editor

Associated with a folder in the folder structure is its content. The content can be viewed by invoking the Folder Content Editor.

Folder Content Editor Explained

The Folder Content Editor is a window that contains object pointers to the actual objects in the library structures. The content window may contain any object in a library structure: library, class, interface or service (method, property, relationship or event) as well as subfolders. Since all objects in the folder window are pointers to object in the library structure, any menu operation performed on a selected object is actually performed on the target object.

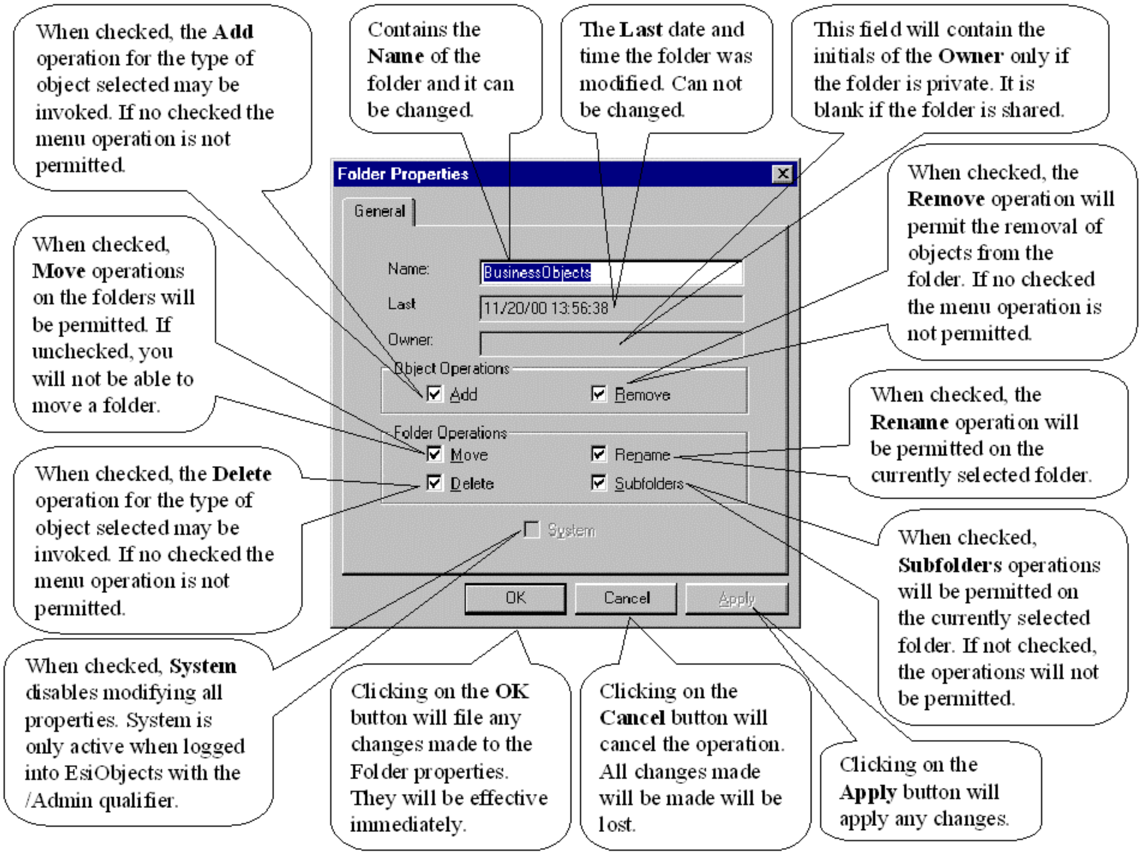


ESI$AccessorMethod.Primary::AcceptVisitor
ESI$Class.Primary::AcceptVisitor
ESI$ClassLibrary.Primary::AcceptVisitor
ESI$Documentation.Primary::AcceptVisitor
ESI$EOVisitor
ESI$Event.Primary::AcceptVisitor
ESI$FindCriteria
ESI$Interface.Primary::AcceptVisitor
ESI$NamedMethod.Primary::AcceptVisitor
ESI$Property.Primary::AcceptVisitor
ESI$Relationship.Primary::AcceptVisitor
ESI$SearchVisitor
ESI$SourceCode.Primary::AcceptVisitor
ESI$SourceStruct.Primary::AcceptVisitor
ESI$TextDocument.Primary::AcceptVisitor
ESI$VariableDefinition.Primary::AcceptVisitor
ESI$VariableDictionary.Primary::AcceptVisitor

The Folder Editor may contain a pointer to an object contained in any library structure that is a part of the session. The references shown in this window represent the actual object found in a session. All popup and main menu operations will be applied to the actual object with the exception of those operations specific to the window content.

Property Sheet

The picture below illustrates and describes the folder property sheet. You can change the characteristics of a folder by modifying its properties. Various check boxes exist for the purpose of permitting or inhibiting menu operations on the folder content objects or the folder itself.



Name: BusinessObjects
Last 11/20/00 13:56:38
Owner:

Object Operations
☑ Add ☑ Remove

Folder Operations
☑ Move ☑ Rename
☑ Delete ☑ Subfolders

□ System

When checked, the **Add** operation for the type of object selected may be invoked. If no checked the menu operation is not permitted.

Contains the **Name** of the folder and it can be changed.

The **Last** date and time the folder was modified. Can not be changed.

This field will contain the initials of the **Owner** only if the folder is private. It is blank if the folder is shared.

When checked, the **Remove** operation will permit the removal of objects from the folder. If no checked the menu operation is not permitted.

When checked, **Move** operations on the folders will be permitted. If unchecked, you will not be able to move a folder.

When checked, the **Rename** operation will be permitted on the currently selected folder.

When checked, the **Delete** operation for the type of object selected may be invoked. If no checked the menu operation is not permitted.

When checked, **Subfolders** operations will be permitted on the currently selected folder. If not checked, the operations will not be permitted.

When checked, **System** disables modifying all properties. System is only active when logged into EsiObjects with the /Admin qualifier.

Clicking on the **OK** button will file any changes made to the Folder properties. They will be effective immediately.

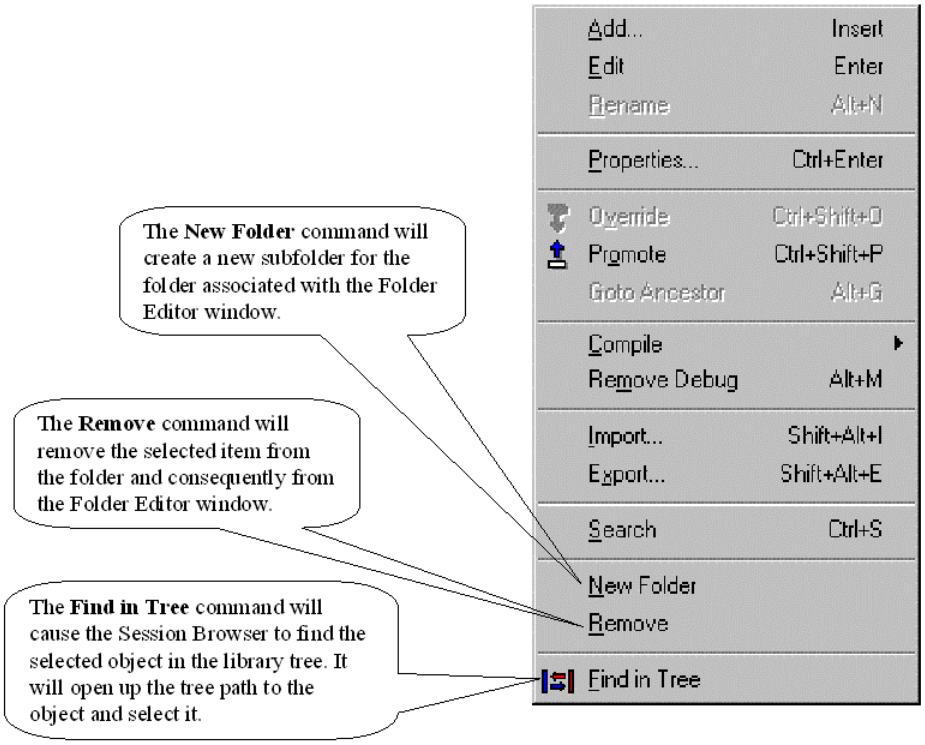
Clicking on the **Cancel** button will cancel the operation. All changes made will be made will be lost.

Clicking on the **Apply** button will apply any changes.

Popup Menu

Menu operations performed on each object in the Folder Editor window are the same operations performed on the object when directly accessed via a library structure. For example, the method popup menu (and main menu commands) will contain the same commands and, indeed, look the same. The same is true of any other object and its popup and main menu commands.

There is one exception to this rule however. Each popup menu will have two additional segments attached to the bottom of the common popup that will contain three commands. These commands are specific to the folders (See each of the appropriate sections in this guide for information about menu commands specific to the library object type.). The following illustrates the specific folder commands.



The **New Folder** command will create a new subfolder for the folder associated with the Folder Editor window.

The **Remove** command will remove the selected item from the folder and consequently from the Folder Editor window.

The **Find in Tree** command will cause the Session Browser to find the selected object in the library tree. It will open up the tree path to the object and select it.

Using the Folder Content Editor

Linking a Library Object to a Folder

You may move any object from a library structure (including the library itself) or folder to a target folder in two different ways.

1) Drag the object from a library or folder to the target folder using the left button of the mouse and drop it. The object will be placed in the folder.
2) Drag the object from a library or folder to the target folder using the right button of the mouse and drop it. A popup menu will appear with two commands: **Link** and **Cancel**. **Cancel** will let you abort the operations and **Link** will complete the operation. If **Link** is chosen, the object will be placed in the folder.

Removing a Library Object Pointer from a Folder

Removing a object pointer from the folder is simple.

1) Within the Session Browser, double click on the folder that contains the object pointer you want to remove.
2) Select the object pointer from the Folder Editor window.
3) Right click on the selected object pointer to invoke the popup menu.
4) Execute the **Remove** command.

Note: Performing this operation will remove the object pointer from the folder; it will not delete the actual object form the library structure.

Indirect Library Operations using the Folder Contents

As stated in the Popup Menu section, all objects in the folder are pointers to the actual object in a library structure that is a part of a session. Any operation you perform on a selected item via a popup (or main) menu will be applied to the object in the library structure.

Activating the Indirect Delete Command

Because the operations are indirect, it is possible to become confused and perform certain operations that have unintentional consequences. The **Delete** command is one of those operations. Because the **Delete** command can have undesirable consequences, it has been made a personal preference. That is, you will not see the **Delete** command unless that preference is enabled. See the Preference Tab Sheet subsection of the Using User Options section in the guide for more information.

Populating the Folder Content

Populating folders is accomplished using drag and drop. Objects can be drug in the following ways:

1. From a library structure and dropped onto a folder in the folder structure.
2. From a library structure and dropped onto a Folder Editor window.
3. Between Folder Editor windows.

Performing Operations on the Folder Content

Operations are performed on a folder object by selecting it and then using the popup menu or the appropriate main menu command.

Note: All operations performed via the Folder Content window are indirectly applied to the actual object in the library structure except the **New Folder**, **Remove** and **Find in Tree**.

Synchronizing the Library Object with the Folder Selection

Often when working out of a folder window, you may want to synchronize the definitional object pointed to with the actual object itself in the Session Browser. You can do this by using the **Find in Tree** function. To use the Find in Tree function, follow these steps:

1) Select the definitional object pointer in the folder you want to synchronize.
2) There are three ways of executing the **Find in Tree** command:
   a) Execute the main menu **View|Find in Tree** command
   b) or, if the Class Toolbar is visible, click on the **Find in Tree** button
   c) or, right click on the selected item invoking the popup menu and execute the **Find in Tree** command.

The tree in the Session Browser will readjust and expand if necessary. The definitional object pointed to by the selected object will be selected and highlighted.

Variable Definition Editor

Variable Definition Editor Explained

The **Variable Definition Editor** lets you declare variables within a class. It may be invoked by double-clicking a variable name in the Variable interface of a Class in the Session Browser.

Objects are distinguished by a unique identifier known as an OID (object ID). The externally visible behavior of an object is defined by its methods and properties. An object encapsulates state information, which is stored using instance and class variables. Their values are accessible only within the definition of the methods and properties of the class.

EsiObjects supports two atomic value types: a string and an Object Identifier (OID). The string definition is based on the ANSI MUMPS definition of a string. The OID has been added to the language specification as a part of the EsiObjects object model.

A variable may be bound to a simple literal string (or number), or it may refer to another object by its OID. The EsiObjects Variable Definition Editor is used to declare these variables and the type of value they will be bound to. You can control if and when a variable is initialized at object creation time and what its initial value will be.

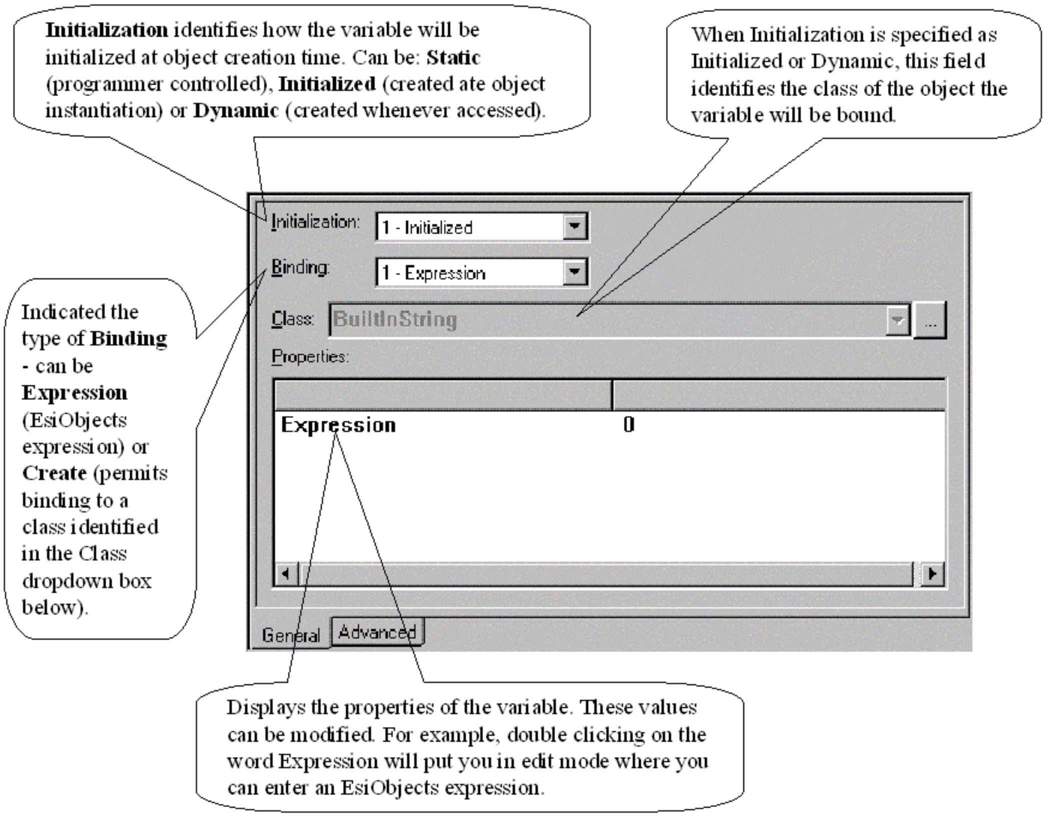
An instance variable for a class may be initialized when an object of the given class is instantiated (created). Initialization may be deferred until the value of the variable is first referenced in code. Alternatively, initialization can be bypassed entirely. In this case, the programmer must explicitly assign the value of the variable.

The initial value may be specified as an expression or an object pointer. In the latter case, you specify the class of the target object, which is created when the variable is initialized.

In addition, if the variable is initialized with the OID of another object, there are advanced features that enable you to define the parameters used when that object is created and the parentage of the object. In other words, is the object referenced by the variable owned by the parent object? Or it is an external object that is a peer of the parent?

General Tab Sheet

The **Variable Definition Editor** is used to declare and edit EsiObjects Variable Definitions. It is invoked by selecting the Variable interface within the Session Browser and then double-clicking on a variable icon in the detail pane. The editor contains two different displays, covering both General and Advanced topics.



Initialization: 1 - Initialized

Binding: 1 - Expression

Class: BuiltInString

Properties:

Expression 0

Initialization identifies how the variable will be initialized at object creation time. Can be: **Static** (programmer controlled), **Initialized** (created ate object instantiation) or **Dynamic** (created whenever accessed).

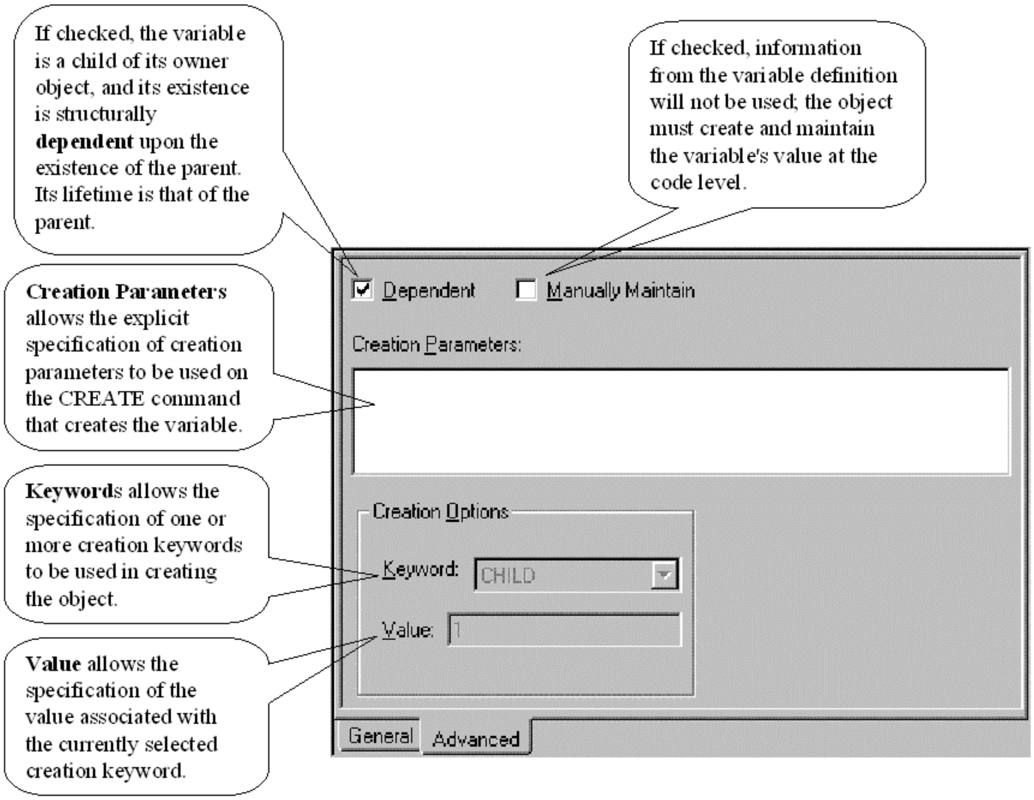
When Initialization is specified as Initialized or Dynamic, this field identifies the class of the object the variable will be bound.

Indicated the type of **Binding** - can be **Expression** (EsiObjects expression) or **Create** (permits binding to a class identified in the Class dropdown box below).

Displays the properties of the variable. These values can be modified. For example, double clicking on the word Expression will put you in edit mode where you can enter an EsiObjects expression.

The Variable Definition Editor's **General** display supports the basics of variable creation. In many cases, it is all that is needed to create a variable definition.

Advanced Tab Sheet



If checked, the variable is a child of its owner object, and its existence is structurally **dependent** upon the existence of the parent. Its lifetime is that of the parent.

If checked, information from the variable definition will not be used; the object must create and maintain the variable's value at the code level.

Creation Parameters allows the explicit specification of creation parameters to be used on the CREATE command that creates the variable.

Keywords allows the specification of one or more creation keywords to be used in creating the object.

Value allows the specification of the value associated with the currently selected creation keyword.

☑ Dependent □ Manually Maintain

Creation Parameters:

Creation Options

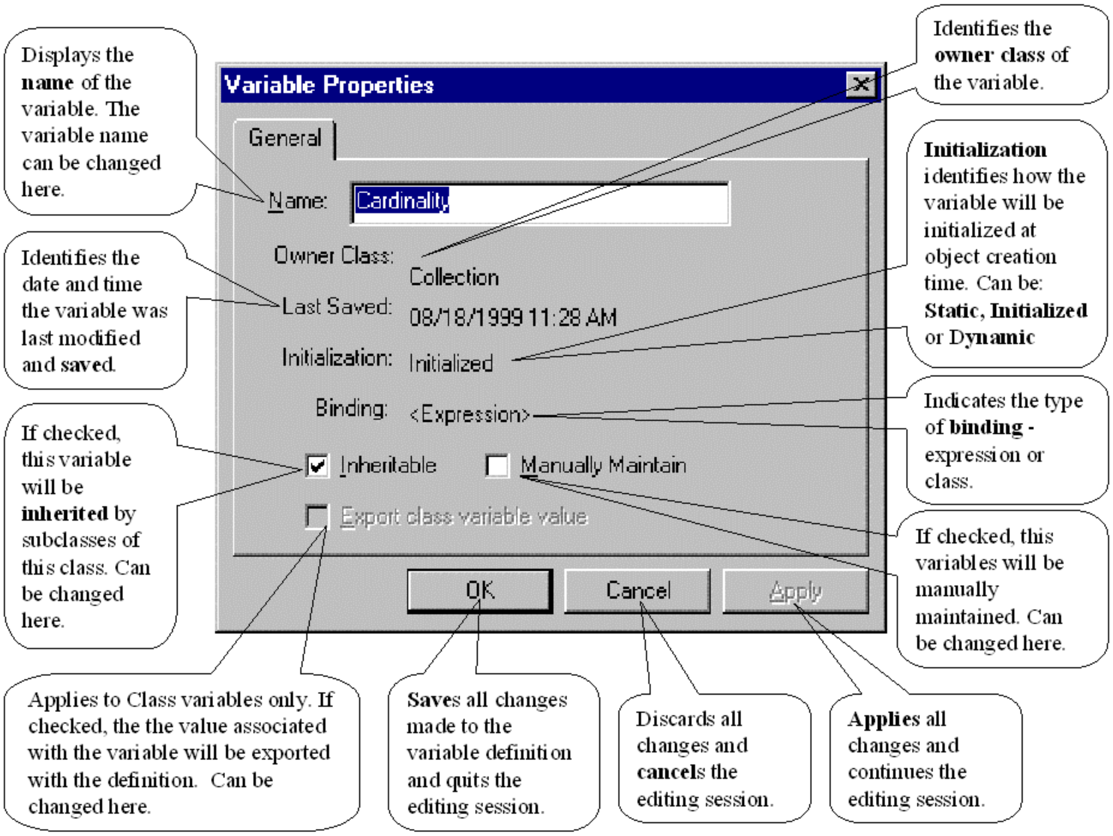
Keyword: CHILD

Value: 1

The Variable Definition Editor's **Advanced** display allows more detailed levels of information to be specified when creating a variable definition. It includes direct support for **CREATE** command parameters, and allows the variable's definition to be *manually* maintained by writing the relevant code "by hand".

Variable Properties

The Variable Properties property sheet can be invoked from the Main Menu **Object|Properties** command or the Variable popup menu **Properties** command when the variable is selected in the Session Browser It displays the properties of a specific variable.



Name: Cardinality

Owner Class: Collection

Last Saved: 08/18/1999 11:28 AM

Initialization: Initialized

Binding: <Expression>

☑ Inheritable ☐ Manually Maintain

☐ Export class variable value

Displays the **name** of the variable. The variable name can be changed here.

Identifies the date and time the variable was last modified and saved.

If checked, this variable will be **inherited** by subclasses of this class. Can be changed here.

Applies to Class variables only. If checked, the the value associated with the variable will be exported with the definition. Can be changed here.

Saves all changes made to the variable definition and quits the editing session.

Discards all changes and **cancels** the editing session.

Applies all changes and continues the editing session.

Identifies the **owner class** of the variable.

Initialization identifies how the variable will be initialized at object creation time. Can be: **Static, Initialized** or **Dynamic**

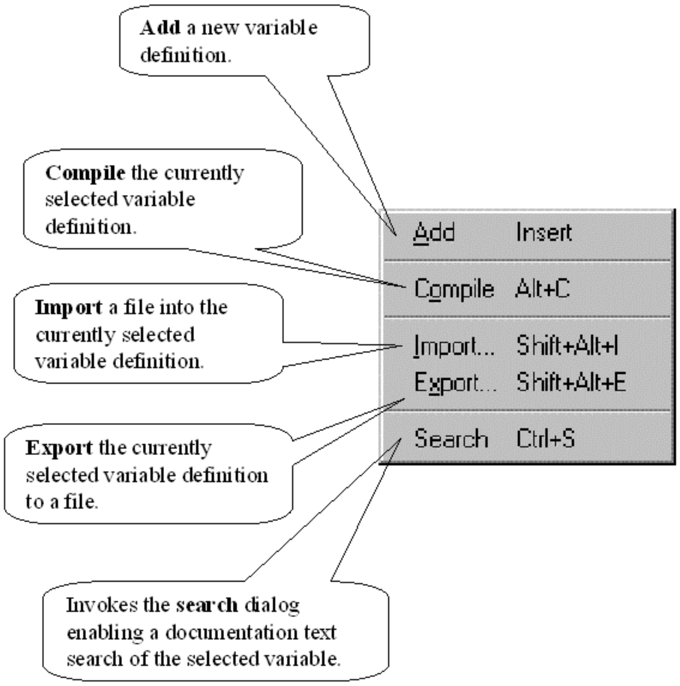
Indicates the type of **binding -** expression or class.

If checked, this variables will be manually maintained. Can be changed here.

Variable Menus

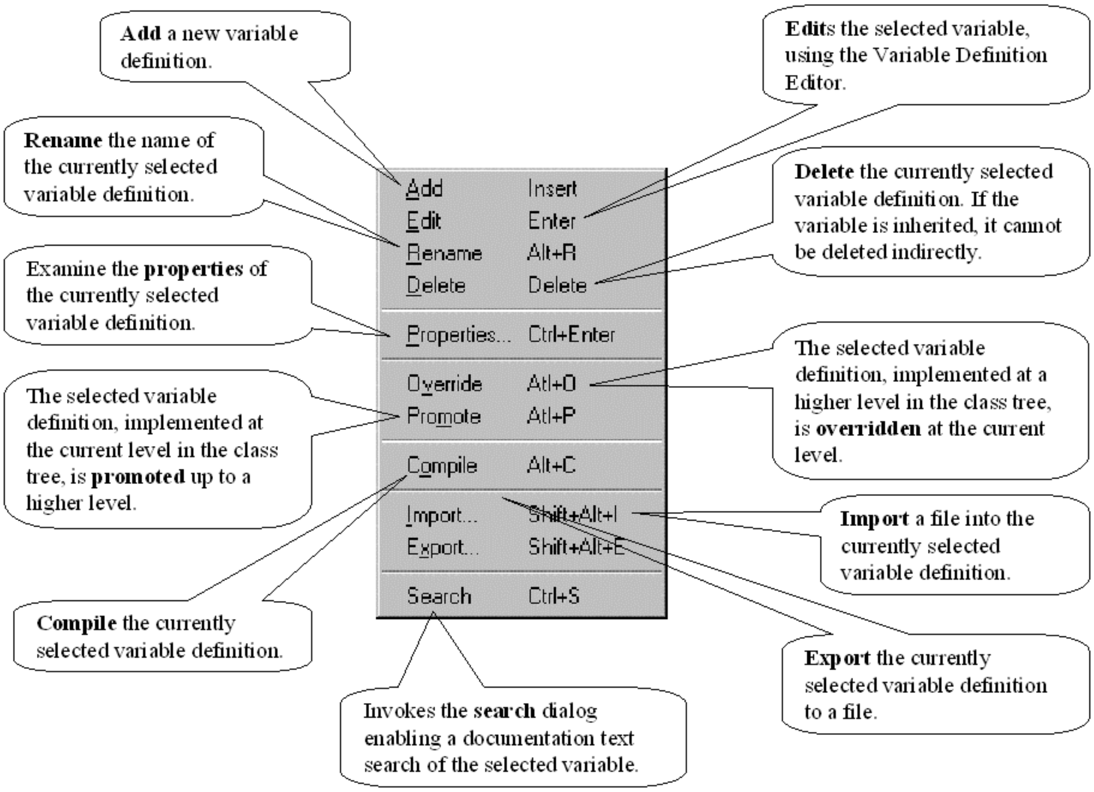
Interface Popup Menu Commands

The Variable popup menu is invoked from the Session Browser, by right clicking on the Variables interface of a class or pressing the **Shift+F10** key combination.


Add a new variable definition.
Compile the currently selected variable definition.
Import a file into the currently selected variable definition.
Export the currently selected variable definition to a file.
Invokes the **search** dialog enabling a documentation text search of the selected variable.
Add Insert
Compile Alt+C
Import... Shift+Alt+I
Export... Shift+Alt+E
Search Ctrl+S

Variables Popup Menu Commands

The Variables popup menu is invoked from the Session Browser, by right clicking on a specific class or instance variable or pressing **Shift+F10** key combination.



Add a new variable definition.

Rename the name of the currently selected variable definition.

Examine the **properties** of the currently selected variable definition.

The selected variable definition, implemented at the current level in the class tree, is **promoted** up to a higher level.

Compile the currently selected variable definition.

Invokes the **search** dialog enabling a documentation text search of the selected variable.

Edits the selected variable, using the Variable Definition Editor.

Delete the currently selected variable definition. If the variable is inherited, it cannot be deleted indirectly.

The selected variable definition, implemented at a higher level in the class tree, is **overridden** at the current level.

Import a file into the currently selected variable definition.

Export the currently selected variable definition to a file.

Using the Variable Definition Editor

Invoking the Variable Definition Editor

There are four ways to invoke the Variable Definition Editor. From within the Session Browser, select the variable by clicking on its icon or name then:

1) Double click on the icon or name.
2) Press the **Enter** key.
3) Pull down the Main Edit Menu and select the **Edit** command.
4) Invoke the popup menu by right clicking on it (or by pressing **Shift+F10)**. Choose the **Edit** command.

Editing Variable Properties

There are three ways to invoke the Variable Property Sheet. From within the Session Browser, select the variable by clicking on its icon or name then:

1) Press the **Ctrl+Enter** key.
2) Execute the **Object|Properties** command.
3) Invoke the popup menu by right clicking on it (or by pressing **Shift+F10)**. Choose the **Properties** command.

The Variable Definition Property Sheet dialog contains only one tab that contains all the properties of a variable. Those properties that can be changed are highlighted and those that cannot are grayed out.

Deleting a Variable

There are three ways to delete a variable. From within the Session Browser, select the variable by clicking on its icon or name then:

1) Press the **Del** key.
2) Pull down the Main Edit Menu and select the **Delete** command.
3) Invoke the popup menu by right clicking on it (or by pressing **Shift+F10)**. Choose the **Delete** command.

Method Editor

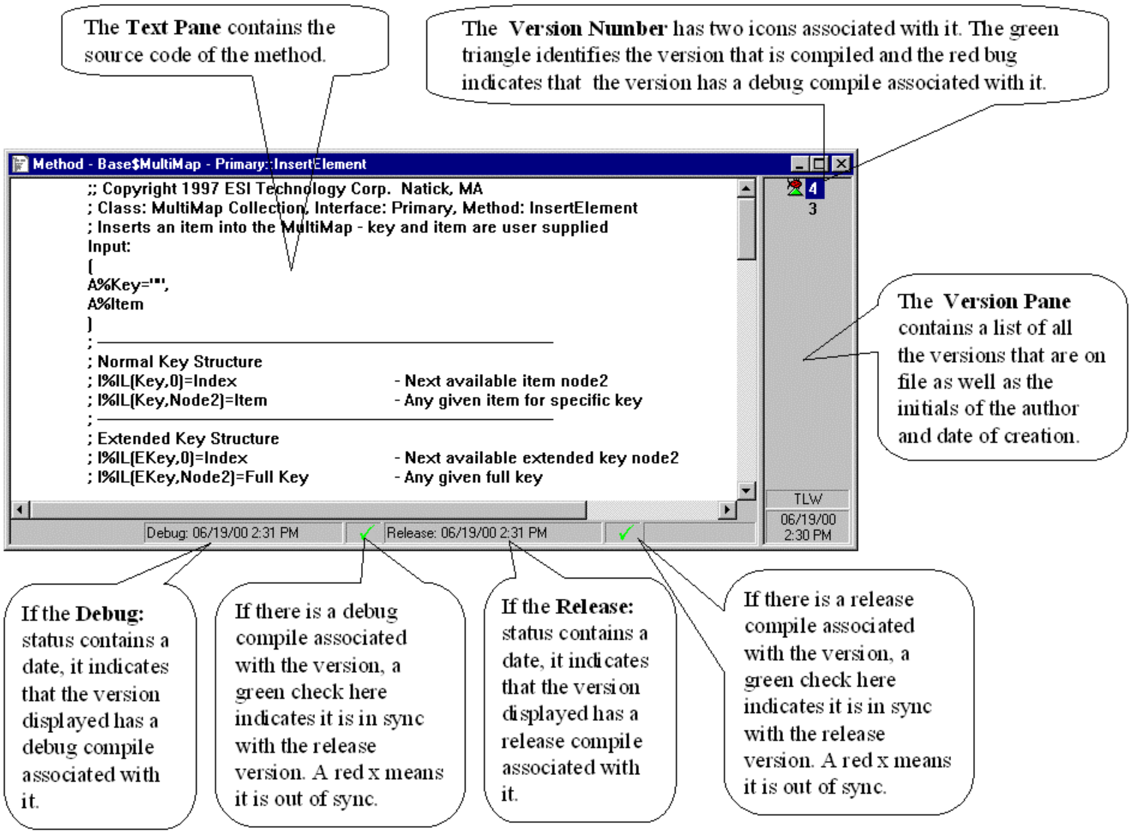
The **Method Editor** enables you to write a method. A method is a body of code that performs a specific operation within the object. Unlike a property, which usually represents the data within the object, a method represents an operation. A method gives the object some of its behavior.

The Method Editor allows you to enter code, check its syntax, compile and save multiple versions of the source code. Compiling a method means compiling the source code for runtime use. The code can be compiled for release or debugging.

A Method Editor is invoked by double clicking on a Method Icon in the Session Browser. Note that once changes are made to the method code, the code must be compiled to reflect the changes in the actual invocation of that method.

Method Editor Explained

The **Method Editor** is used to edit EsiObjects methods. It appears in the client area of the EsiObjects Main Window . It can be invoked by double-clicking on a Method icon in the library tree structure. The illustration below describes all of the components of the Method Editor.



The **Text Pane** contains the source code of the method.

The **Version Number** has two icons associated with it. The green triangle identifies the version that is compiled and the red bug indicates that the version has a debug compile associated with it.

Method - Base$MultiMap - Primary::InsertElement
4
3
TLW
06/19/00 2:30 PM
Debug: 06/19/00 2:31 PM
✓
Release: 06/19/00 2:31 PM
✓

```
;; Copyright 1997 ESI Technology Corp.  Natick, MA
; Class: MultiMap Collection, Interface: Primary, Method: InsertElement
; Inserts an item into the MultiMap - key and item are user supplied
Input:
(
A%Key="",
A%Item
)
; ____________________________________________________________
; Normal Key Structure
; I%IL(Key,0)=Index                       - Next available item node2
; I%IL(Key,Node2)=Item                    - Any given item for specific key
; ____________________________________________________________
; Extended Key Structure
; I%IL(EKey,0)=Index                      - Next available extended key node2
; I%IL(EKey,Node2)=Full Key               - Any given full key
```

The **Version Pane** contains a list of all the versions that are on file as well as the initials of the author and date of creation.

If the **Debug:** status contains a date, it indicates that the version displayed has a debug compile associated with it.

If there is a debug compile associated with the version, a green check here indicates it is in sync with the release version. A red x means it is out of sync.

If the **Release:** status contains a date, it indicates that the version displayed has a release compile associated with it.

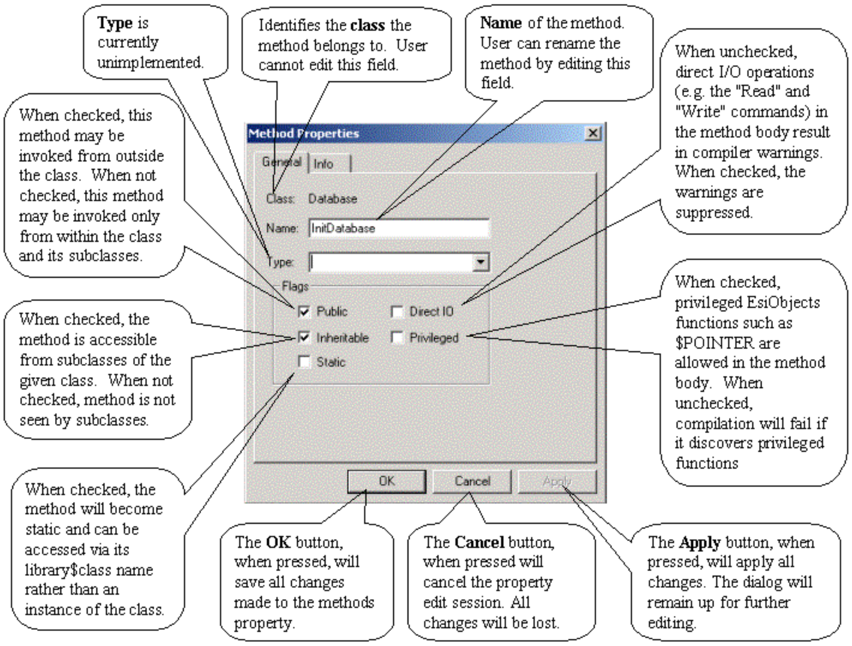
If there is a release compile associated with the version, a green check here indicates it is in sync with the release version. A red x means it is out of sync.

Method Properties

Selecting the Properties entry from the appropriate pull-down or pop-up menu accesses the Method Properties dialog. It lets the user view and edit the properties of the method.

General Method Property Sheet

The General property sheet of a method contains editable information about the method. It lets you change this information, changing the behavioral characteristics of the method.



Class: Database

Name: InitDatabase

Type:

Flags

☑ Public
□ Direct IO
☑ Inheritable
□ Privileged
□ Static

Type is currently unimplemented.

Identifies the **class** the method belongs to. User cannot edit this field.

Name of the method. User can rename the method by editing this field.

When unchecked, direct I/O operations (e.g. the "Read" and "Write" commands) in the method body result in compiler warnings. When checked, the warnings are suppressed.

When checked, this method may be invoked from outside the class. When not checked, this method may be invoked only from within the class and its subclasses.

When checked, the method is accessible from subclasses of the given class. When not checked, method is not seen by subclasses.

When checked, privileged EsiObjects functions such as $POINTER are allowed in the method body. When unchecked, compilation will fail if it discovers privileged functions

When checked, the method will become static and can be accessed via its library$class name rather than an instance of the class.

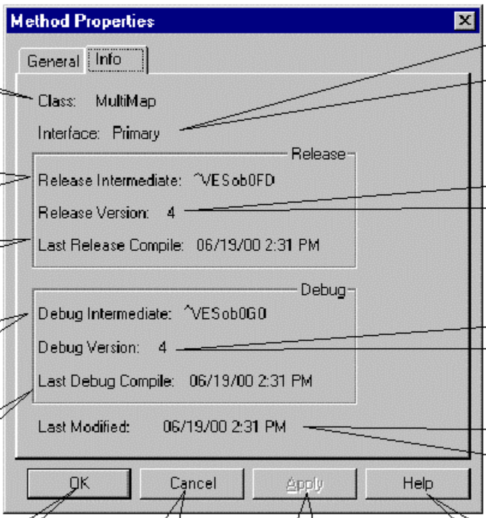
The **OK** button, when pressed, will save all changes made to the methods property.

The **Cancel** button, when pressed will cancel the property edit session. All changes will be lost.

The **Apply** button, when pressed, will apply all changes. The dialog will remain up for further editing.

Information Method Property Sheet

The Info property sheet contains information about the method. It cannot be edited.



Class: MultiMap

Interface: Primary

Release Intermediate: ^VESob0FD

Release Version: 4

Last Release Compile: 06/19/00 2:31 PM

Debug Intermediate: ^VESob0G0

Debug Version: 4

Last Debug Compile: 06/19/00 2:31 PM

Last Modified: 06/19/00 2:31 PM

Identifies the **class** the method belongs to. User cannot edit this field.

Identifies which **interface** in the class the method belongs to. User cannot edit this field.

Name of M routine containing **release** compiled code for the method.

Identifies the currently compiled **release version** of the method.

Date and time of **last release compile**.

Name of M routine containing debug compiled code for the method.

Identifies the current compiled **debug version** of the method.

Date and time the method source code was **last saved** after being modified.

Date and time of **last debug compile**.

The OK button, when pressed, will save all changes made to the methods property.

The **Cancel** button, when pressed will cancel the property edit session. All changes will be lost.

The **Apply** button, when pressed, will apply all changes. The dialog will remain up for further editing.

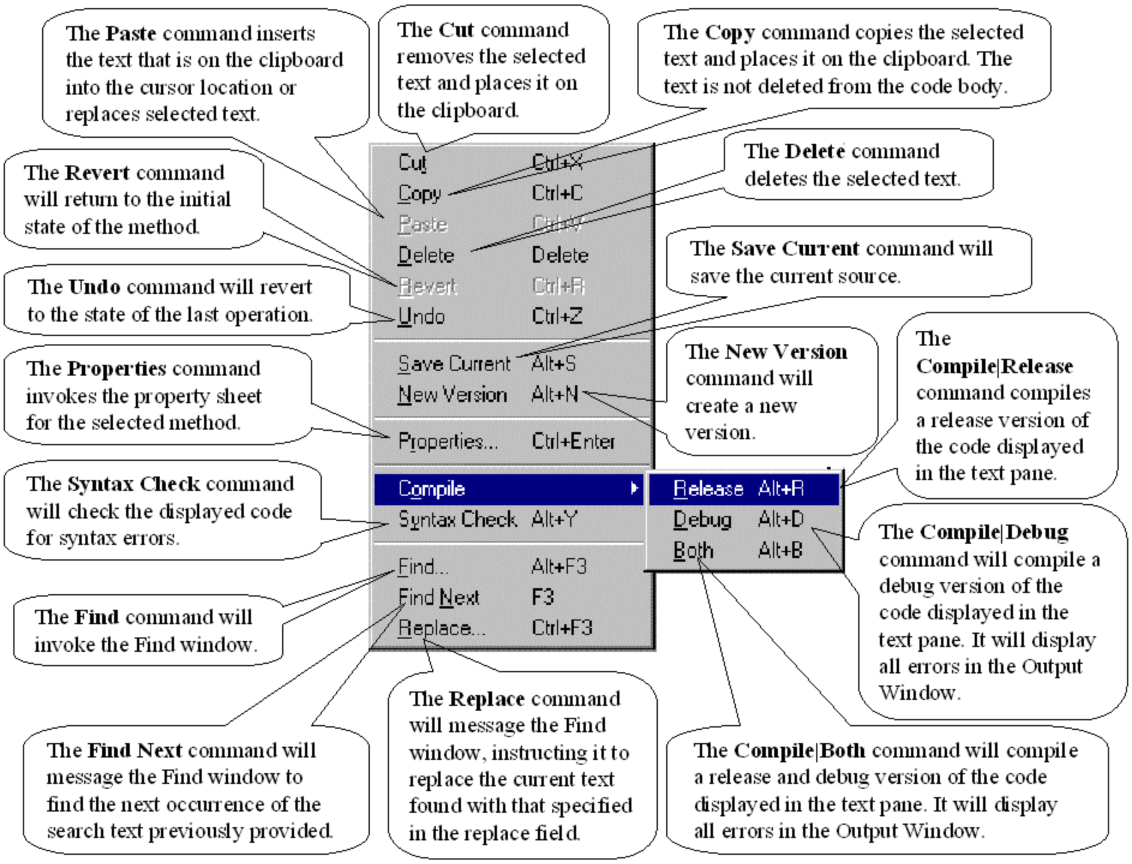
Pressing the **Help** button will invoke the Acrobat Reader and this documentation will be displayed.

Method Menus

Access to method functionality is available through the Main Menu as well as popup menus within each windowpane. The popup menu commands are explained below.

Source Code Popup Menu

The Source Code popup menu is invoked by right clicking inside the source code pane of the Method Editor window or pressing **Shift+F10** key combination.



The **Paste** command inserts the text that is on the clipboard into the cursor location or replaces selected text.

The **Cut** command removes the selected text and places it on the clipboard.

The **Copy** command copies the selected text and places it on the clipboard. The text is not deleted from the code body.

The **Revert** command will return to the initial state of the method.

The **Delete** command deletes the selected text.

The **Undo** command will revert to the state of the last operation.

The **Save Current** command will save the current source.

The **Properties** command invokes the property sheet for the selected method.

The **New Version** command will create a new version.

The **Compile|Release** command compiles a release version of the code displayed in the text pane.

The **Syntax Check** command will check the displayed code for syntax errors.

The **Compile|Debug** command will compile a debug version of the code displayed in the text pane. It will display all errors in the Output Window.

The **Find** command will invoke the Find window.

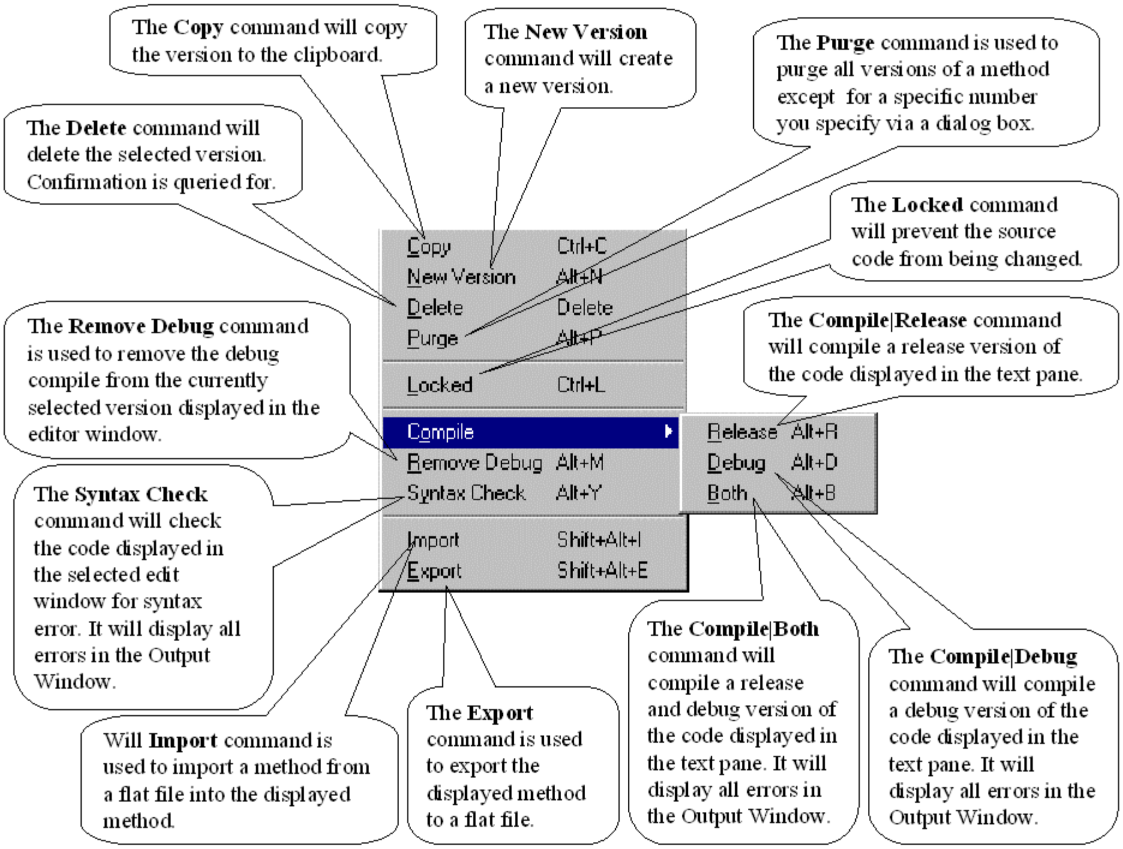
The **Find Next** command will message the Find window to find the next occurrence of the search text previously provided.

The **Replace** command will message the Find window, instructing it to replace the current text found with that specified in the replace field.

The **Compile|Both** command will compile a release and debug version of the code displayed in the text pane. It will display all errors in the Output Window.

Version Popup Menu

The Version popup menu is invoked by right clicking inside the version history pane of the Method Editor Window or pressing the **Shift+F10** key combination.



The **Copy** command will copy the version to the clipboard.

The **New Version** command will create a new version.

The **Purge** command is used to purge all versions of a method except for a specific number you specify via a dialog box.

The **Delete** command will delete the selected version. Confirmation is queried for.

The **Locked** command will prevent the source code from being changed.

The **Remove Debug** command is used to remove the debug compile from the currently selected version displayed in the editor window.

The **Compile|Release** command will compile a release version of the code displayed in the text pane.

The **Syntax Check** command will check the code displayed in the selected edit window for syntax error. It will display all errors in the Output Window.

Will **Import** command is used to import a method from a flat file into the displayed method.

The **Export** command is used to export the displayed method to a flat file.

The **Compile|Both** command will compile a release and debug version of the code displayed in the text pane. It will display all errors in the Output Window.

The **Compile|Debug** command will compile a debug version of the code displayed in the text pane. It will display all errors in the Output Window.

Using the Method Editor

Creating a Method

To create a method, follow the steps below.

1) In the Session Browser, expand the class to which you wish to add a method by clicking on the expansion box (box with the + in it).

2) Now expand the interface that will hold the method. This will display all exiting services as a sub-tree to the interface.

3) Click the right mouse button on the interface name to bring up the popup menu. Select the **Add** option. (Note that you also could press the **Insert** key to add an item.)

4) The Add to Interface dialog appears. Enter the name of the method. Valid names must begin with an alpha character and can contain up to 32 alphanumeric characters. Make the name something that gives a sense of what behavior the method provides.

5) Pull down the combo box and select Method from the list.
6) Click on the OK button to add the method to the interface.

Editing a Method

Once a method has been added to the interface, you can manipulate it in many ways. By selecting the method and right clicking on it, a popup menu is displayed that allows you to delete, edit, and rename the method among other operations. Note that each operation has a keyboard equivalent that will invoke the action directly.

To edit a method, perform the following steps:

1. Select the method to edit. As described above, clicking the right mouse button on the method name will display a popup menu from which you can select the Edit option.
2. The method editor will appear in the client area of the Main Window. It will contain the source code to be edited. You can now create or modify the code. You can also create new versions as well as compile and/or syntax check the code. If you want to create or modify documentation on the method, make sure the Documentation Window is active. Simply click in the Documentation Window and start typing the text. If the Documentation Toolbar is not active, chose the **View|Toolbars|Documentation** to activate it.
3. Additionally, pressing the Enter key or double-clicking on the method name in the interface will also invoke the editor.

Deleting a Method

To delete a method, follow these steps:

1) In the Session Browser select the method that you want to delete.
2) Right click on the method icon and select the Delete command from the menu.
3) A verification dialog will ask you if you want to continue. Answer **Yes**. At this point the method will be deleted and the library structure will readjust to the deletion.

Reusing the Method Editor Window

User Option Preference

The User Option Preference tab sheet contains a check box call Redisplay. When this box is checked, EsiObjects will always look of a method editor in the client area before creating a new one. If one exists, it will reuse that Method Editor Window, displaying the newly selected method code in that window. If the method in that window had changes made to it, the window will let you save it before proceeding.

Drag-and-Drop

Another approach to reusing a Method Editor window that is already displayed in the client area is to drag-and-drop the method name into an open editor as outlined below:

1. There must be a Method Editor already open in the client area. And some portion of the code pane must be visible.
2. Select the method to edit from the Session Browser, hold down the left mouse button on the item name and drag the cursor to the editor's code pane.
3. Note that the cursor will indicate when a valid area to drop the item is reached by changing to an cursor arrow with a plus box attached to it. When the cursor shows this indication, lift the left mouse button to drop the method.
4. The context is switched to the Method Editor context.
5. If the original method being edited has been modified without being saved, you will be prompted to save the changes prior to the context being switched. Click **Save** to save the changes, **Discard** to throw away the changes, or **Cancel** to cancel the drop operation.

Editing Method Properties

Properties of a method can be edited using the Method Property sheets. There are at least 3 ways to invoke the Method Property sheet:

1) From within the Session Browser, select the appropriate method and press **Ctrl+Enter**.
2) From within the Session Browser, select the appropriate method and invoke the popup menu by right clicking on the name or icon, or by pressing **Shift+F10**. Choose the **Properties** menu item.
3) From within the method editor, select **Properties** from the **Edit** menu.

Once the property sheet is displayed in the client area, you can edit those fields that are changeable.

See the Method Properties under Method Editor Explained section above for a complete description of all the fields.

Managing Source Versions

Explicit Source Management

When saving the source code of a method (using the source code popup menu) there are four distinct options related to compiling and managing source code versions:

1) **Save Current** Save the source code under the version number currently being edited.
2) **New Version** Save the source code under a new (highest) version number.
3) **Syntax Check** Check the syntax of the current source text.
4) **Compile** Compile the current source text.

Source Code User Options

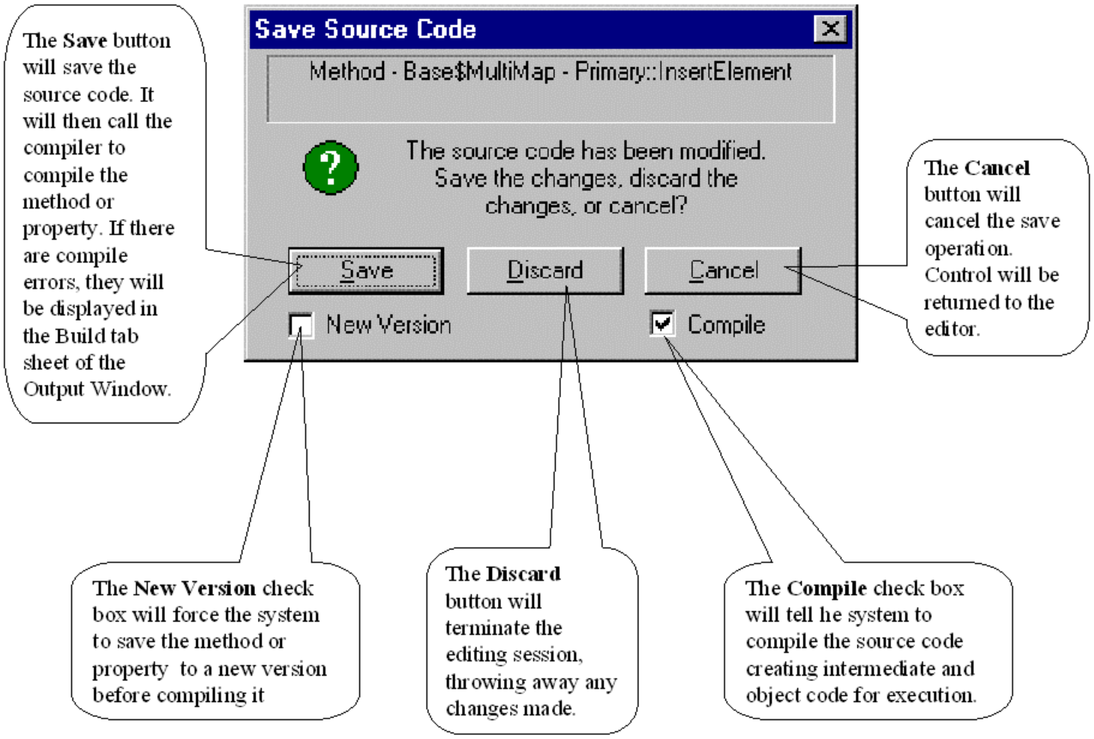
In the EsiObjects User Options, there are two specific options related to source code version control:

Auto New Version When prompted to save source code, the **New Version** check box on the Save dialog will default to what is set here. When checked, the default action on saving source code will be to create a new version of the source code.

Compile On Save When prompted to save source code, the **Compile** check box on the Save dialog will default to what is set here. When checked, the default action on saving source code will be to compile the source code after saving.

Default Save Options

If the user closes a source code object without saving changes, then the Save Source Code dialog shown below will be displayed.



Method - Base$MultiMap - Primary::InsertElement

The source code has been modified. Save the changes, discard the changes, or cancel?

□ New Version

☑ Compile

The **Save** button will save the source code. It will then call the compiler to compile the method or property. If there are compile errors, they will be displayed in the Build tab sheet of the Output Window.

The **Cancel** button will cancel the save operation. Control will be returned to the editor.

The **New Version** check box will force the system to save the method or property to a new version before compiling it

The **Discard** button will terminate the editing session, throwing away any changes made.

The **Compile** check box will tell he system to compile the source code creating intermediate and object code for execution.

The options that are selected, by default, will be determined by the appropriate user options.

Property Editor

The **Property Editor** is a tool that enables you to define a property within a particular interface within a class. A property is a specialized method that supports up to 10 types of access known as *accessors*. Properties are typically used to expose the state of an object. For example, the Value accessor can retrieve the value of an instance variable and return it to the caller. It could also perform some calculation on the values internal to the object. In addition to retrieving a value from an object, the Assign accessor is designed to alter the object's state. It can do this by assigning new instance variable values to the object.

One important thing to remember is that each accessor is associated with the EsiObjects language. For example, the Value accessor would be invoked in the following construct:

```
Set A%Temp=I%Customer.Name
```

This statement will cause the Value accessor of the Name property associated with the object accessed by I%Customer to be executed. The value returned form that accessor would be bound to the temporary variable A%Temp for local use.

Conversely, the Assign accessor would be invoked for the following construct:

```
Set I%Customer.Name="ACME Tire Company"
```

The Assign accessor of the Name property associated with the I%Customer object would be passed a parameter that would contain the name string on the right. The Assign accessor code would associate the value passed in with the proper instance variable.

The **Property Editor** enables you to define the code (if any) for the particular types of access you are going to allow on the property.

Property Editor Explained

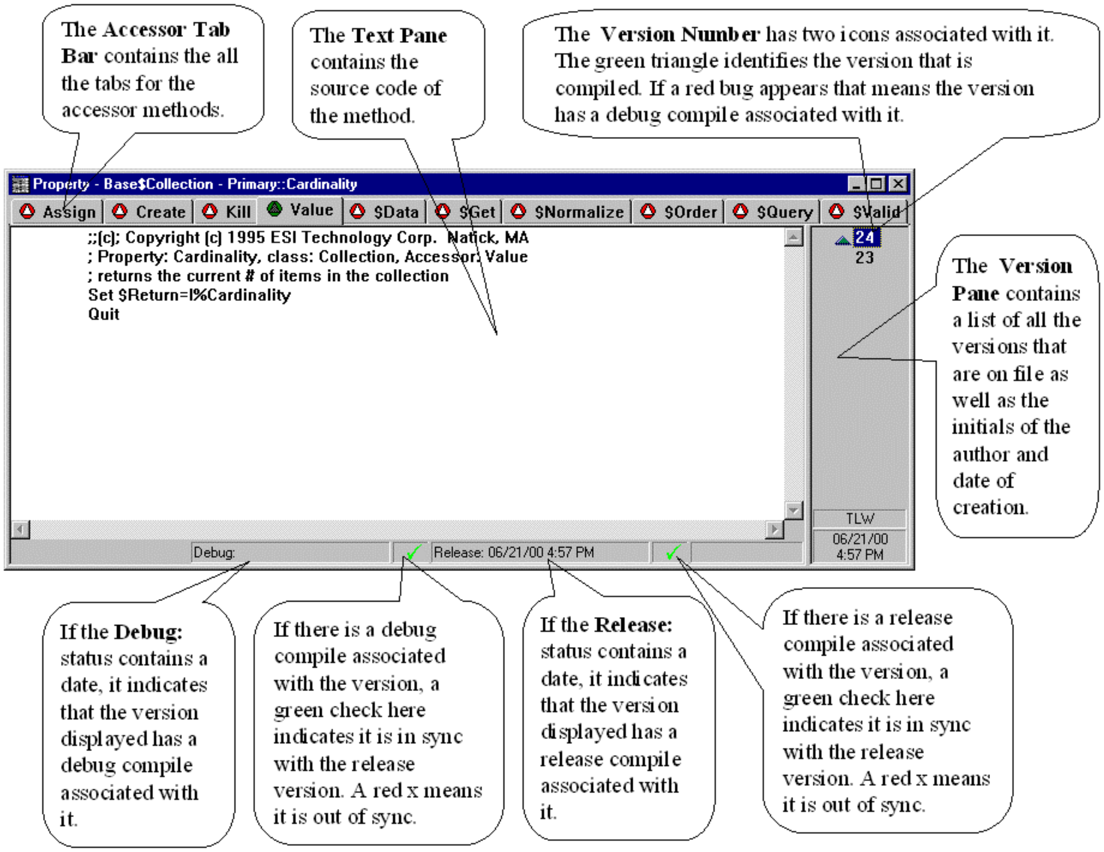
The **Property Editor** allows you to enter code, check its syntax, compile and save multiple versions of the source code for each accessor. Compiling a property accessor means compiling the source code for runtime use. The code can be compiled for release, debugging or both.

A Property Editor is invoked by double clicking on a Property Icon in the Session Browser. Note that once changes are made to the property accessor code, the code must be compiled to reflect the changes in the actual invocation of that accessor.

Property Editor Window

The **Property Editor** is used to edit Properties. It appears as a client area of the EsiObjects CDE Main Window.

The picture below illustrates the general components of the Property Editor. Keep in mind that for a specific accessor, most of the functionality is the same as a method. In fact, internally to EsiObjects, each accessor is a method.



The **Accessor Tab Bar** contains the all the tabs for the accessor methods.

The **Text Pane** contains the source code of the method.

The **Version Number** has two icons associated with it. The green triangle identifies the version that is compiled. If a red bug appears that means the version has a debug compile associated with it.

```
;;[c]; Copyright (c) 1995 ESI Technology Corp.  Natick, MA
; Property: Cardinality, class: Collection, Accessor: Value
; returns the current # of items in the collection
Set $Return=I%Cardinality
Quit
```

Debug:

Release: 06/21/00 4:57 PM

The **Version Pane** contains a list of all the versions that are on file as well as the initials of the author and date of creation.

If the **Debug:** status contains a date, it indicates that the version displayed has a debug compile associated with it.

If there is a debug compile associated with the version, a green check here indicates it is in sync with the release version. A red x means it is out of sync.

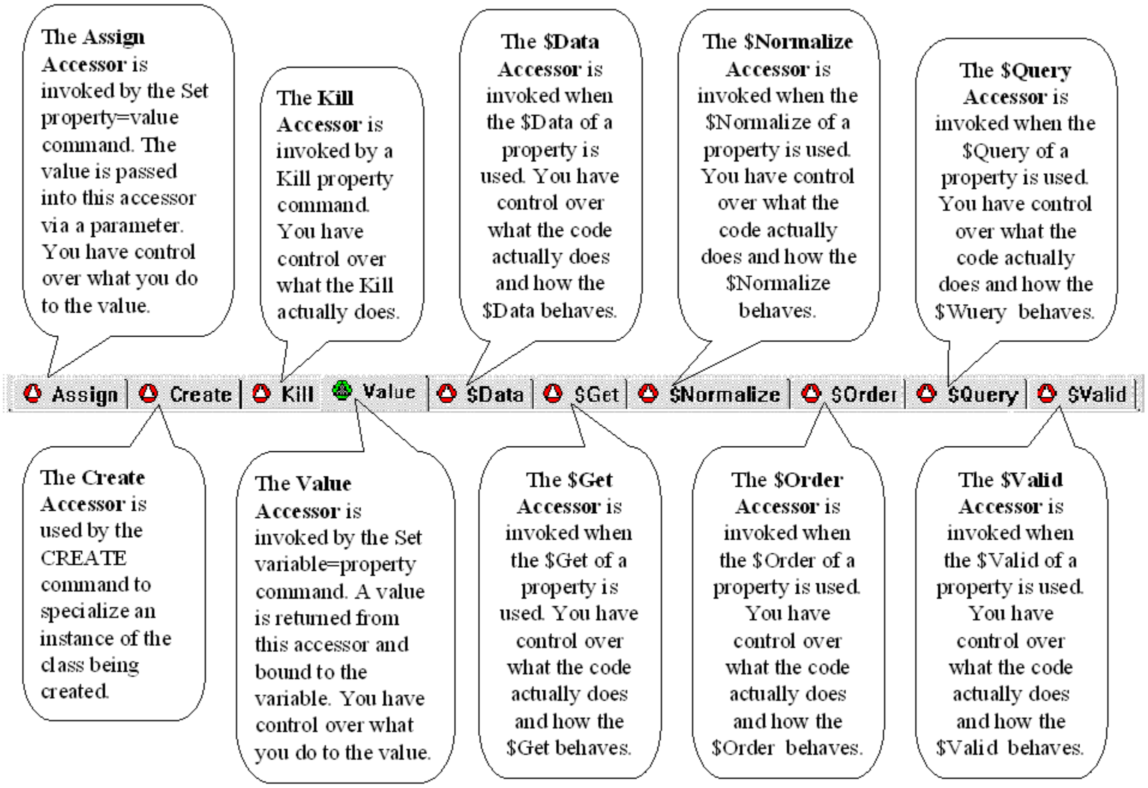
If the **Release:** status contains a date, it indicates that the version displayed has a release compile associated with it.

If there is a release compile associated with the version, a green check here indicates it is in sync with the release version. A red x means it is out of sync.

One of the User Preferences provided by EsiObjects in its **Tools|Options** menu entry is the **Compile on Save** feature. When this preference is selected, a release compile is automatically performed whenever a version is changed and saved. Checking this preference causes the check box in the Save Source Code dialog to be checked. If **Compile on Save** is not selected, the user can save changes to the source code without compiling it. When the code is invoked, the source code shown on screen may be out of sync with what actually gets executed. EsiObjects detects an out of sync condition when the current release or debug version was modified and saved after compilation. It alerts the user to this condition by displaying a red x after the release or debug compilation timestamp. The out of sync marker goes away when a release and debug compile are done against the same source code.

Property Editor Accessor Tab Bar

The tab bar contains the name and implementation status of each of the properties ten accessor methods.



The **Assign Accessor** is invoked by the Set property=value command. The value is passed into this accessor via a parameter. You have control over what you do to the value.

The **Kill Accessor** is invoked by a Kill property command. You have control over what the Kill actually does.

The **$Data Accessor** is invoked when the $Data of a property is used. You have control over what the code actually does and how the $Data behaves.

The **$Normalize Accessor** is invoked when the $Normalize of a property is used. You have control over what the code actually does and how the $Normalize behaves.

The **$Query Accessor** is invoked when the $Query of a property is used. You have control over what the code actually does and how the $Wuery behaves.

The **Create Accessor** is used by the CREATE command to specialize an instance of the class being created.

The **Value Accessor** is invoked by the Set variable=property command. A value is returned from this accessor and bound to the variable. You have control over what you do to the value.

The **$Get Accessor** is invoked when the $Get of a property is used. You have control over what the code actually does and how the $Get behaves.

The **$Order Accessor** is invoked when the $Order of a property is used. You have control over what the code actually does and how the $Order behaves.

The **$Valid Accessor** is invoked when the $Valid of a property is used. You have control over what the code actually does and how the $Valid behaves.

The icon that appears to the left of each accessor name contains a color that indicates the accessor’s status. A description is given below:

Icon	**Meaning**
🔴	Not implemented. The specified accessor is not implemented; even so, default template source code appears as a user preference.
🟢	Implemented here. The accessor is implemented by the selected interface. In the case of an inherited interface, "here" refers to *the class that implements the interface*.
🟡	Inherited. The accessor is implemented at a class that is an ancestor of the class that implements the selected interface.

There are ten different accessor methods, each used for a different purpose.

Item **Description**

Assign Invoked whenever there is an attempt to assign a value to the property. The accessor's first argument is the value assigned to it; the remaining arguments, if any, are array subscripts specified for the property.

Create Invoked when there is an attempt to initially create the property. The accessor's first argument is the value assigned to it; the remaining arguments, if any, are array subscripts specified for the property.

Kill Invoked when there is an attempt to kill the property. The accessor's arguments, if any, are array subscripts specified for the property.

Value Invoked whenever there is an attempt to reference the property's value. The accessor's arguments, if any, are array subscripts specified for the property.

$Data Invoked whenever the **$DATA** function is applied to the property. The accessor's arguments, if any, are array subscripts specified for the property.

$Get Invoked whenever the property is the first argument of the **$GET** function. The accessor's first argument is the default value to be returned if the property considers itself to be undefined; the remaining arguments, if any, are array subscripts specified for the property.

$Normalize Invoked whenever the property is the first argument of the **$NORMALIZE** function, which is used to normalize a potential value to one that is appropriate for the property. For example, an integer-valued property might always return an integer. The accessor's arguments are input values to be normalized.

$Order Invoked whenever the property is the first argument of a **$ORDER** function. The accessor's first argument is the direction (1 by default, -1 for reverse **$ORDER**); the remaining arguments are the array subscripts specified for the property.

$Query Invoked whenever the property is the first argument of a **$QUERY** function. The accessor's arguments are the subscripts, if any, specified for the property.

$Valid Invoked whenever the property is the first argument of the **$VALID** function. The accessor's first argument is the value to be validated; the remaining arguments, if any, are the array subscripts specified for the property.

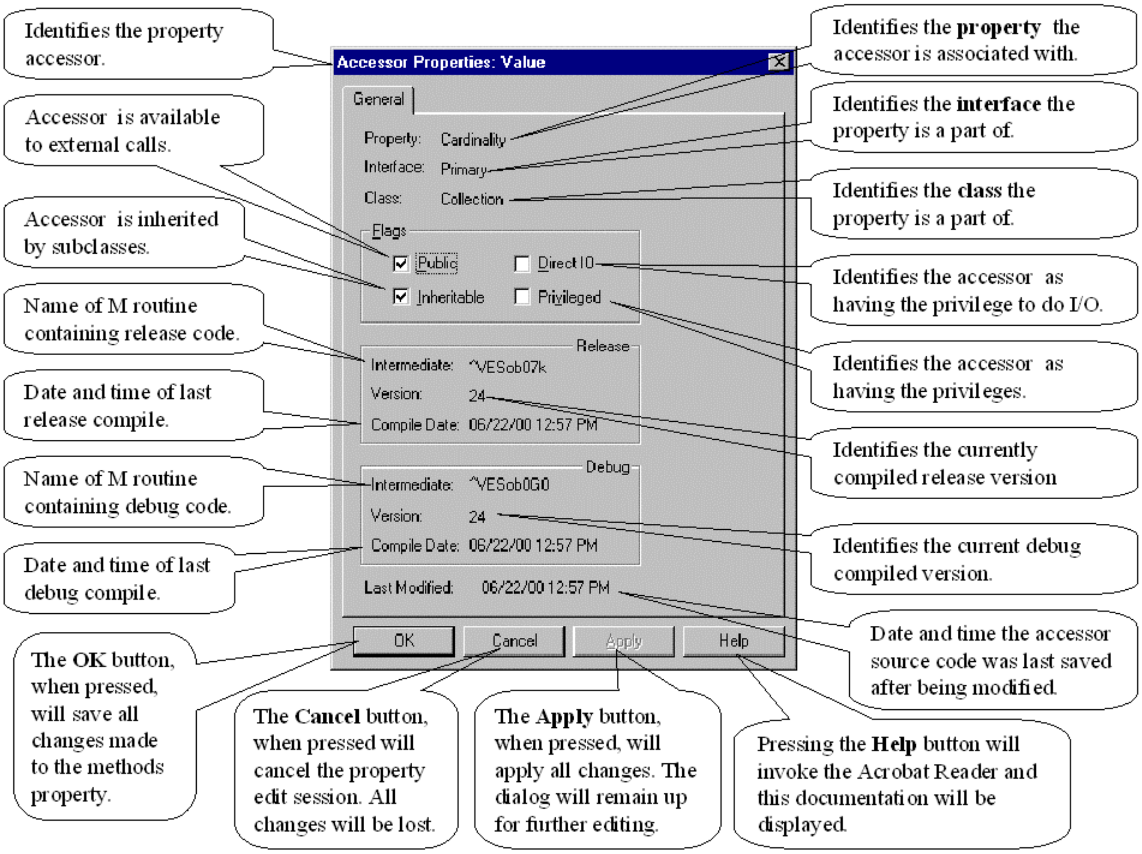
Property and Accessor Properties

There are two levels of property properties. The property level properties are on the property itself. Accessor level properties are on each accessor of the property. Accessors are, in essence, individual methods associated with the property.

The following sections identify each property type.

Accessor Properties

Right clicking on the selected accessor’s tab and choosing the Properties command from the popup accesses the Accessor Properties dialog. It lets you view and edit the properties of the selected accessor.



Property: Cardinality
Interface: Primary
Class: Collection

Flags
☑ Public ☐ Direct IO
☑ Inheritable ☐ Privileged

Release
Intermediate: ^VESob07k
Version: 24
Compile Date: 06/22/00 12:57 PM

Debug
Intermediate: ^VESob0G0
Version: 24
Compile Date: 06/22/00 12:57 PM

Last Modified: 06/22/00 12:57 PM

Identifies the property accessor.

Accessor is available to external calls.

Accessor is inherited by subclasses.

Name of M routine containing release code.

Date and time of last release compile.

Name of M routine containing debug code.

Date and time of last debug compile.

Identifies the **property** the accessor is associated with.

Identifies the **interface** the property is a part of.

Identifies the **class** the property is a part of.

Identifies the accessor as having the privilege to do I/O.

Identifies the accessor as having the privileges.

Identifies the currently compiled release version

Identifies the current debug compiled version.

Date and time the accessor source code was last saved after being modified.

The **OK** button, when pressed, will save all changes made to the methods property.

The **Cancel** button, when pressed will cancel the property edit session. All changes will be lost.

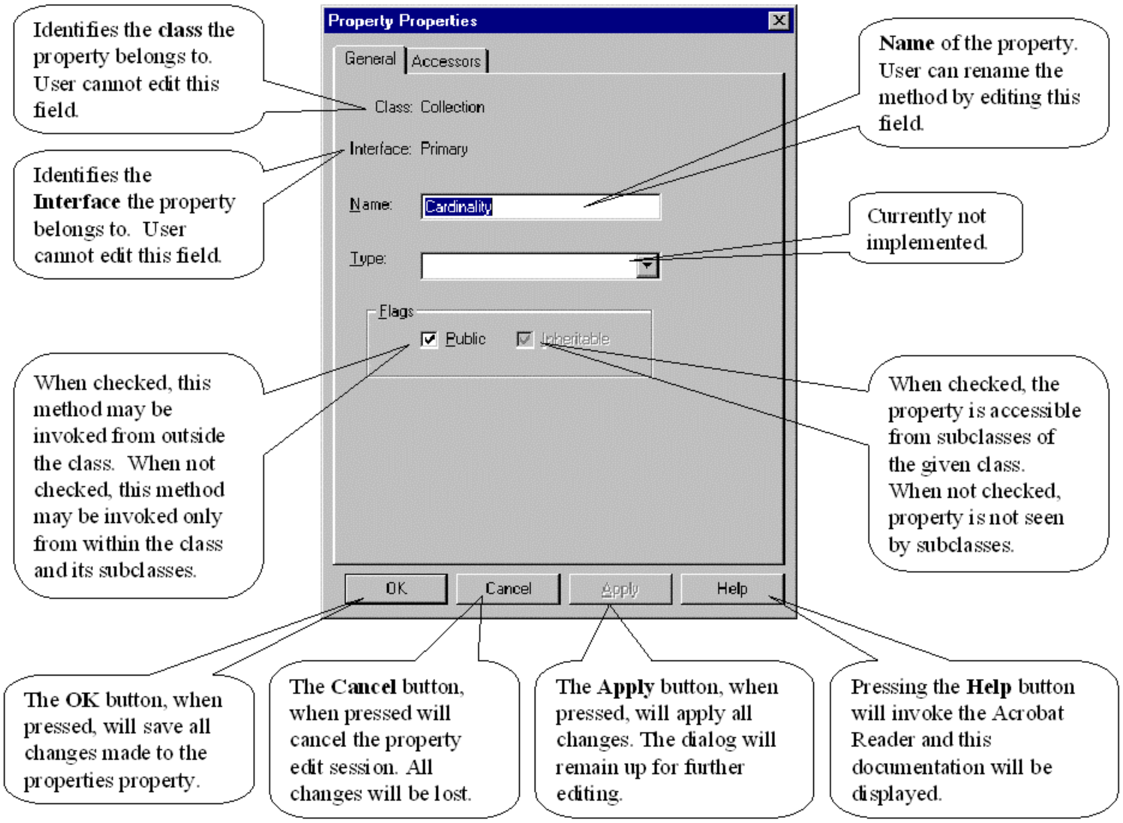
The **Apply** button, when pressed, will apply all changes. The dialog will remain up for further editing.

Pressing the **Help** button will invoke the Acrobat Reader and this documentation will be displayed.

Property Properties

Selecting the **Object|Properties** command from the Main Menu accesses the Property Properties dialog. It lets you view and edit the properties of the property.

These are the items on the General tab:



Class: Collection

Interface: Primary

Name: Cardinality

Type:

Flags

☑ Public ☑ Inheritable

Identifies the **class** the property belongs to. User cannot edit this field.

Identifies the **Interface** the property belongs to. User cannot edit this field.

When checked, this method may be invoked from outside the class. When not checked, this method may be invoked only from within the class and its subclasses.

Name of the property. User can rename the method by editing this field.

Currently not implemented.

When checked, the property is accessible from subclasses of the given class. When not checked, property is not seen by subclasses.

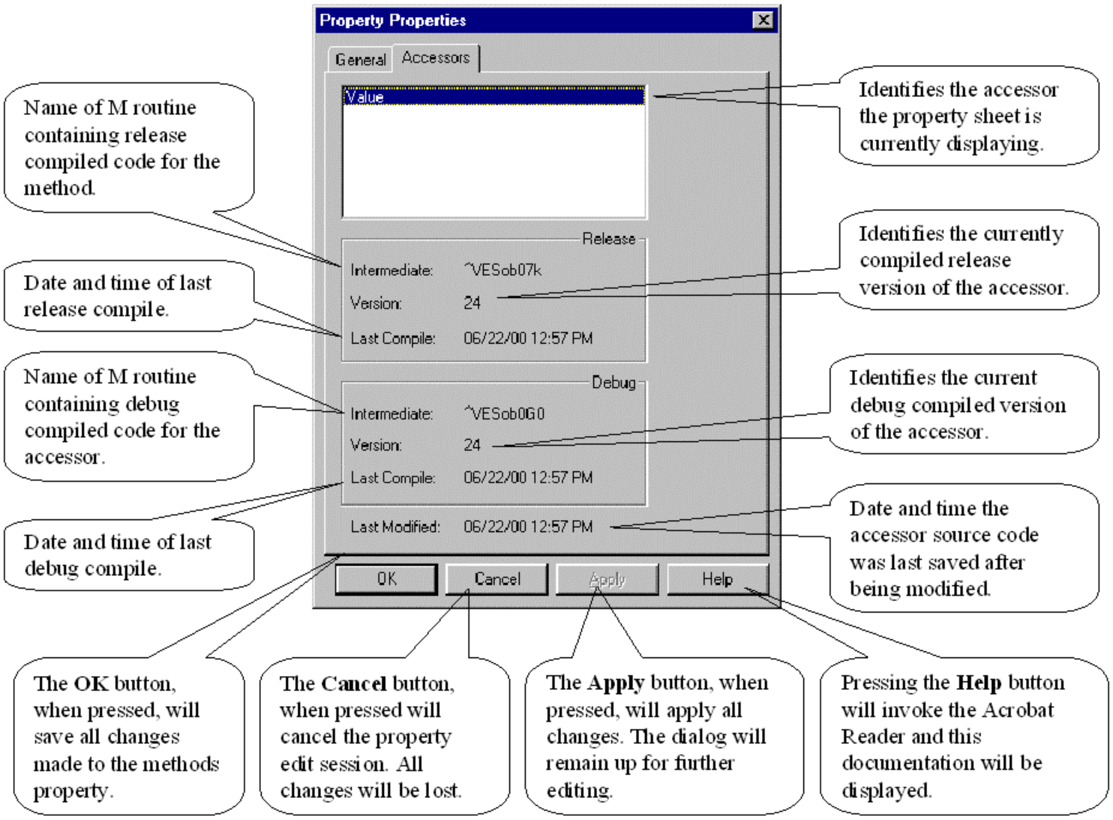
The **OK** button, when pressed, will save all changes made to the properties property.

The **Cancel** button, when pressed will cancel the property edit session. All changes will be lost.

The **Apply** button, when pressed, will apply all changes. The dialog will remain up for further editing.

Pressing the **Help** button will invoke the Acrobat Reader and this documentation will be displayed.

The Accessors tab shown below lists of all the accessor methods defined for the property. You must click on the name of an accessor method from this list in order to show information about the particular accessor method.



Value

Release
Intermediate: ^VESob07k
Version: 24
Last Compile: 06/22/00 12:57 PM

Debug
Intermediate: ^VESob0G0
Version: 24
Last Compile: 06/22/00 12:57 PM

Last Modified: 06/22/00 12:57 PM

Name of M routine containing release compiled code for the method.

Date and time of last release compile.

Name of M routine containing debug compiled code for the accessor.

Date and time of last debug compile.

Identifies the accessor the property sheet is currently displaying.

Identifies the currently compiled release version of the accessor.

Identifies the current debug compiled version of the accessor.

Date and time the accessor source code was last saved after being modified.

The **OK** button, when pressed, will save all changes made to the methods property.

The **Cancel** button, when pressed will cancel the property edit session. All changes will be lost.

The **Apply** button, when pressed, will apply all changes. The dialog will remain up for further editing.

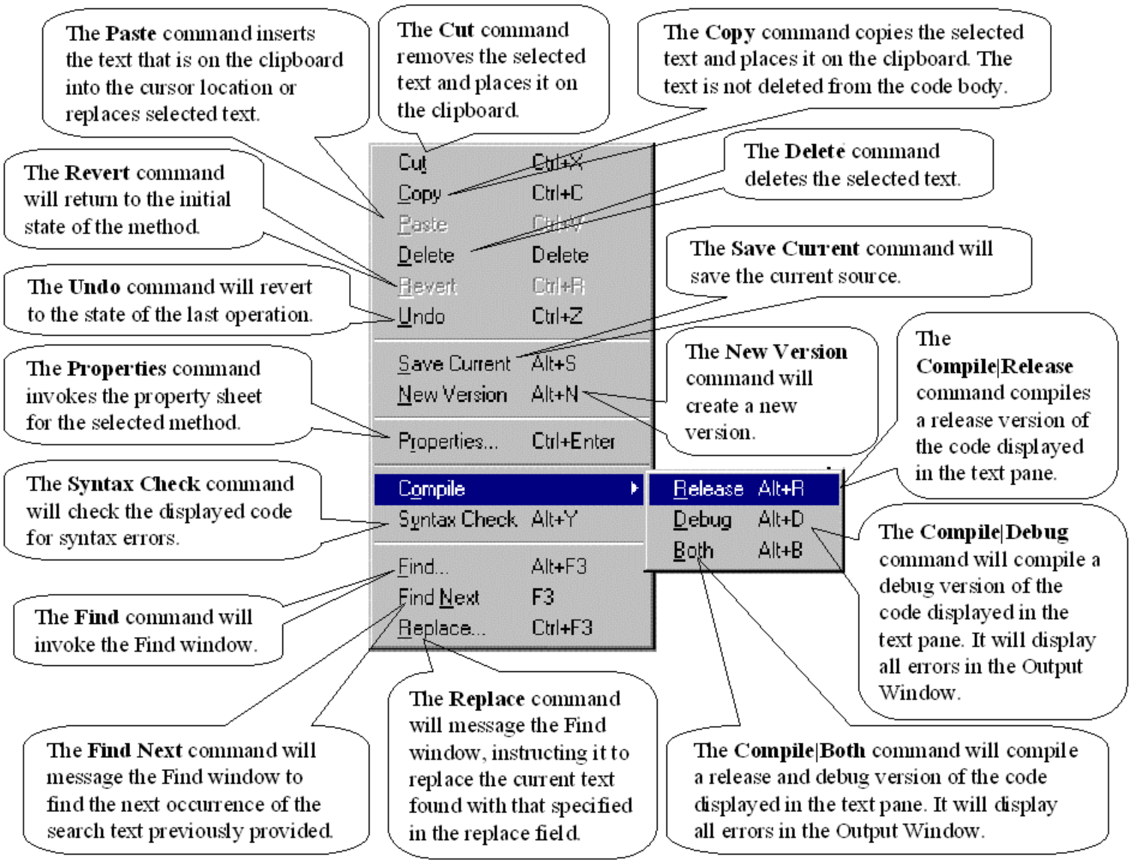
Pressing the **Help** button will invoke the Acrobat Reader and this documentation will be displayed.

Property Menus

Access to Property functionality is available through the Main Menu as well as popup menus within each windowpane and the Accessor popup menus. The popup menu commands are explained below.

Source Code Popup Menu

The Source Code popup menu is invoked by right clicking inside the source code pane of the Property Editor window or pressing **Shift+F10** key combination.



The **Paste** command inserts the text that is on the clipboard into the cursor location or replaces selected text.

The **Cut** command removes the selected text and places it on the clipboard.

The **Copy** command copies the selected text and places it on the clipboard. The text is not deleted from the code body.

The **Revert** command will return to the initial state of the method.

The **Delete** command deletes the selected text.

The **Undo** command will revert to the state of the last operation.

The **Save Current** command will save the current source.

The **Properties** command invokes the property sheet for the selected method.

The **New Version** command will create a new version.

The **Compile|Release** command compiles a release version of the code displayed in the text pane.

The **Syntax Check** command will check the displayed code for syntax errors.

The **Compile|Debug** command will compile a debug version of the code displayed in the text pane. It will display all errors in the Output Window.

The **Find** command will invoke the Find window.

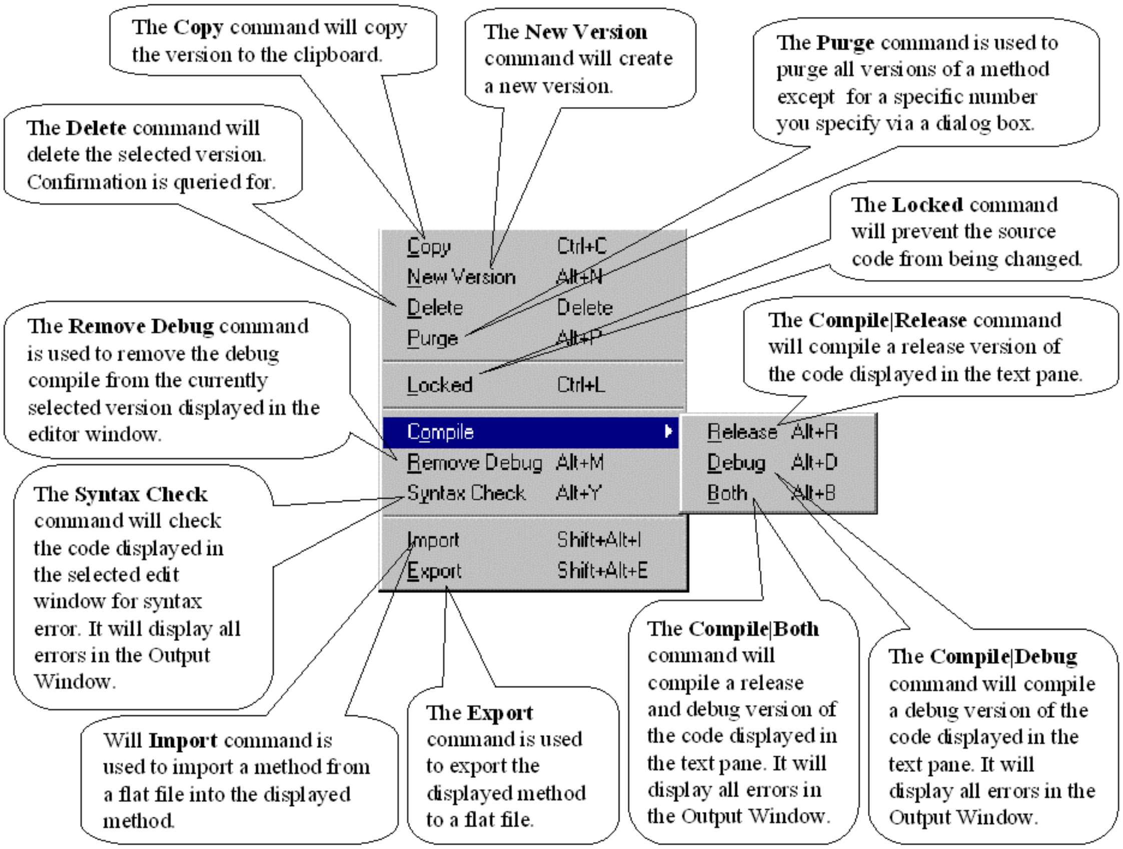
The **Find Next** command will message the Find window to find the next occurrence of the search text previously provided.

The **Replace** command will message the Find window, instructing it to replace the current text found with that specified in the replace field.

The **Compile|Both** command will compile a release and debug version of the code displayed in the text pane. It will display all errors in the Output Window.

Version Popup Menu

The Version popup menu is invoked by right clicking inside the version history pane of the Property Editor Window or pressing the **Shift+F10** key combination.



The **Copy** command will copy the version to the clipboard.

The **New Version** command will create a new version.

The **Purge** command is used to purge all versions of a method except for a specific number you specify via a dialog box.

The **Delete** command will delete the selected version. Confirmation is queried for.

The **Locked** command will prevent the source code from being changed.

The **Remove Debug** command is used to remove the debug compile from the currently selected version displayed in the editor window.

The **Compile|Release** command will compile a release version of the code displayed in the text pane.

The **Syntax Check** command will check the code displayed in the selected edit window for syntax error. It will display all errors in the Output Window.

The **Compile|Both** command will compile a release and debug version of the code displayed in the text pane. It will display all errors in the Output Window.

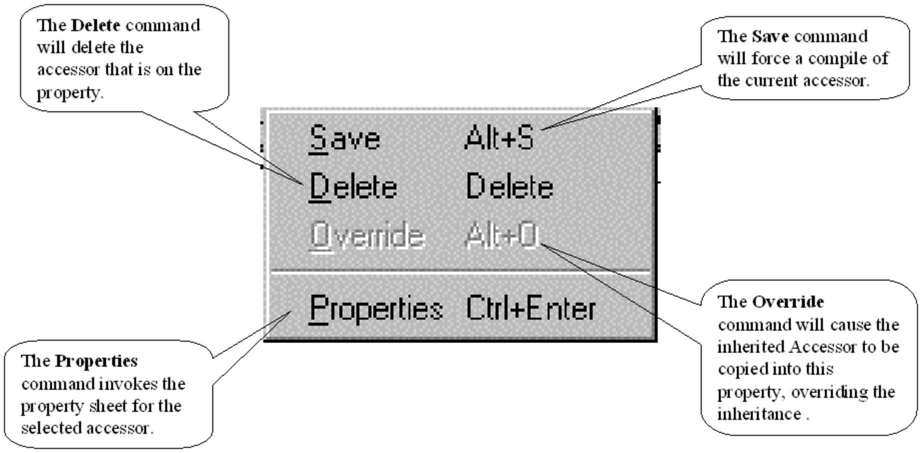
The **Compile|Debug** command will compile a debug version of the code displayed in the text pane. It will display all errors in the Output Window.

Will **Import** command is used to import a method from a flat file into the displayed method.

The **Export** command is used to export the displayed method to a flat file.

Accessor Tab Popup Menu

The Accessor tab popup menu is invoked by right clicking on an accessor tab in the Property Editor or pressing **Shift+F10** key combination.



The **Delete** command will delete the accessor that is on the property.

The **Save** command will force a compile of the current accessor.

The **Properties** command invokes the property sheet for the selected accessor.

The **Override** command will cause the inherited Accessor to be copied into this property, overriding the inheritance .

Using the Property Editor

Creating a Property

The illustration below represents the EsiObjects Session Browser. It exists as a tab sheet in the Session Browser window.

To create a Property, follow the steps below.

1) In the Session Browser, expand the class to which you wish to add a property by clicking on the expansion box (box with the + in it).
2) Now expand the interface that will hold the property. This will display all exiting services as a sub-tree to the interface.
3) Click the right mouse button on the interface name to bring up the popup menu. Select the **Add** option. (Note that you also could press the **Insert** key to add an item.)

4) The Add to Interface dialog appears. Enter the name of the property. Valid names must begin with an alpha character and can contain up to 32 alphanumeric characters. Make the name something that gives a sense of what behavior the method provides.
5) Pull down the combo box and select Property from the list.
6) Click on the OK button to add the property to the interface.

Editing a Property

Once a property has been added to the interface, you can manipulate it in many ways. By selecting the property and right clicking on it, a popup menu is displayed that allows you to delete, edit, and rename the method among other operations. Note that each operation has a keyboard equivalent that will invoke the action directly.

To edit a property, perform the following steps:

1) Select the property to edit. As described above, clicking the right mouse button on the property name will display a popup menu from which you can select the Edit option.
2) The property editor will appear in the client area of the Main Window. It will contain the source code to be edited. You can now create or modify the code. You can also create new versions as well as compile and/or syntax check the code. If you want to create or modify documentation on the property, make sure the Documentation Window is active. Simply click in the Documentation Window and start typing the text. If the Documentation Toolbar is not active, chose the **View|Toolbars|Documentation** to activate it.
3) Pressing the Enter key or double-clicking on the property name in the interface will also invoke the editor.

Deleting a Property

To delete a property, follow these steps:

4) In the Session Browser select the property that you want to delete.
5) Right click on the property icon and select the Delete command from the menu.
6) A verification dialog will ask you if you want to continue. Answer **Yes**. At this point the property will be deleted and the library structure will readjust to the deletion.

Reusing the Property Editor Window

User Option Preference

The User Option Preference tab sheet contains a check box call Redisplay. When this box is checked, EsiObjects will always look of a property editor in the client area before creating a new one. If one exists, it will reuse that Property Editor Window, displaying the newly selected property code in that window. If the property in that window had changes made to it, the window will let you save it before proceeding.

Drag-and-Drop

Another approach to reusing a Property Editor window that is already displayed in the client area is to drag-and-drop the property name into an open editor as outlined below:

1) There must be a Property Editor already open in the client area. And some portion of the code pane must be visible.
2) Select the property to edit from the Session Browser, hold down the left mouse button on the item name and drag the cursor to the editor's code pane.
3) Note that the cursor will indicate when a valid area to drop the item is reached by changing to an cursor arrow with a plus box attached to it. When the cursor shows this indication, lift the left mouse button to drop the property.
4) The context is switched to the Property Editor context.
5) If the original property being edited has been modified without being saved, you will be prompted to save the changes prior to the context being switched. Click **Save** to save the changes, **Discard** to throw away the changes, or **Cancel** to cancel the drop operation.

Editing Property Properties

Properties of a property can be edited using the Properties Property sheets. There are at least 3 ways to invoke the Properties Property sheet:

1) From within the Session Browser, select the appropriate property and press **Ctrl+Enter**.
2) From within the Session Browser, select the appropriate property and invoke the popup menu by right clicking on the name or icon, or by pressing **Shift+F10**. Choose the **Properties** menu item.
3) From within the property editor, select **Properties** from the **Edit** menu.
4) Once the property sheet is displayed in the client area, you can edit those fields that are changeable.

See the Property Properties under Property and Accessor Properties section above for a complete description of all the fields.

Managing Source Versions

Explicit Source Management

When saving the source code of a property (using the source code popup menu) there are four distinct options related to compiling and managing source code versions:

1) **Save Current** Save the source code under the version number currently being edited.
2) **New Version** Save the source code under a new (highest) version number.

3) **Syntax Check** Check the syntax of the current source text.

4) **Compile** Compile the current source text.

Source Code User Options

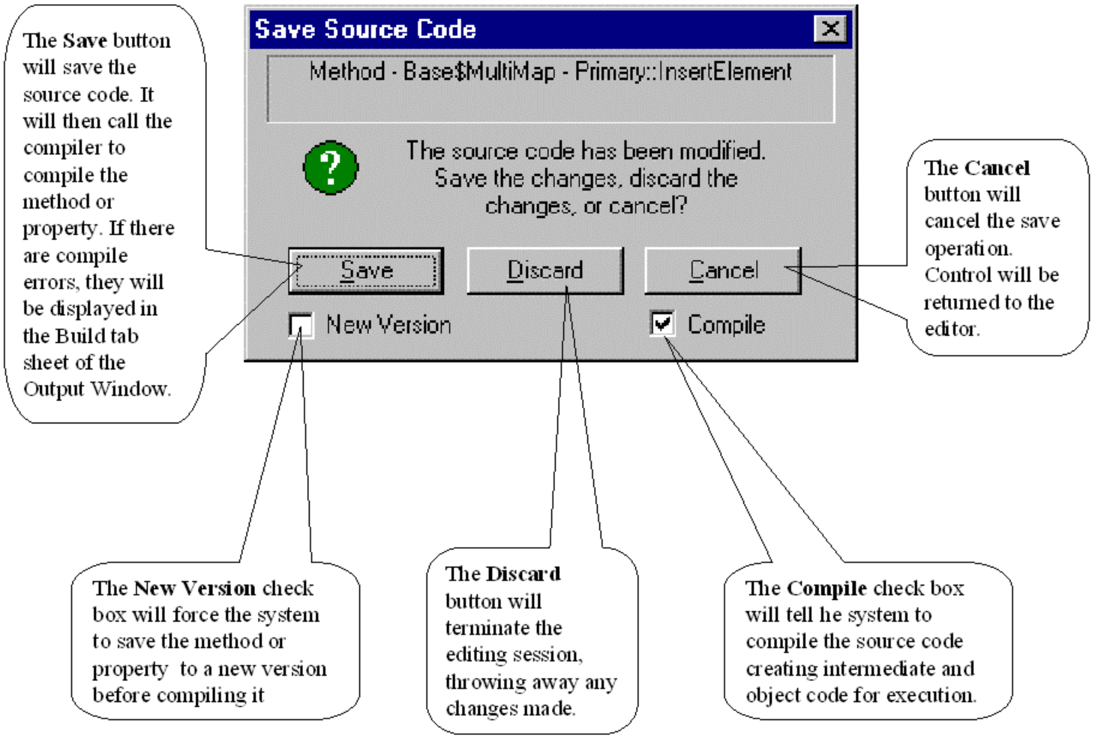
In the EsiObjects User Options, there are two specific options related to source code version control:

Auto New Version When prompted to save source code, the **New Version** check box on the Save dialog will default to what is set here. When checked, the default action on saving source code will be to create a new version of the source code.

Compile On Save When prompted to save source code, the **Compile** check box on the Save dialog will default to what is set here. When checked, the default action on saving source code will be to compile the source code after saving.

Default Save Options

If the user closes a source code object without saving changes, then the Save Source Code dialog shown below will be displayed.



Save Source Code

Method - Base$MultiMap - Primary::InsertElement

The source code has been modified. Save the changes, discard the changes, or cancel?

□ New Version

☑ Compile

The **Save** button will save the source code. It will then call the compiler to compile the method or property. If there are compile errors, they will be displayed in the Build tab sheet of the Output Window.

The **Cancel** button will cancel the save operation. Control will be returned to the editor.

The **New Version** check box will force the system to save the method or property to a new version before compiling it

The **Discard** button will terminate the editing session, throwing away any changes made.

The **Compile** check box will tell he system to compile the source code creating intermediate and object code for execution.

The options that are selected, by default, will be determined by the appropriate user options.

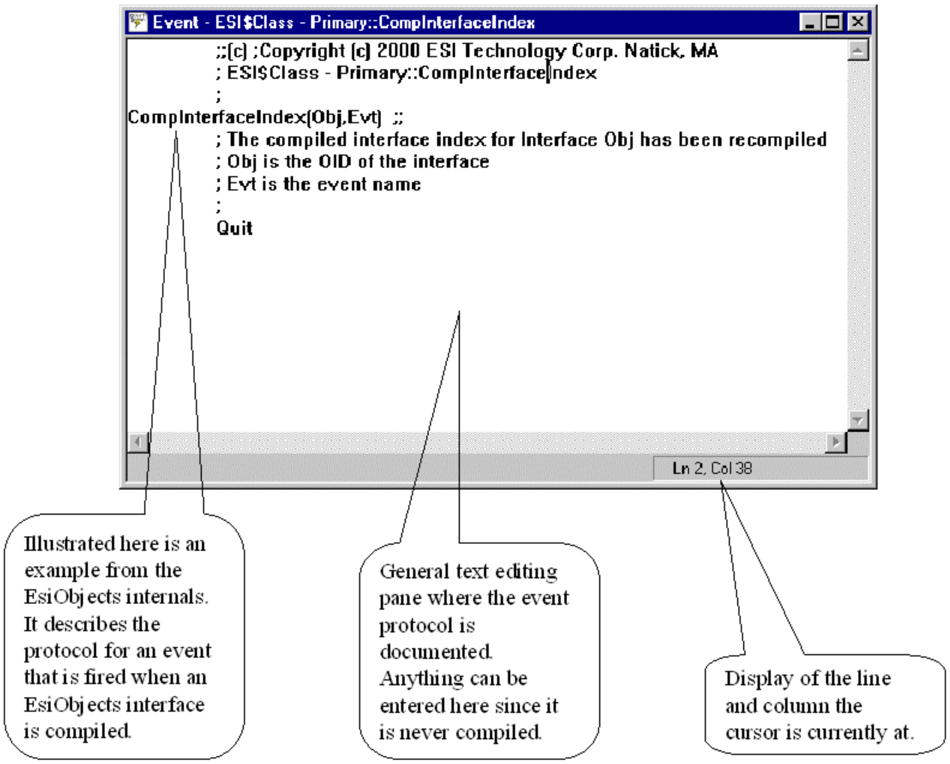
Event Template Editor

Event Template Editor Explained

Event Template Editor Window

The **Event Template Editor** is different from the other editors (method, property, and variable) in that the information entered here is not used at any time. It is used primarily for documentation purposes. But this does not lessen the importance of this editor and the need to use it whenever an object triggers an event.

The Event Template Editor is illustrated below. Each component is explained.



```
          ;;(c) ;Copyright (c) 2000 ESI Technology Corp. Natick, MA
          ; ESI$Class - Primary::CompInterfaceIndex
          ;
CompInterfaceIndex(Obj,Evt)  ;;
          ; The compiled interface index for Interface Obj has been recompiled
          ; Obj is the OID of the interface
          ; Evt is the event name
          ;
          Quit
```

Illustrated here is an example from the EsiObjects internals. It describes the protocol for an event that is fired when an EsiObjects interface is compiled.

General text editing pane where the event protocol is documented. Anything can be entered here since it is never compiled.

Display of the line and column the cursor is currently at.

In EsiObjects, events are used throughout the system. For example, the modification of a property will automatically generate an event so that other processes using that property may be notified of the event. They must be watching for that event to receive the notification.

Any object developed using EsiObjects may throw an event when some condition must be broadcast. This is accomplished via the **Event** command. The EsiObjects Language

Reference and Programmers Reference Guides discuss the **Event, Watch and Ignore** commands and event processing in EsiObjects.

The Session Browser allows you to add events to an interface. The Event Template Editor is used to enter the handler prototype that the event requires. Therefore, anytime you use the **Event** command in a method, make sure that the event is added via the Session Browser (to document the fact that an event of that name is thrown by the object) and the necessary protocol is specified via this editor.

If a user of the class wishes to hook a handler to the event, they merely invoke this editor to access the template. They can then copy the template code to the handler method to insure that their handler has the proper protocol in place.

Event Template Menus

Template Popup Menu

The Event Template popup menu is invoked by right clicking in the Event Template Editor or pressing **Shift+F10** key combination.

Using the Event Template Editor

Creating an Event Template

To create a method, property or event template, follow the steps below.

1) In the Session Browser, expand the class to which you wish to add an event template by clicking on the expansion box (box with the + in it).
2) Now expand the interface that will hold the event. This will display all exiting services as a sub-tree to the interface.
3) Click the right mouse button on the interface name to bring up the popup menu. Select the Add option. (Note that you also could press the Insert key to add a service.)
4) The **Add to Interface** dialog appears. Enter the name of the event. Valid names must begin with an alpha character and can contain up to 32 alphanumeric characters. Make the name something that gives a sense of what event within the object can be hooked to.
5) Pull down the combo box and select Event from the list.
6) Click on the OK button. The event template will be added to the interface.

Editing an Event Template

Once an event has been added to the interface, you can manipulate it in many ways. By selecting the event and right clicking on it, a popup menu is displayed that allows you to delete, edit, and rename the event among other operations. Note that each operation has a keyboard equivalent that will invoke the action directly.

To edit an event:

1) Select the event to edit. As described above, clicking the right mouse button on the event name will display a popup menu from which you can select the Edit option.
2) The Event Template Editor will be displayed in the Main Window client area for the event. You can now create or modify the documentation for the event. *Please note that the Event Template is just that, it is a template for the event. Defining it in the interface of the class has significance, however, what is entered in the text area of the editor has only documentation significance. Use it to document the event protocol (The formal structure of the Input Specification of the method that will receive the callback when an event is fired).*
3) Additionally, pressing the Enter key or double-clicking on the item will also invoke the editor.

Using Drag-and-Drop to Edit an Service

Each time you edit an event, a new editor may be created for the item. To edit an event without creating a new editor, drag-and-drop the item to an open editor as outlined below:

1) There must be an Event Template Editor already open and some portion of the code panel must be visible somewhere on the desktop.
2) Select the item to edit from the Session Browser, hold down the left mouse button on the item name and drag the cursor to the editor's text area
3) Note that the cursor will indicate when a valid area to drop the item is reached by displaying a cursor with a plus character in a box. When the cursor shows this indication, lift the left mouse button to drop the event.
4) The context of the browser is switched to the dropped event.
5) If the original event being browsed had been modified without being saved, you will be prompted to save the changes prior to the editor context being switched.

Deleting an Event

To delete an event template, follow the steps below.

1) In the Session Browser, expand the class to the event template by clicking on the expansion box (box with the + in it).
2) Click the right mouse button on the event template name to bring up the popup menu. Execute the **Delete** command. (Note that you also could press the **Del** key to add a service.)
3) The **Delete** validation dialog will appear. Click the **Yes** button to delete the event template of **No** to abort the delete.

Relationship Wizard

Relationship Wizard Explained

EsiObjects provides Method, Properties and Events services within any of its interfaces. In keeping with the philosophy to hide complexity and provide you, the programmer, with tools that eliminate redundant work, EsiObjects also contains a wizard to assist you in creating a relationship between two classes. Additionally, the runtime component of EsiObjects takes over the responsibility of maintaining the relationship.

A **binary relationship** is an association between two classes, a **source class** and a **target class**. A binary relationship may view as an attribute of its source class. Within a relationship, an object of the source class may be associated with zero or more objects of the target class. A source class may contain multiple relationships to the same or different target classes. For a particular relationship, there is usually a corresponding **inverse** relationship from the target class back to the source class.

A relationship may have a **cardinality** of "one" or "many". In the former case, an object of the source class may be associated with at most one object of the target class. In the latter case, an object of the source class may be associated with a collection of objects in the target class.

The specific object-to-object mappings of a relationship are established dynamically. There are two rules that the associations must obey:

- Referential integrity must be maintained. This means that when an object in a relationship's target class is deleted, objects in the source class can no longer maintain their associations with it.
- A relationship and its inverse (when the inverse exists) must map consistently relative to each other.

EsiObjects treats relationships as objects belonging to the class ESI$Relationship.

Using the Relationships Wizard

Creating a Relationship

To create a relationship, you must select an interface of the source class and invoke the "Add to Interface" dialog using the Interface popup menu. If you choose "Relationship" as the kind of item to add to the given interface, then a new wizard (sequence of dialogs) guides you through setting up the relationship. We use a wizard because relationships require this information at creation time, and we do not expect you to have to pull up a Properties dialog sheet to set it.

You must specify the following information when creating a relationship:

- **Target class**. The name of the relationship's target class. This is an editable field. (A dropdown selection box allows existing classes to be chosen. A separate button allows the user to change the library from which classes may be selected.)
- **Cardinality**. Selected as a radio button, either "One" or "Many". Default is "One".
- **Public flag**. Whether the relationship is accessible from classes outside the source class. Selected as a checkbox. Enabled by default.
- **Inheritable flag**. Whether the relationship may be inherited by subclasses of the source class. Selected as a checkbox. Enabled by default.

You may specify the following optional information, also within the wizard, when creating a relationship:

- **Shape**. For a relationship of cardinality "many", the type of collection (e.g. Set, List, Array, Bag) of objects in the target class to which an object in the source class will be associated. Default is Set. (A dropdown selection box allows you to select from among the subclasses of Base$Collection.)
- **Inverse**. The name of the inverse relationship in the target class. This may be left blank, as it is not required that the inverse relationship be defined. This is an editable field, which may also be modified by a drag and drop operation. You also need to identify the interface to which the inverse relationship belongs, with a default of Primary.

The relationship creation wizard will offer you a choice of options for the inverse relationship: **None**, **New**, or **Existing**. If you select New, the wizard would step you through the creation of a new inverse relationship. If you select Existing, the wizard would let you choose the inverse from a list of existing relationships mapping from the target class to the source class. This list includes only relationships with no inverses.

Editing a Relationship

Unlike the other kinds of items in a class' interface, a relationship contains no editable code and therefore has no editor in EsiObjects. It does have a property sheet, which in addition to the above information also contains:

- **Source class**. The name of the source class that the relationship belongs to. It may not be changed.
- **Interface**. The name of the interface in which the relationship is defined. It may not be changed.
- **Name**. The name of the relationship. Editable (as with property sheets for methods, properties and events).

Deleting a Relationship

There are three ways to delete a relationship. From within the Session Browser, select the relationship by clicking on its icon or name then:

- Press the **Del** key.
- Pull down the Main Edit Menu and select the **Delete** command.
- Invoke the popup menu by right clicking on it (or by pressing **Shift+F10)**. Choose the **Delete** command.

Promote, Override and Copy Command Not Supported

Methods, properties and events are class services. In the case of methods and properties, they are simply code bodies. An event is simply a definition. These services are available at the definitional level. When invoked, they simply execute within the context of an instance. They do are not associated with instance structures.

However, relationships are actually structures that exist between two objects. Once instantiated, they are typically immutable during their lifetime. Like most features of object orientation, they are dependent upon their definitions being unaltered.

As a consequence of the immutability of relationships, operations like Promote, Override and Copy are not permitted as they are for the other services.

Search and Edit

Application libraries often contain many classes, and in many cases, the application will contain numerous libraries. As the application increases in size, it becomes harder for the programmer to keep track of all the classes and services they contain. EsiObjects itself consists of two large libraries of classes, most of which are reusable by the application.

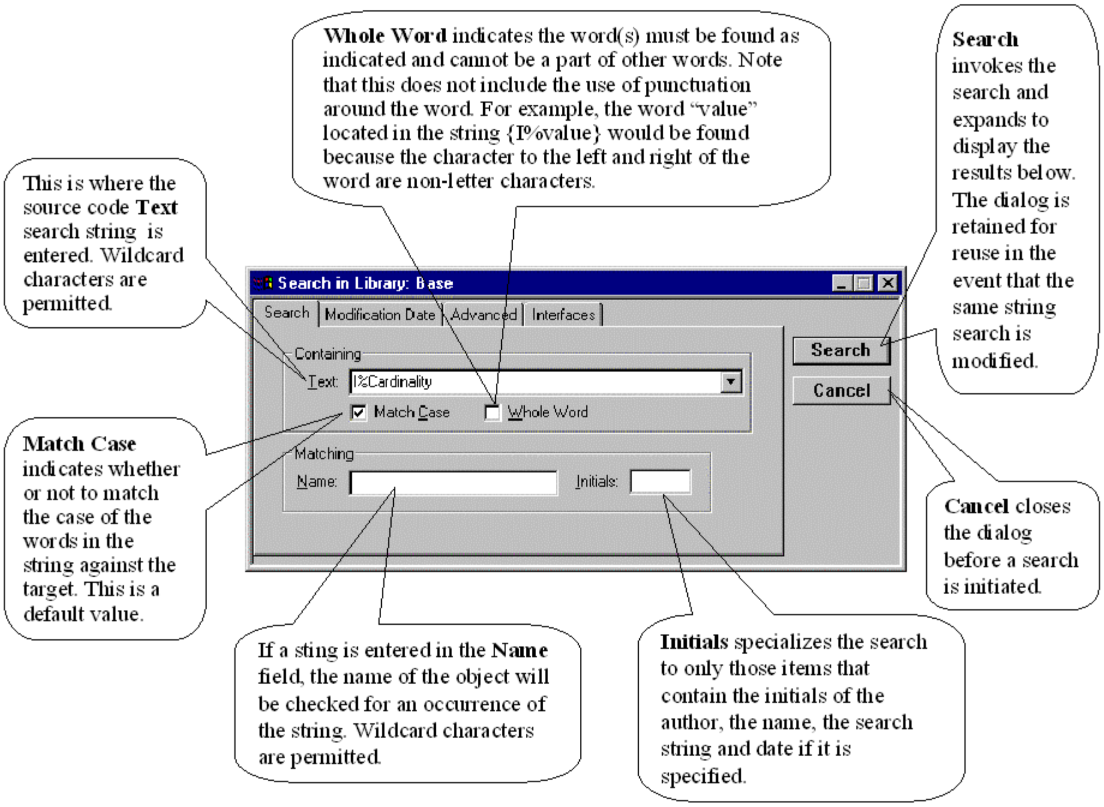
Search and Edit Explained

EsiObjects provides a powerful search engine for searching within a particular library. Once the objects are found based on the criteria specified, they are displayed as pointers in a record oriented window in the same format that folders display their contents.

As explained in the section Session Structures section above, a session consists of two types of structures: the library structure and the folder structure. The library structure contains hierarchical class structures and is used to partition the classes logically, according to some application requirement. Folders contain pointers to definitional objects within a library. Folders are used to condense a disparate set of definitional objects into one window list for easy access.

The Search tool is a useful programming tool that allows you to search in EsiObjects libraries for occurrences of specific strings and associated information such as the author’s initials and date ranges. In general, this tool is used to search for string occurrences that are numerous and may be disseminated throughout a library. This can save a significant amount of time as the programmer may want to modify, replace and/or remove selected strings and associated information.

The Search tool of EsiObjects lets you select the search range by first pointing to a component using the Session Browser (the range). There are three levels within the library tree that constitute a search range: the entire library, a specific class or a specific interface within a class. Selecting the appropriate level within the tree and then executing either the popup menu **Search** command (or the main menu **Tools|Search|Selected** command) will bring up the **Search In** form (shown below). Within these permitted search ranges, you can also specialize the search to a service source code body and/or the name of the service by checking the appropriate check boxes (**Source Code** and **Name**) on the **Search In** form.



Whole Word indicates the word(s) must be found as indicated and cannot be a part of other words. Note that this does not include the use of punctuation around the word. For example, the word "value" located in the string {I%value} would be found because the character to the left and right of the word are non-letter characters.

Search invokes the search and expands to display the results below. The dialog is retained for reuse in the event that the same string search is modified.

This is where the source code **Text** search string is entered. Wildcard characters are permitted.

Match Case indicates whether or not to match the case of the words in the string against the target. This is a default value.

If a sting is entered in the **Name** field, the name of the object will be checked for an occurrence of the string. Wildcard characters are permitted.

Initials specializes the search to only those items that contain the initials of the author, the name, the search string and date if it is specified.

Cancel closes the dialog before a search is initiated.

Search in Library: Base

Containing

Text: I%Cardinality

☑ Match Case □ Whole Word

Matching

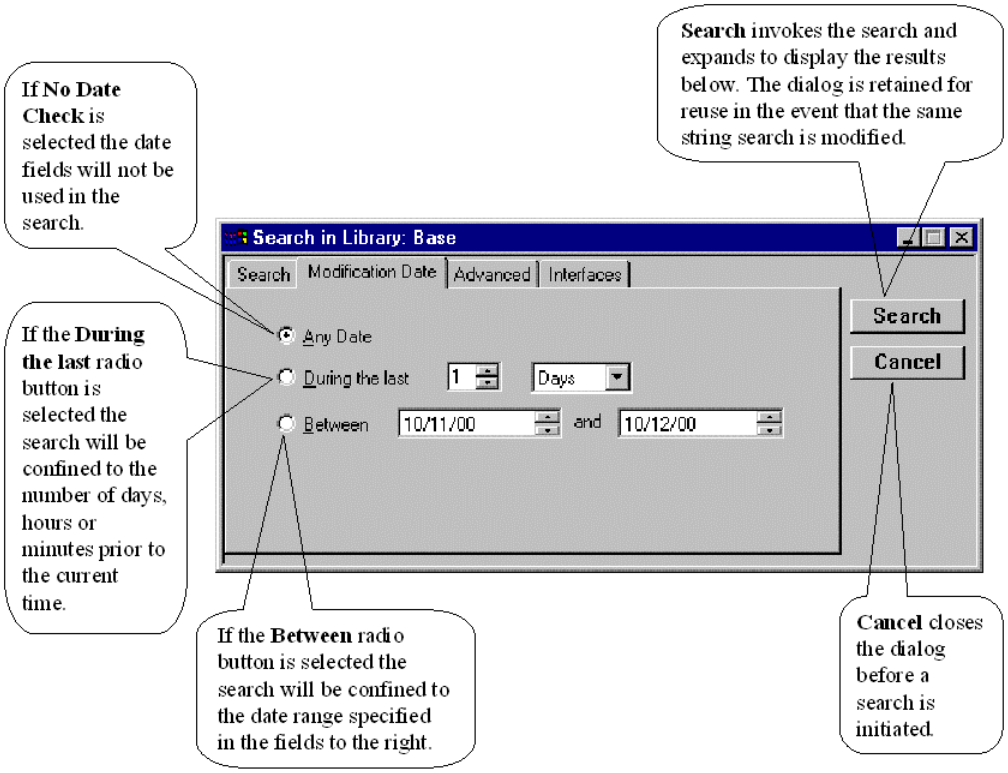
Name:

Initials:

The first tab sheet shown below, **Search,** contains two groupings of fields, that is, **Containing** and **Matching**. Each group of fields lets you enter specific search criteria. The **Text** field lets you enter a sting of characters to search for. If specified, this search will be confined to the source code of those services specified on the Advanced tab sheet. The **Matching** group contains two fields. The **Initials** field, if specified, specializes the search to the initials of the programmer that are associated with the specified code body. The **Name** field, if specified, specializes the search to the name of the object being searched.

The **Text Match** string search can be modified to either search for the literal specification of the string by checking the **Match Case** check box or you can force a search independent of case by clearing the check box. Additionally, you can search for a whole word by checking the **Whole Word** check box. If you clear the check box, the search will treat the search string as a partial string. A whole word search has a specialized meaning. Read the description in the illustration above.

The second tab sheet shown below, **Modification Date,** contains three radio buttons. They let you ignore the date check, specify a date range (the **Between** radio button) or specify a number of days, hours or minutes prior to the current time (the **During the last** button). If you specify a range, it will specialize the search to that date range.



● Any Date

○ During the last 1 Days

○ Between 10/11/00 and 10/12/00

If **No Date Check** is selected the date fields will not be used in the search.

Search invokes the search and expands to display the results below. The dialog is retained for reuse in the event that the same string search is modified.

If the **During the last** radio button is selected the search will be confined to the number of days, hours or minutes prior to the current time.

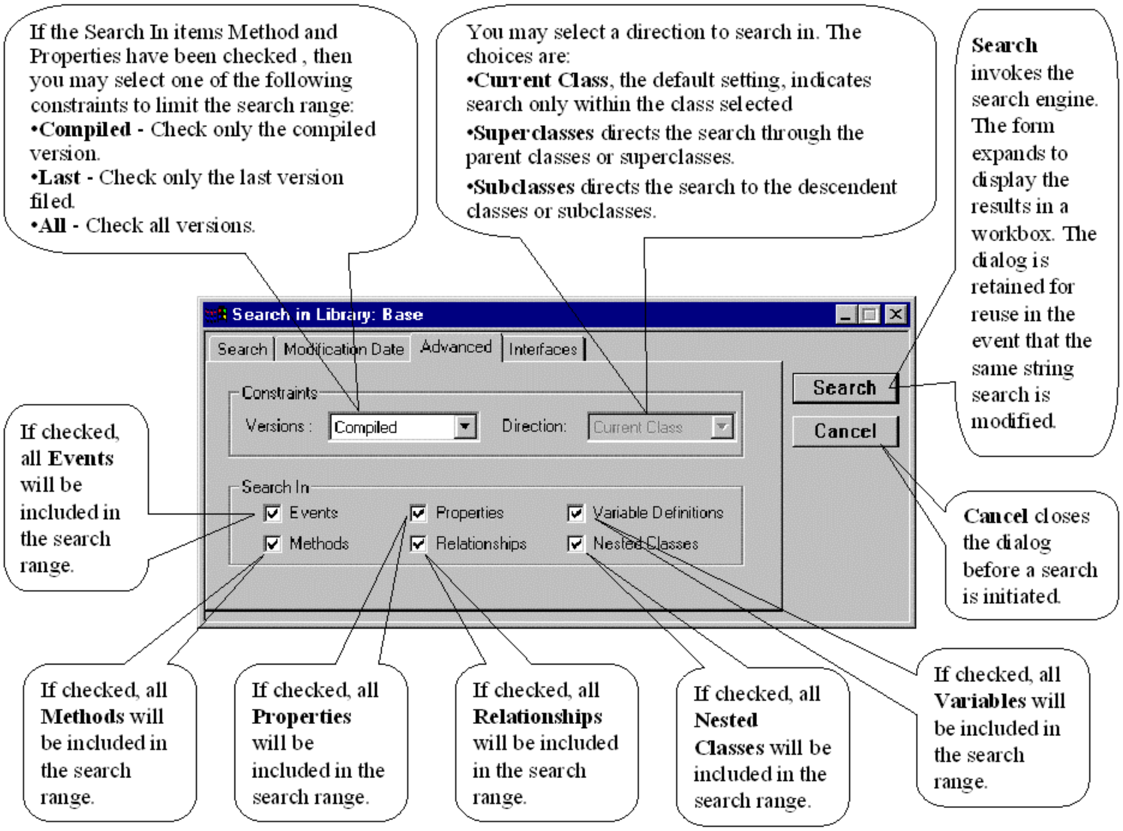
If the **Between** radio button is selected the search will be confined to the date range specified in the fields to the right.

Cancel closes the dialog before a search is initiated.

The important thing to remember is that, if specified, these four search criteria form a logical *and* at search time. That is, if you specify the Text string “I%Cardinality”, the Initials “JAM”, a name “List” and a date range of 20 days, the search engine will find and report all objects that contain the specified source code text string *and* initials *and* name *and* in the date range specified. This lets you narrow the search down to only those objects that meet the criteria.

The diagram below illustrates and explains all the fields that let you specify the search range specialization and search criteria.

The third tab sheet shown below, **Advanced,** allows you to further specialize the search range to components contained in a class. Within the **Search In** group, the **Events, Methods, Properties, Relationships, Variable Definitions** and **Nested Classes** check boxes let you limit the search to these particular objects. They are selected by default. Deselecting them will eliminate these items from the search range.



Versions : Compiled
Direction: Current Class

☑ Events
☑ Methods
☑ Properties
☑ Relationships
☑ Variable Definitions
☑ Nested Classes

If the Search In items Method and Properties have been checked , then you may select one of the following constraints to limit the search range:
•**Compiled -** Check only the compiled version.
•**Last -** Check only the last version filed.
•**All -** Check all versions.

You may select a direction to search in. The choices are:
•**Current Class**, the default setting, indicates search only within the class selected
•**Superclasses** directs the search through the parent classes or superclasses.
•**Subclasses** directs the search to the descendent classes or subclasses.

Search invokes the search engine. The form expands to display the results in a workbox. The dialog is retained for reuse in the event that the same string search is modified.

If checked, all **Events** will be included in the search range.

Cancel closes the dialog before a search is initiated.

If checked, all **Methods** will be included in the search range.

If checked, all **Properties** will be included in the search range.

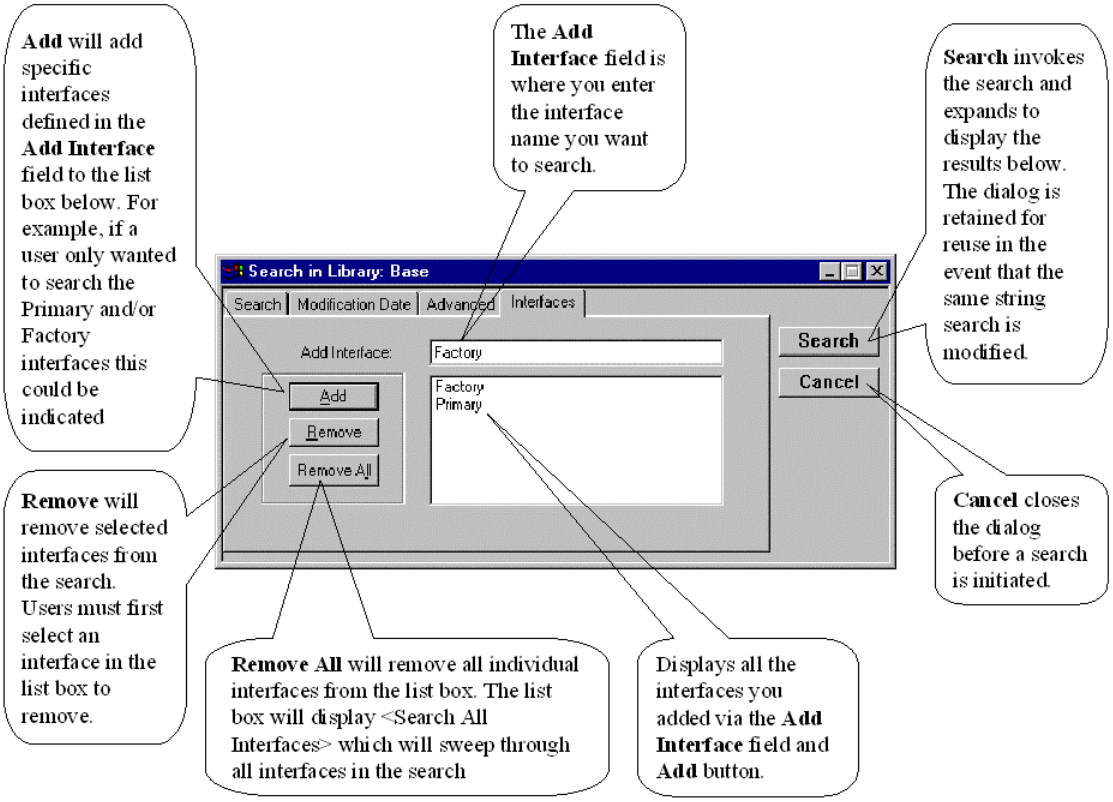
If checked, all **Relationships** will be included in the search range.

If checked, all **Nested Classes** will be included in the search range.

If checked, all **Variables** will be included in the search range.

The **Constraints** section defines global limitations on the search. EsiObjects implements source code versioning. Using the **Versions** combo box, you can select the range of versions to search. Additionally, selecting the direction of the search in the Direction combo box can modify the search range. See the illustration above for details on these fields.

The fourth tab sheet, **Interfaces,** allows you to specialize the search range to a selected set of class interfaces. Selecting from the list box, you can search all interfaces by accepting the default <Search All Interface> or specify each individually in the **Add Interface** field. This tab is only accessible when a class or class library is being searched.



Add will add specific interfaces defined in the **Add Interface** field to the list box below. For example, if a user only wanted to search the Primary and/or Factory interfaces this could be indicated

The **Add Interface** field is where you enter the interface name you want to search.

Search invokes the search and expands to display the results below. The dialog is retained for reuse in the event that the same string search is modified.

Remove will remove selected interfaces from the search. Users must first select an interface in the list box to remove.

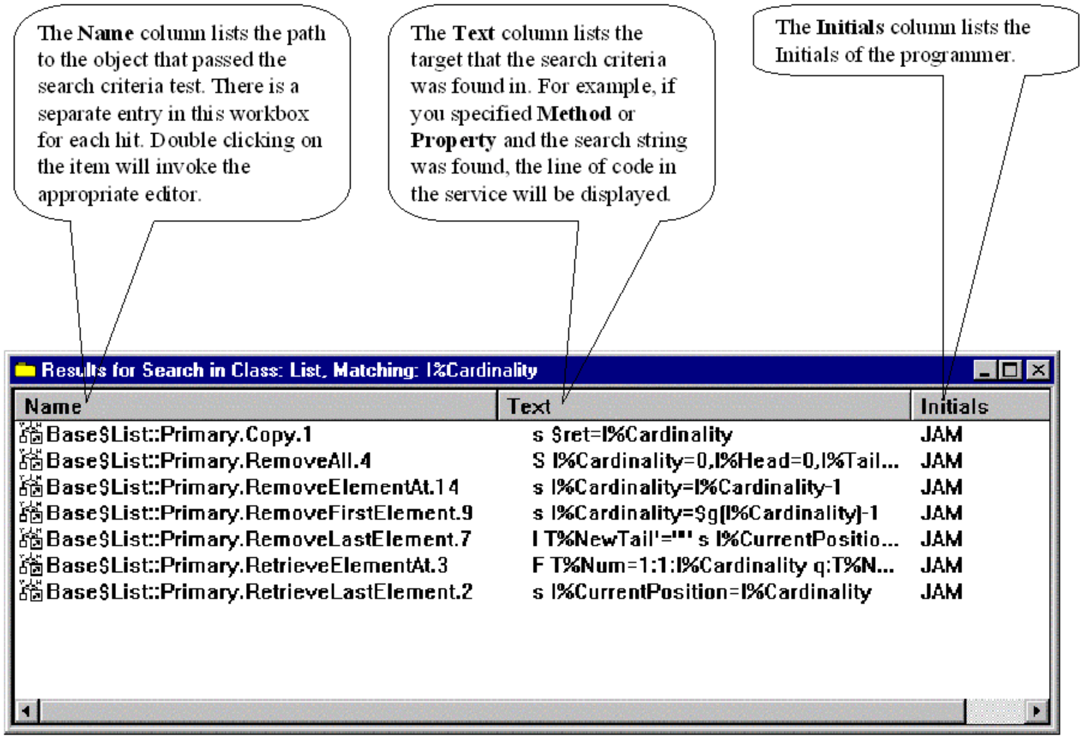
Cancel closes the dialog before a search is initiated.

Remove All will remove all individual interfaces from the list box. The list box will display <Search All Interfaces> which will sweep through all interfaces in the search

Displays all the interfaces you added via the **Add Interface** field and **Add** button.

Each **Search In** form that is invoked is specific to the level in the library tree that it was selected from as indicated in the form title.

The results of a search are displayed in a detached window. The window is record oriented and is similar to a Folder window. The diagram below illustrates the results of a search in the List class for the instance variable “I%Cardinality” and the initials “JAM”. Each object path listed under the Name column is identical to the path used in a folder window. The commonality is no coincidence. Once found, these objects can be selected and transferred to a folder via drag and drop where you can store their pointers for future work or export. Like folders, the appropriate editor or property sheet for any object in the collection can be invoked by double clicking on the pointer.



The **Name** column lists the path to the object that passed the search criteria test. There is a separate entry in this workbox for each hit. Double clicking on the item will invoke the appropriate editor.

The **Text** column lists the target that the search criteria was found in. For example, if you specified **Method** or **Property** and the search string was found, the line of code in the service will be displayed.

The **Initials** column lists the Initials of the programmer.

Results for Search in Class: List, Matching: I%Cardinality

Name	Text	Initials
Base$List::Primary.Copy.1	s $ret=I%Cardinality	JAM
Base$List::Primary.RemoveAll.4	S I%Cardinality=0,I%Head=0,I%Tail...	JAM
Base$List::Primary.RemoveElementAt.14	s I%Cardinality=I%Cardinality-1	JAM
Base$List::Primary.RemoveFirstElement.9	s I%Cardinality=$g(I%Cardinality)-1	JAM
Base$List::Primary.RemoveLastElement.7	I T%NewTail'="" s I%CurrentPositio...	JAM
Base$List::Primary.RetrieveElementAt.3	F T%Num=1:1:I%Cardinality q:T%N...	JAM
Base$List::Primary.RetrieveLastElement.2	s I%CurrentPosition=I%Cardinality	JAM

Using Search and Edit

Performing the Search

Follow these steps to initiate a search.

1. To begin a search, first open the Session Browser, then highlight a library, class or interface on the tree view in which you want to search.
2. Then right click on an item to invoke the pop-up menu and execute the Search command (or execute the **Tools|Search|Selected** command).
3. The **Search In** form will appear. Specify the search criteria and click on the **Search** button.
4. Once the search has finished and if it found objects conforming to the search criteria, a detached window will appear containing the object pointers found in the selected hierarchy.

Using the Search Results

Once the search has completed, a detached window will appear containing pointers to the definitional object conforming to the search criteria. You can use the results in the following ways:

1. Select all or a part of the entries and drag them to a folder which is persistent. The search window is not, it will be deleted when you shut the EsiObjects client down. This will make them available in future sessions.
2. Select any one of the objects and perform any permitted operation on it indirectly, just like you would as if it were a folder.

Debugging Tools

Interactive Debugger

Because developing software systems is usually a complex process and the mere fact that human beings make mistakes, **Interactive Debuggers** have become an integral part of software development environments.

The fundamental purpose of a debugger is to let the programmer control the execution of a software component so that the context of execution can be observed as each instruction is executed. A good debugger will let the programmer set break points, control the flow of execution in various ways and modify the state (variables) of objects. It will display the sequence of code execution, the variable states at each step and the execution stack contents.

The Interactive Debugger Explained

The EsiObjects Interactive Debugger is normally used when an error terminates the execution of an application and the cause is not obvious. However, debuggers are actually good tools for learning the internals of applications as well. Programmers who are given responsibility for maintaining existing applications find that using the debugger to explore the code and object state is a fast way to learn the application internals.

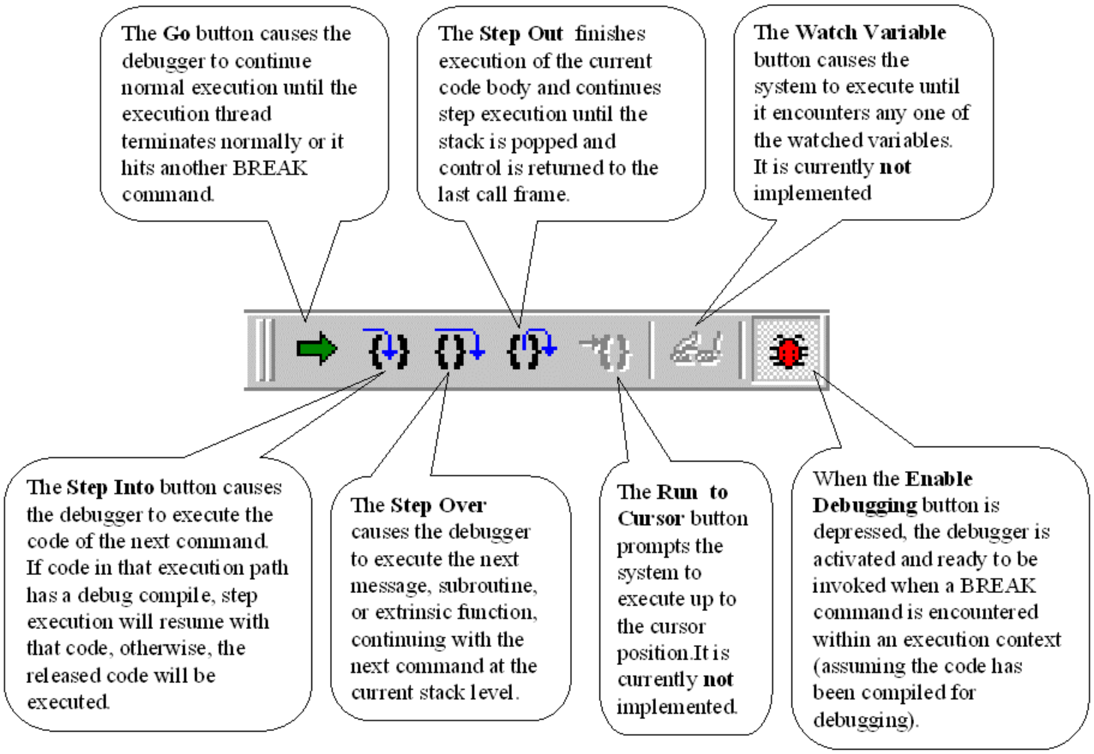
Using a debugger is a habit the programmer must learn. Rather than wasting time looking at the code, activating the debugger and stepping through the execution sequence usually exposes the problem immediately, saving an incredible amount of development time.

Interactive Debugger Window

The Interactive Debugger contains a toolbar and a Debugging Window. The Debugging Window contains a status bar and three major panes. The following sections will describe each component.

Debugger Execution Toolbar

The **Debugger Execution Toolbar** contains buttons that active/inactive the debugger as well as control the execution flow of the debugger. The buttons on the toolbar are explained below.



The **Go** button causes the debugger to continue normal execution until the execution thread terminates normally or it hits another BREAK command.

The **Step Out** finishes execution of the current code body and continues step execution until the stack is popped and control is returned to the last call frame.

The **Watch Variable** button causes the system to execute until it encounters any one of the watched variables. It is currently **not** implemented

The **Step Into** button causes the debugger to execute the code of the next command. If code in that execution path has a debug compile, step execution will resume with that code, otherwise, the released code will be executed.

The **Step Over** causes the debugger to execute the next message, subroutine, or extrinsic function, continuing with the next command at the current stack level.

The **Run to Cursor** button prompts the system to execute up to the cursor position.It is currently **not** implemented.

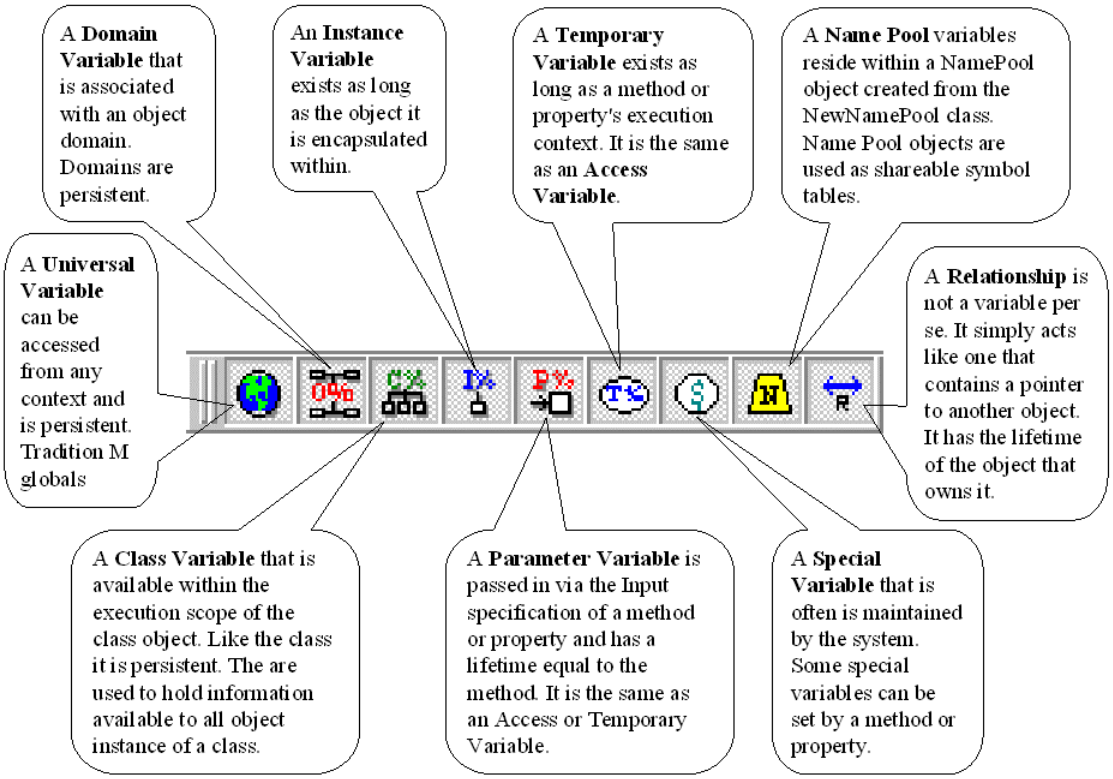
When the **Enable Debugging** button is depressed, the debugger is activated and ready to be invoked when a BREAK command is encountered within an execution context (assuming the code has been compiled for debugging).

Debugger Symbol Toolbar

The **Debugger Symbol Toolbar** contains a button for each symbol scope supported by EsiObjects. Often, when debugging, you are only interested in seeing specific variables. These buttons let you toggle the display of scoped variables on and off. If the button is depressed, the system will display that variable in the Symbols tab sheet of the Debugger Window.

Refer to the Variables section of the EsiObjects Language Reference Guide for more information on symbols and scoping.

The Symbol toolbar is illustrated below. Each button is described.



A **Domain Variable** that is associated with an object domain. Domains are persistent.

An **Instance Variable** exists as long as the object it is encapsulated within.

A **Temporary Variable** exists as long as a method or property's execution context. It is the same as an **Access Variable**.

A **Name Pool** variables reside within a NamePool object created from the NewNamePool class. Name Pool objects are used as shareable symbol tables.

A **Universal Variable** can be accessed from any context and is persistent. Tradition M globals

A **Relationship** is not a variable per se. It simply acts like one that contains a pointer to another object. It has the lifetime of the object that owns it.

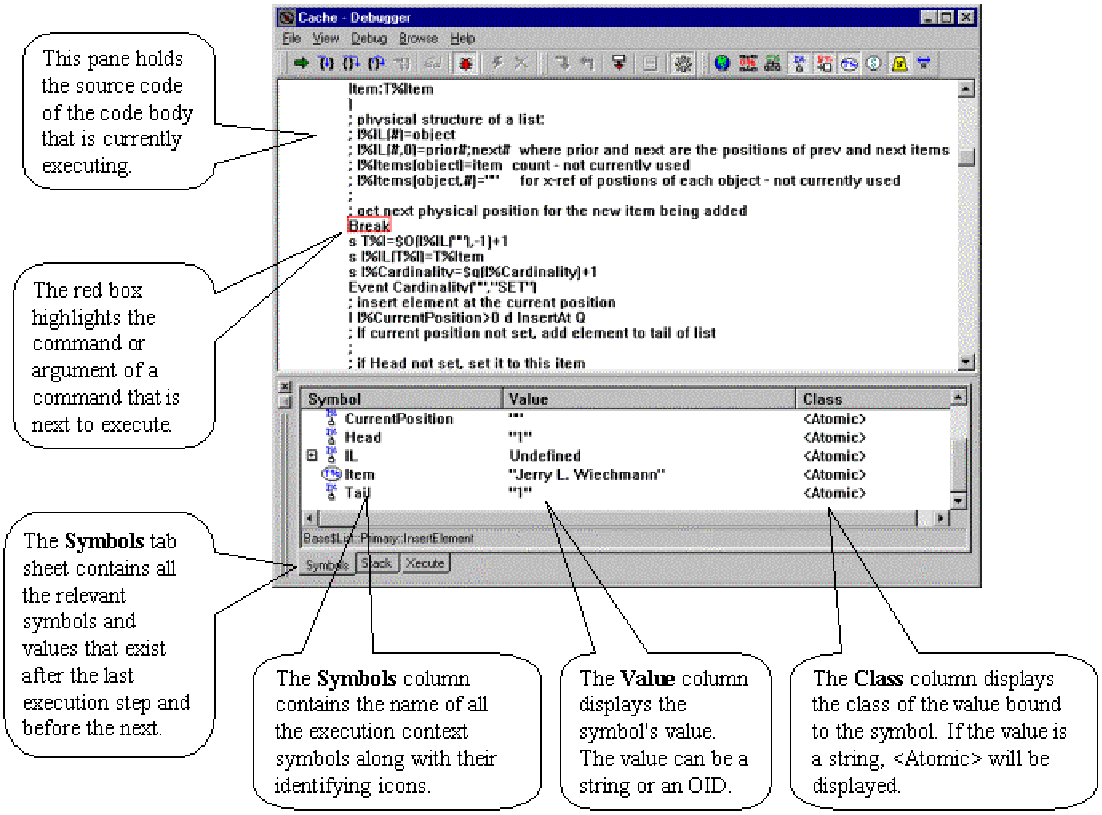
A **Class Variable** that is available within the execution scope of the class object. Like the class it is persistent. The are used to hold information available to all object instance of a class.

A **Parameter Variable** is passed in via the Input specification of a method or property and has a lifetime equal to the method. It is the same as an Access or Temporary Variable.

A **Special Variable** that is often is maintained by the system. Some special variables can be set by a method or property.

Debugger Window with Symbol Display

The Debugger Window is not a part of the EsiObjects Main Window. It is a separate window. It has it’s own menu and is used exclusively for displaying the execution context of a code body that has a debug compile associated with it. The picture below illustrates the Debugger Window with the Symbol tab sheet exposed. The Symbols tab sheet is part of the State window that displays all the relevant symbols and their values after the last execution step.



```
Item:T%Item
]
; physical structure of a list:
; I%IL(#)=object
; I%IL(#,0)=prior#;next#  where prior and next are the positions of prev and next items
; I%Items(object)=item  count - not currently used
; I%Items(object,#)=""     for x-ref of postions of each object - not currently used
;
; get next physical position for the new item being added
Break
s T%I=$O(I%IL(""),-1)+1
s I%IL(T%I)=T%Item
s I%Cardinality=$g(I%Cardinality)+1
Event Cardinality("","SET")
; insert element at the current position
I I%CurrentPosition>0 d InsertAt Q
; If current position not set, add element to tail of list
;
; if Head not set, set it to this item
```

Symbol	Value	Class
CurrentPosition	""	<Atomic>
Head	"1"	<Atomic>
IL	Undefined	<Atomic>
Item	"Jerry L. Wiechmann"	<Atomic>
Tail	"1"	<Atomic>

This pane holds the source code of the code body that is currently executing.

The red box highlights the command or argument of a command that is next to execute.

The **Symbols** tab sheet contains all the relevant symbols and values that exist after the last execution step and before the next.

The **Symbols** column contains the name of all the execution context symbols along with their identifying icons.

The **Value** column displays the symbol's value. The value can be a string or an OID.

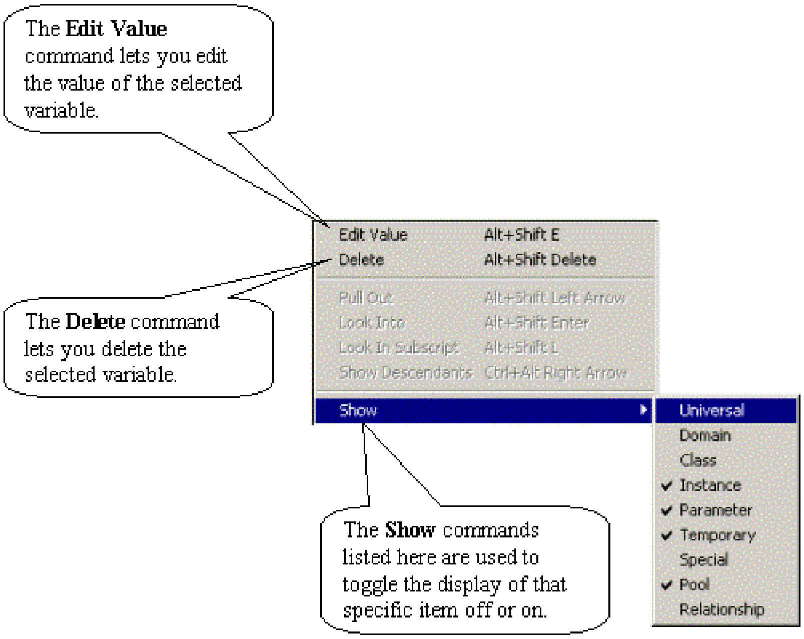
The **Class** column displays the class of the value bound to the symbol. If the value is a string, <Atomic> will be displayed.

Debugger Window Symbol Display Popup Menu

The symbol popup menu is invoked by right clicking on a variable in the Symbol tab sheet. Any variable can have two types of values: **string** or **object**. The menu commands highlighted on the object popup menu are dependent upon what value type is bound to the variable.

String Value Popup Menu

The illustration below describes the string commands you can use to modify the variable or its value.



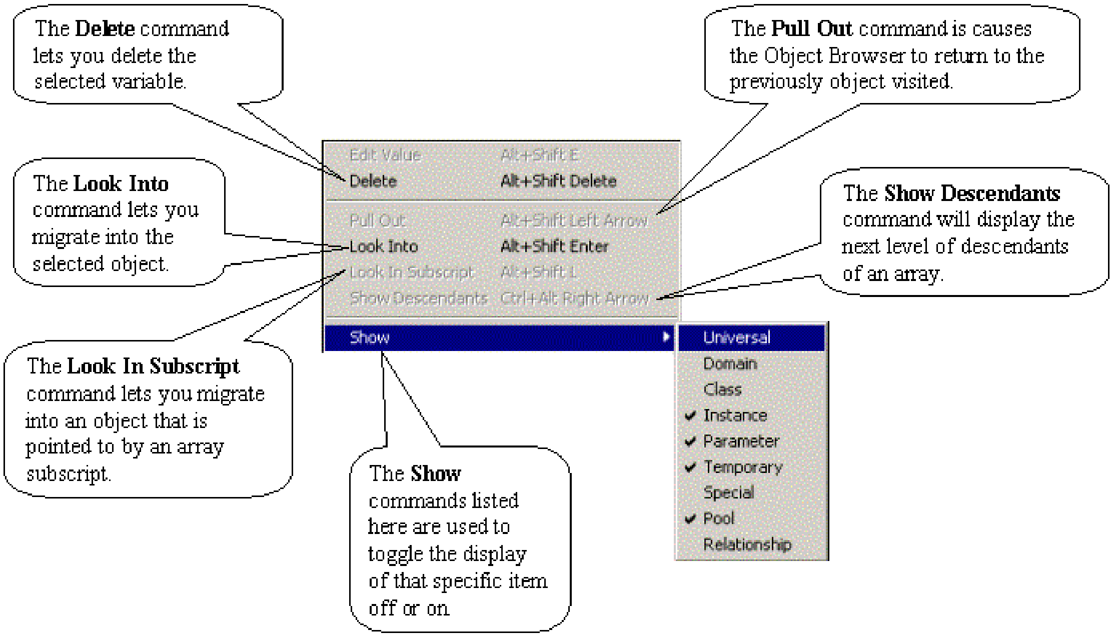
The **Edit Value** command lets you edit the value of the selected variable.

The **Delete** command lets you delete the selected variable.

The **Show** commands listed here are used to toggle the display of that specific item off or on.

Class Value Popup Menu

The illustration below describes the object commands you can use to modify or migrate the object.



The **Delete** command lets you delete the selected variable.

The **Pull Out** command is causes the Object Browser to return to the previously object visited.

The **Look Into** command lets you migrate into the selected object.

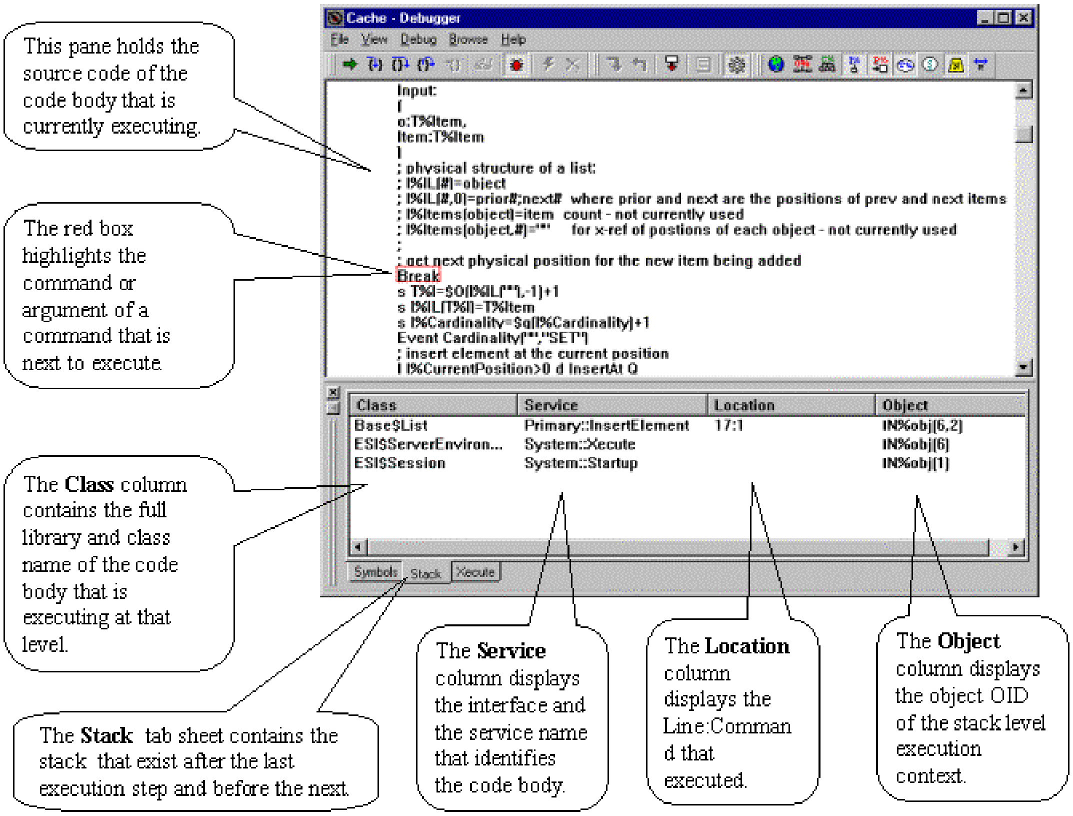
The **Show Descendants** command will display the next level of descendants of an array.

The **Look In Subscript** command lets you migrate into an object that is pointed to by an array subscript.

The **Show** commands listed here are used to toggle the display of that specific item off or on

Debugger Window with Stack Display

The picture below illustrates the Debugger Window with the State windows Stack tab sheet displayed. The Stack tab sheet contains the stack state before the command or command argument outlined in the red box is to execute.



```
Input:
(
o:T%Item,
Item:T%Item
)
; physical structure of a list:
; I%IL(#)=object
; I%IL(#,0)=prior#;next#  where prior and next are the positions of prev and next items
; I%Items(object)=item  count - not currently used
; I%Items(object,#)=""     for x-ref of postions of each object - not currently used
;
; get next physical position for the new item being added
Break
s T%I=$O(I%IL(""),-1)+1
s I%IL(T%I)=T%Item
s I%Cardinality=$g(I%Cardinality)+1
Event Cardinality("","SET")
; insert element at the current position
I I%CurrentPosition>0 d InsertAt Q
```

Class	Service	Location	Object
Base$List	Primary::InsertElement	17:1	IN%obj(6,2)
ESI$ServerEnviron...	System::Xecute		IN%obj(6)
ESI$Session	System::Startup		IN%obj(1)

This pane holds the source code of the code body that is currently executing.

The red box highlights the command or argument of a command that is next to execute.

The **Class** column contains the full library and class name of the code body that is executing at that level.

The **Stack** tab sheet contains the stack that exist after the last execution step and before the next.

The **Service** column displays the interface and the service name that identifies the code body.

The **Location** column displays the Line:Command that executed.

The **Object** column displays the object OID of the stack level execution context.

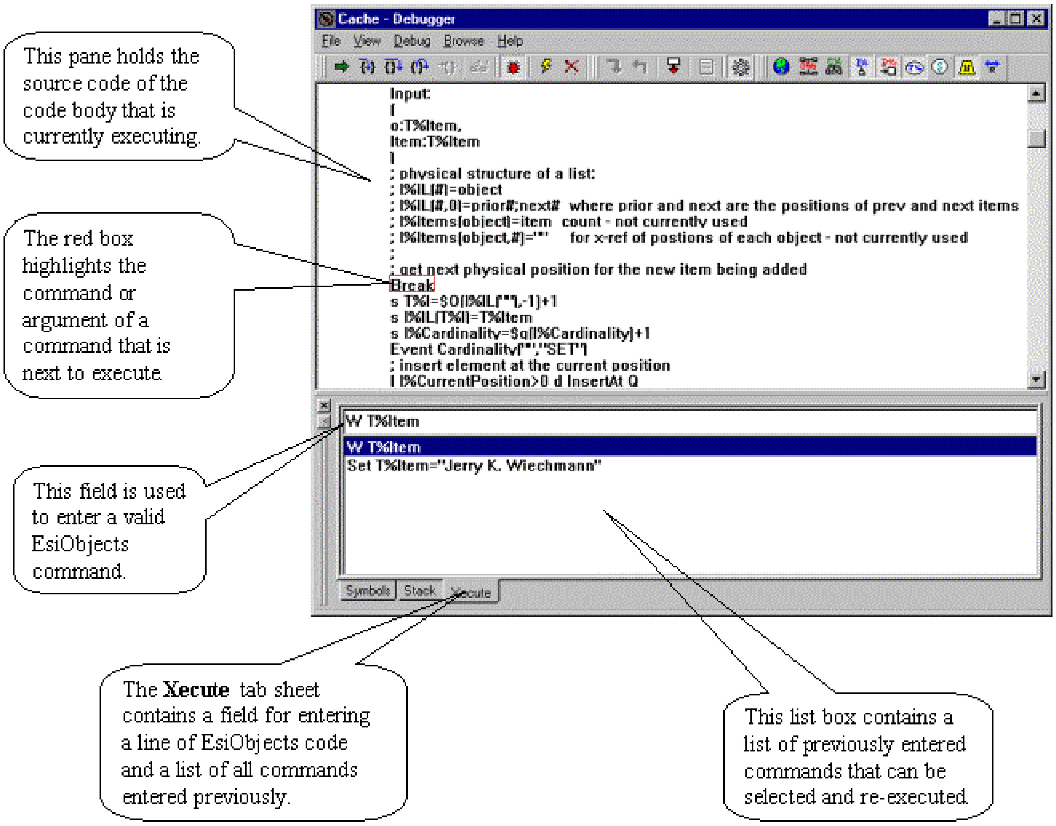
Debugger Window with Xecute Display

The picture below illustrates the Debugger Window with the State window Xecute tab sheet displayed.

The Xecute shell is different from the normal stand-along Xecute Shell in that it operates off the execution stack. This means that all context symbols are available to you.

The Xecute tab sheet contains a field that you can use to enter full EsiObjects commands and a list box that retains a history of all the commands entered in the event that you may want to re-execute them at a later time.

In addition to executing commands within the Xecute field, you can evaluate expressions by simply preceding the expression with an equals sign (=). If there was not an error, the expression will display with its value separated by an equal character. For example: **=10+20** would appear as **10+20=30**.



```
Input:
[
o:T%Item,
Item:T%Item
]
; physical structure of a list:
; I%IL(#)=object
; I%IL(#,0)=prior#;next#  where prior and next are the positions of prev and next items
; I%Items(object)=item  count - not currently used
; I%Items(object,#)=""     for x-ref of postions of each object - not currently used
;
; get next physical position for the new item being added
Break
s T%I=$O(I%IL(""),-1)+1
s I%IL(T%I)=T%Item
s I%Cardinality=$g(I%Cardinality)+1
Event Cardinality("","SET")
; insert element at the current position
I I%CurrentPosition>0 d InsertAt Q
```

```
W T%Item
```

```
W T%Item
Set T%Item="Jerry K. Wiechmann"
```

This pane holds the source code of the code body that is currently executing.

The red box highlights the command or argument of a command that is next to execute.

This field is used to enter a valid EsiObjects command.

The **Xecute** tab sheet contains a field for entering a line of EsiObjects code and a list of all commands entered previously.

This list box contains a list of previously entered commands that can be selected and re-executed.

Debugger: Main Menu Items

The **Debugger Window** is separate from the EsiObjects Main window and consequently has a separate menu. However, all menus that are in common to the Main Window are identical at the command level. The **Debugger Window** has one menu item specific to the debugger – the **Debug** menu item.

Main File Menu

Menu	Command	Description
File	**Print...**	Invokes the print dialog and then prints the selected object or text to the selected printer if you choose to proceed.
	Print Setup...	Invokes the Printer Setup form and lets you change the printer setup parameters. If you choose to proceed, the printer and printing options will be changed.
	Exit	Shuts down the Debugger Window.

Main View Menu

Menu	Command	Description

View		
	Toolbars|Debugging Actions	Executing this command will toggle the Debugging Actions toolbar to hide and appear.
	Toolbars|Symbol Types	Executing this command will toggle the Symbol Types toolbar to hide and appear.
	Toolbars|Browse Actions	Executing this command will toggle the Browse Actions toolbar to hide and appear.
	State	This command toggles the State window display off and on. The State window contains the Symbols, Stack and Xecute tab sheets.

Main Debug Menu

Menu	Command	Description
Debug		
	Go	Executing this command will cause the program to execute to completion or the next break point.
	Run to Cursor	Causes the program to execute up to the current cursor position in the code window.
	Step Into	Causes execution to step into the next code block.
	Step Out	Causes the execution to continue until the execution stack is popped.
	Step Over	The next block of code will be executed. The debugger will stop upon returning to the current level of execution.
	Debugger Active	Toggles the debugger as active or inactive.

Main Browse Menu

Menu	Command	Description
Browse		
	Look Into	Within the context of the Object Browser, if a variable is selected that has a OID associated with it, executing this command will force the Object Browser to migrate that link and display the context of the object pointed to by the subscript OID.
	Look In Subscript	Within the context of the Object Browser, if a variable is selected that has a OID as a subscript, executing this command will force the Object Browser to migrate that link and display the context of the object pointed to by the subscript OID.
	Pull Out	Executing this command will force the Object Browser to return to the object it came from and redisplay its context.
	Watch	Not Implemented Yet.

Show Descendants Not Implemented Yet.

Refresh This command, when executed, will totally refresh the Object Browsers display of an object state (variables and values).

Show History The Object Browser keeps track of the objects it migrates through. Executing this command will force a List History list box to appear, displaying the migration history.

Edit Value When you have selected a variable within the Object Browser that has a string value bound to it, executing this command put you into edit mode. The value of the variable can then be modified.

Goto Definition Not Implemented Yet

Class Not Implemented Yet

Evaluation Not Implemented Yet

Recycle You have control over whether a completely new Object Browser is instantiated every time you migrate to a new object. The Recycle button on the browsers toolbar controls this. If the button is depressed, that means that only one instance of the browser will exist for all migrations. This command will indicate that by a √ in front of it. Executing this command toggles the Recycle button between the recycle and no recycle states.

Auto Refresh If you are changing the state of an object using the Object Browsers embedded Xecute Shell, you can use this toggle command to turn auto-refresh on and off. When on (indicated by a √ in front of the command), changing the state of the object being browsed will automatically cause the display to refresh. Toggling the Auto Refresh command causes the equivalent Auto Refresh button on the Object Browsers toolbar to pop in and out.

Main Help Menu

Menu	Command	Description
Help		
	Getting Started	Activates the Acrobat Reader and displays the Getting Started Tutorial. This tutorial is designed to teach you some fundamental object oriented concepts. It is primarily designed to teach you how to use the EsiObjects tool set.

Administrator’s Guide
Activates the Acrobat Reader and displays the Administrator’s Guide. This guide contains all the information needed to start and shutdown the EsiObjects system as well as how to install and set up the servers for the supported M systems.

Language Reference Guide
Activates the Acrobat Reader and displays the Language Reference Guide. This guide contains all the information you will need to use the EsiObjects language. Each language element is explained in detail.

Programmer’s Reference Guide
Activates the Acrobat Reader and displays the Programmer’s Reference Guide. This guide contains all the information you will need to know about objects and how to use them within your application.

Tools Guide
Activates the Acrobat Reader and displays the Tools Guide. This guide contains extensive information about the EsiObjects tool set. Each GUI object is described in detail along with instructions on how to use it.

About EsiObjects
Invokes a dialog that displays current status information about the EsiObjects Class Development Environment.

Using the Interactive Debugger

Because developing software systems is usually a complex process and the mere fact that human beings make mistakes, **Interactive Debuggers** have become an integral part of software development environments.

The fundamental purpose of a debugger is to let the programmer control the execution of a software component so that the context of execution can be observed as each instruction is executed. A good debugger will let the programmer set break points, control the flow of execution in various ways and modify the state (variables) of objects. It will display the sequence of code execution, the variable states at each step and the execution stack contents.

The EsiObjects Interactive Debugger should be used when an error terminates the execution of an application and the cause is not obvious. Using a debugger is a habit the programmer must learn. Rather than wasting time looking at the code, activating the debugger and stepping through the execution sequence usually exposes the problem immediately, saving a lot of development time.

To use the debugger, follow the steps outlined below.

1) From the main menu, select the **View| Debugger** command. This will invoke the debugger, displaying the window with three empty panes.

2) The debugger should be started in active mode. This is indicated by **Enable Debugging** button being depressed. If it is not, depress the button to activate it. You may pop it out to de-activate the debugger at any time. In active mode, the

debugger will automatically react when a **BREAK** command is encountered while executing a code body that has a compiled debug version.

3) To actually force the debugging session to start, you must physically insert a **BREAK** command in the code to be debugged. Make sure you insert it prior to the suspected problem area. You must compile a debug version of the code.

4) Now run the application such that the code will encounter the **BREAK** command. At this point, the debugger will automatically be activated. The panes will fill in and control will be handed over to you. Note: *The EsiObjects system is modal at this point. The debugger has control of the system. To use any other tool in the EsiObjects system, the debugger must be deactivated.*

At this point you may use the Step buttons to execute the code. The Step and other buttons are described as follows:

This is the **Go** (F5 accelerator key) button. Clicking on this button will terminate step mode and continue normal execution of the code until it execute to the end or it encounters another BREAK command. All buttons will gray out indicating the end of this debugging session.

This is the **Step Into** (F9 accelerator key) button. Clicking on this button will tell the debugger to step into the next message, subroutine or extrinsic function *if it has a compiled version*. If it does not have a compiled version, it will execute as normal.

This is the **Step Over** (F8 accelerator key) button. Clicking on this button will tell the debugger to step over the next message, subroutine or extrinsic function.

This is the **Step Out** (Shift+F9 accelerator key) button. Clicking on this button will tell the debugger to step out of the current message, subroutine or extrinsic function.

This is the **Run to Cursor** button and is currently **not implemented**.

This is the **Watch Variable** button and is currently **not implemented**.

Here are some important things to remember when using the Interactive Debugger:

- The Interactive Debugger must be started (window present) and activated (**Enable Debugger** button depressed).
- The code body must have a **BREAK** command inserted at the appropriate point and the code body must have a Debug compile available. (Bug icon next to the version number in the appropriate editor window).
- At any point within the step-by-step process, you may depress the **Go** button or pop out the **Enable Debugging** button to continue normal execution.

Xecute Shell

The Xecute Shell Explained

The Xecute Shell is the EsiObjects equivalent of the M programmer's prompt. It allows you to enter commands and send messages.

Most programming operations can be accomplished with the EsiObjects browsers and editors. However, sometimes it is necessary to enter commands at a prompt. The Xecute Shell lets you do this. The important thing to remember about the Xecute Shell is that it operates within the context of an object. The Xecute Shell is invoked from the **Object|Xecute Shell** command of the Main Menu. Remember that when an object is active within the client area or in the Session Browser window, commands active within the context of that object are highlighted in the Main Menu. They operate on the object. If you invoke the Xecute Shell command, a window will be brought up that lets you enter commands for *direct* execution. These commands operate within the context of the active object.

Some things you can do from the Xecute Shell are:

- Invoke the Object Browser to explore the internals of the current or another object.
- Instantiate objects directly.
- Send messages to objects.
- Enter standard M code. For example, it is acceptable to call a conventional M routine that runs in the main console window. Run the routine, switch to the console window, and go.

A line of code entered from the Xecute Shell works just like a line of code in one of the object's methods.

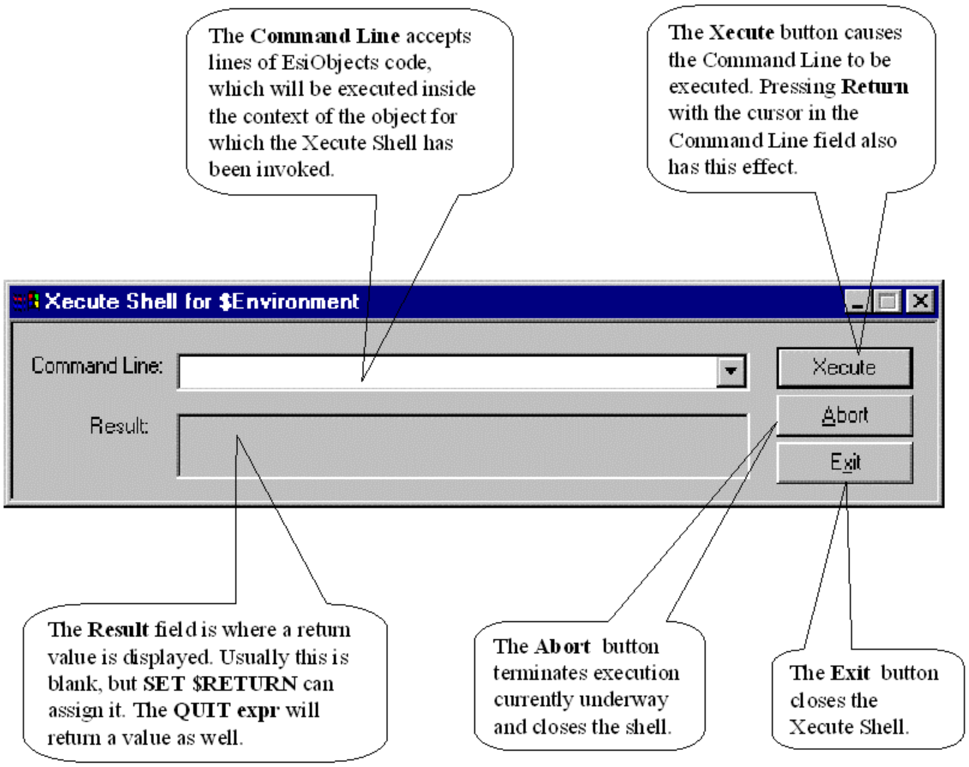
Xecute Shells are also found in other contexts, including in the Object Browser. The Object Browser can be invoked from the Xecute Shell via the **ZVIEW** command. The Object Browser has a Xecute Shell built into it. In addition to executing lines of code, the Object Browser shell lets the programmer evaluate expressions.

The advantage of using the Object Browser is that it shows you the instance variables of the browsed object. In addition to the Xecute Shell functionality, it gives you the opportunity to examine, browse and edit variables interactively.

Xecute Shell Window

The **Xecute Shell** window allows commands to be entered in the context of any object. It is a simpler tool than the Object Browser, and its purpose is more tightly constrained. The Xecute Shell for the **Environment** is invoked from the EsiObjects CDE Main Window, by selecting the **Xecute Shell** item of the **Object** menu.

The picture below illustrates and describes the components of the Xecute Shell.



The **Command Line** accepts lines of EsiObjects code, which will be executed inside the context of the object for which the Xecute Shell has been invoked.

The **Xecute** button causes the Command Line to be executed. Pressing **Return** with the cursor in the Command Line field also has this effect.

The **Result** field is where a return value is displayed. Usually this is blank, but **SET $RETURN** can assign it. The **QUIT expr** will return a value as well.

The **Abort** button terminates execution currently underway and closes the shell.

The **Exit** button closes the Xecute Shell.

Using the Xecute Shell

The Xecute Shell Window contains a Command Line field and a Result field. The Command Line field accepts EsiObjects commands and is similar to the direct mode prompt (>) on an M system.

To open the Xecute Shell:

1) Execute the Main Menu command **Object|Xecute Shell**.

Note: Notice that the Xecute Shell is operating within the context of your environment object assigned to when you started your session. This environment is always available to you through the $ENV special variable. Because you have programming access to the environment, you must exercise caution. The environment uses variables. You do not want to alter or delete these variables. Since you will often use the environment to create test objects and set test variables, you should confine variable name creation to a namespace that does not conflict with the environments. For example, I%Test is not used.

2) To use the Xecute Shell, enter a command line in the Command Line drop-down combo box. The Command Line field is a drop down box, which allows you to retrieve commands that were entered previously. You can select a command from the

list and edit it or execute the command line as it is again. Note that these commands are lost if you close the Xecute Shell window.

3) Click **Xecute** button to execute the command.

If the command is an EsiObjects language construct that returns a value, it will appear in the Result field. Most lines of code do not have return values unless they set the **$RETURN** special variable. If you want to execute an expression and return it to the Return field, you can use the Set $Return=Expression or Quit Expression construct. In either case the return value will be sent to the Return field of the Xecute Shell window.

Executing a command does not close the Xecute Window. You can keep the Xecute Window open during your EsiObjects programming session. Like an M prompt, it is always available for entering commands and sending messages. If you find that the Xecute Window gets in your way when working with other browsers, you can minimize the window.

6. Click **Exit** to close the Xecute Window.

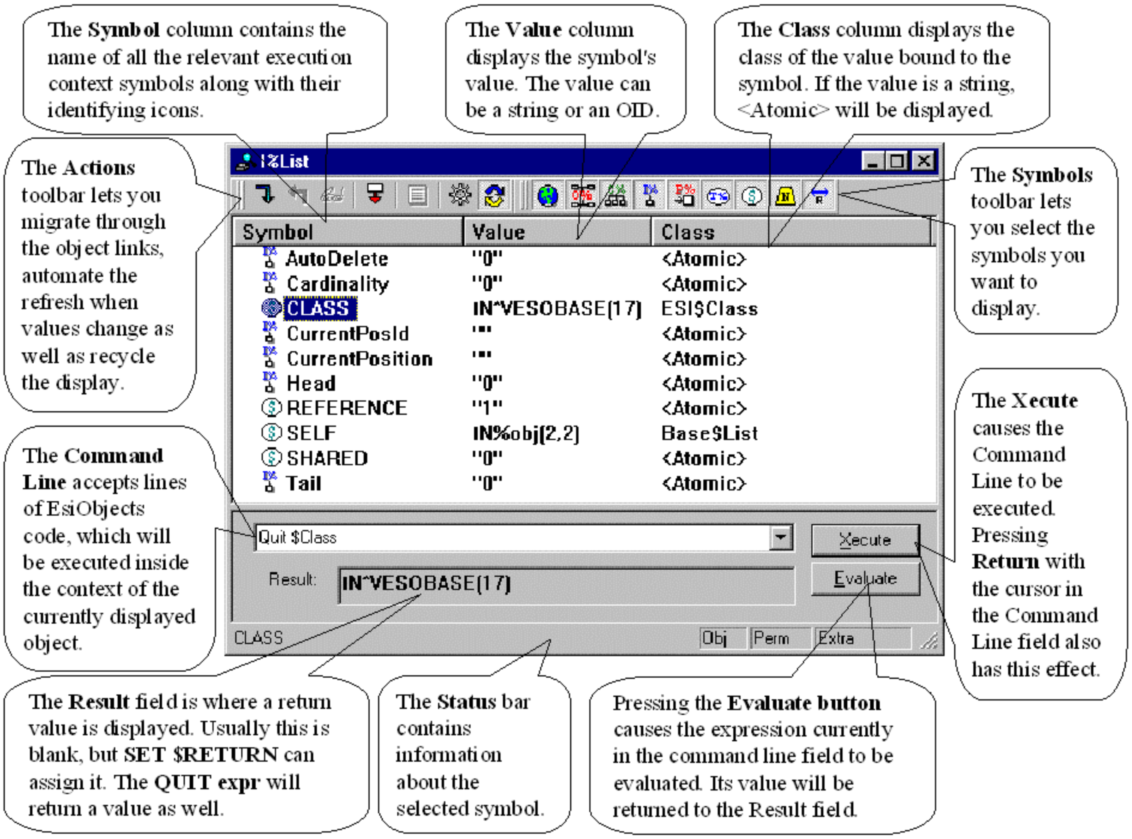
Object Browser

The Object Browser Explained

The **Object Browser** allows the internals of any object in the system to be examined. The browser exposes the internal state of the object - it is a programming tool. It is a static debugging tool that lets you examine the internals of an object as well as actually change the values of variables. Normally, it is invoked by first issuing the **ZVIEW** command via the Xecute Shell. See the EsiObjects Language Reference Guide See the section Xecute Shell of this guide for more information on this subject.

Object Browser Window

The following picture illustrates and describes the major components of the Object Browser.



Symbol	Value	Class
AutoDelete	"0"	<Atomic>
Cardinality	"0"	<Atomic>
CLASS	IN^VESOBASE(17)	ESI$Class
CurrentPosId	""	<Atomic>
CurrentPosition	""	<Atomic>
Head	"0"	<Atomic>
REFERENCE	"1"	<Atomic>
SELF	IN%obj(2,2)	Base$List
SHARED	"0"	<Atomic>
Tail	"0"	<Atomic>

Quit $Class

Result: IN^VESOBASE(17)

CLASS

The **Symbol** column contains the name of all the relevant execution context symbols along with their identifying icons.

The **Value** column displays the symbol's value. The value can be a string or an OID.

The **Class** column displays the class of the value bound to the symbol. If the value is a string, <Atomic> will be displayed.

The **Actions** toolbar lets you migrate through the object links, automate the refresh when values change as well as recycle the display.

The **Symbols** toolbar lets you select the symbols you want to display.

The **Command Line** accepts lines of EsiObjects code, which will be executed inside the context of the currently displayed object.

The **Xecute** causes the Command Line to be executed. Pressing **Return** with the cursor in the Command Line field also has this effect.

The **Result** field is where a return value is displayed. Usually this is blank, but **SET $RETURN** can assign it. The **QUIT expr** will return a value as well.

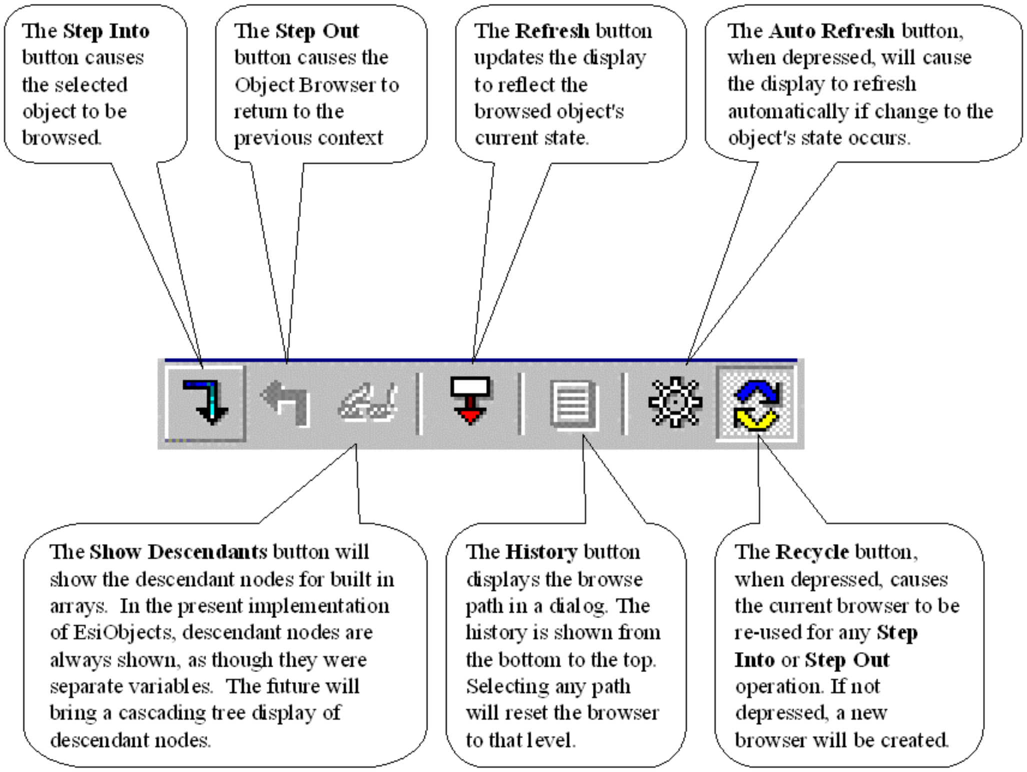
The **Status** bar contains information about the selected symbol.

Pressing the **Evaluate button** causes the expression currently in the command line field to be evaluated. Its value will be returned to the Result field.

The Object Browser contains two toolbars, a status bar. Each will be explained in detail in the following sections. For information on the embedded Xecute Shell, see the section in this Guide called Xecute Shell.

Object Browser Actions Toolbar

The **Object Browser Actions Toolbar** buttons let you migrate through objects, control the refreshing of the browser as well as the instantiation of new browsers. The picture below illustrates and describes all the buttons.



The **Step Into** button causes the selected object to be browsed.

The **Step Out** button causes the Object Browser to return to the previous context

The **Refresh** button updates the display to reflect the browsed object's current state.

The **Auto Refresh** button, when depressed, will cause the display to refresh automatically if change to the object's state occurs.

The **Show Descendants** button will show the descendant nodes for built in arrays. In the present implementation of EsiObjects, descendant nodes are always shown, as though they were separate variables. The future will bring a cascading tree display of descendant nodes.

The **History** button displays the browse path in a dialog. The history is shown from the bottom to the top. Selecting any path will reset the browser to that level.

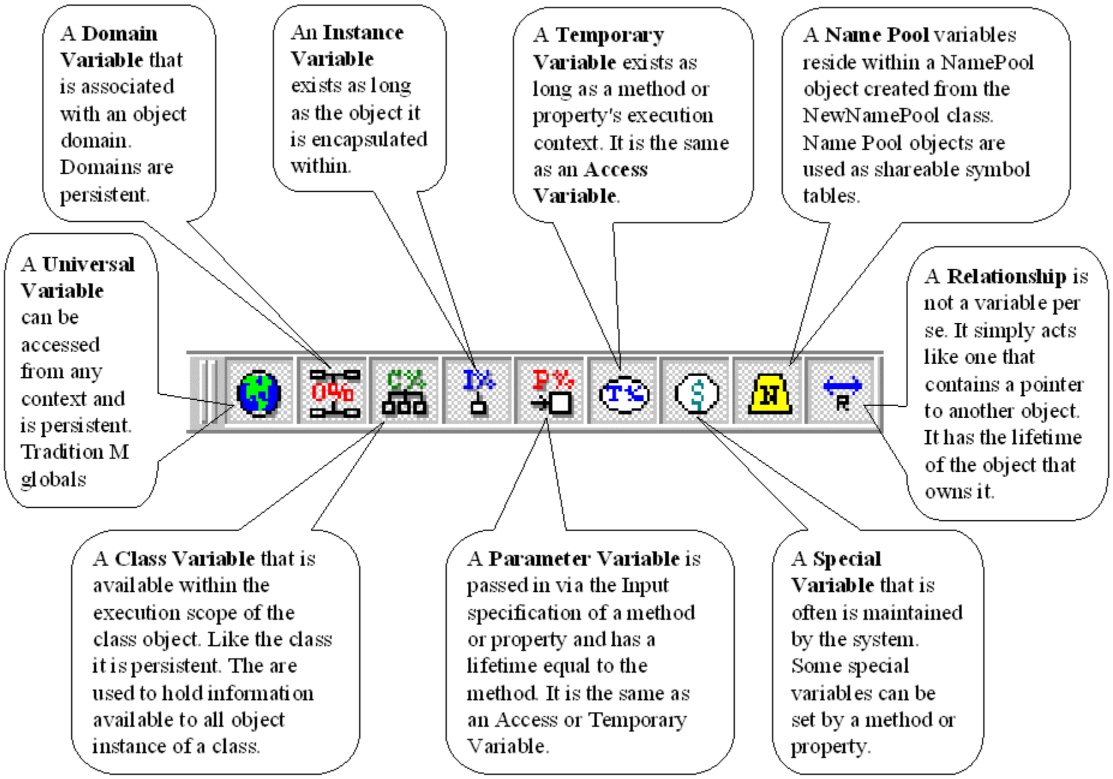
The **Recycle** button, when depressed, causes the current browser to be re-used for any **Step Into** or **Step Out** operation. If not depressed, a new browser will be created.

Object Browser Symbol Toolbar

The **Object Browser Symbol Toolbar** contains a button for each symbol scope supported by EsiObjects. Often, when developing an application, you are only interested in exploring the internals of an object. While doing so, these buttons let you toggle the display of scoped variables on and off. If the button is depressed, the system will display that variable.

Refer to the Variables section of the EsiObjects Language Reference Guide for more information on symbols and scoping.

The Symbol toolbar is illustrated below. Each button is described.



A **Domain Variable** that is associated with an object domain. Domains are persistent.

An **Instance Variable** exists as long as the object it is encapsulated within.

A **Temporary Variable** exists as long as a method or property's execution context. It is the same as an **Access Variable**.

A **Name Pool** variables reside within a NamePool object created from the NewNamePool class. Name Pool objects are used as shareable symbol tables.

A **Universal Variable** can be accessed from any context and is persistent. Tradition M globals

A **Relationship** is not a variable per se. It simply acts like one that contains a pointer to another object. It has the lifetime of the object that owns it.

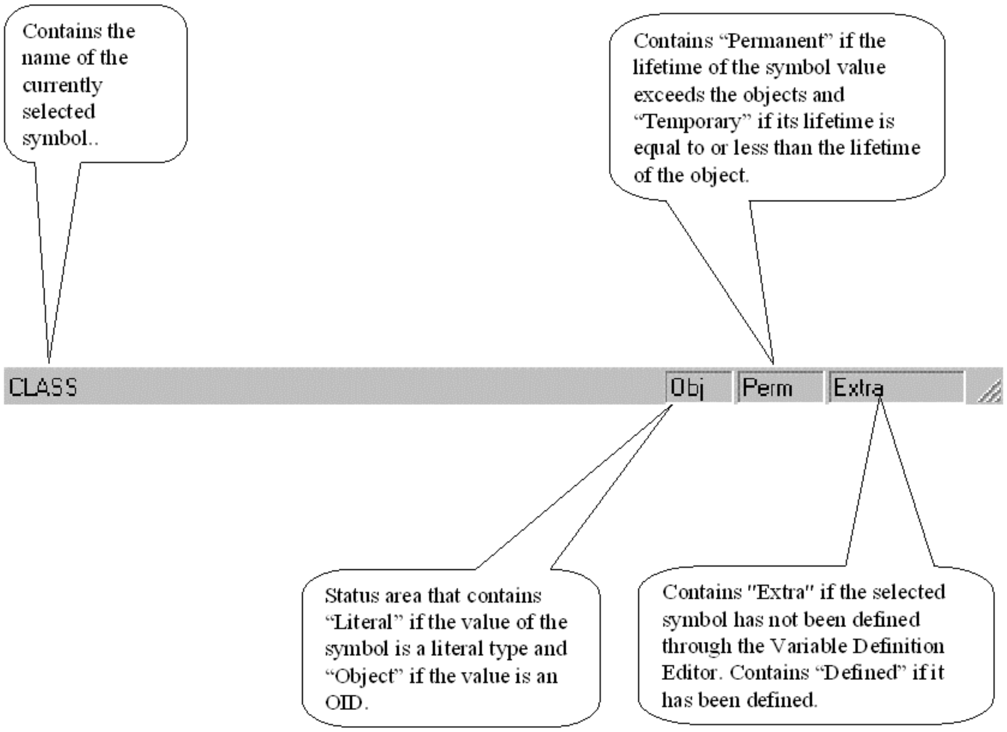
A **Class Variable** that is available within the execution scope of the class object. Like the class it is persistent. The are used to hold information available to all object instance of a class.

A **Parameter Variable** is passed in via the Input specification of a method or property and has a lifetime equal to the method. It is the same as an Access or Temporary Variable.

A **Special Variable** that is often is maintained by the system. Some special variables can be set by a method or property.

Object Browser Status Bar

The **Object Browser Status Bar** displays the status of a selected symbol. The picture below illustrates and describes each component of the status bar.



Contains the name of the currently selected symbol..

Contains "Permanent" if the lifetime of the symbol value exceeds the objects and "Temporary" if its lifetime is equal to or less than the lifetime of the object.

Status area that contains "Literal" if the value of the symbol is a literal type and "Object" if the value is an OID.

Contains "Extra" if the selected symbol has not been defined through the Variable Definition Editor. Contains "Defined" if it has been defined.

Browsing Dead Objects

If an instance variable contains a reference to an object that no longer exists, or if the currently browsed object has been destroyed, then the object browser's display will be adjusted as follows:

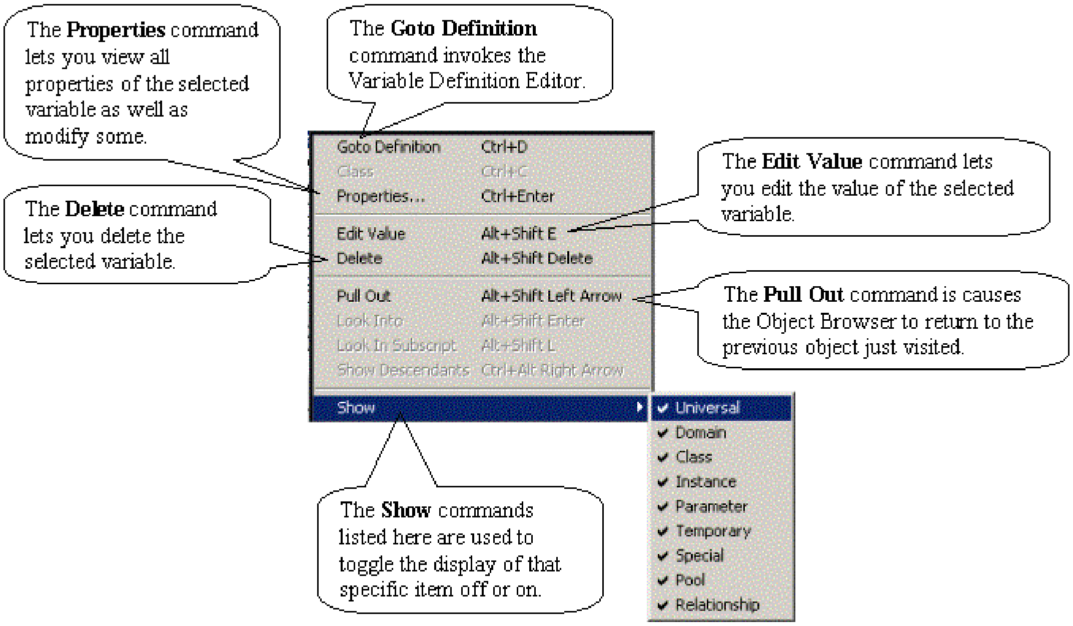
- Dead objects will show MIA in the status area when they are selected.
- The value column for a dead object will read "Dead Object…" or "Dead Protected Object…", depending on its type.
- When you browse a dead object, the text and background colors will be inverted in the object browser window.
- Dead objects will show the **$REFERENCE** value as zero (0).

Object Browser Popup Menu

The Object Browser popup menu is invoked by right clicking on a variable in the Object Browser's main panel or pressing **Shift+F10** key combination after selecting the variable. Any variable can have two types of values: **string** or **object**. The menu commands highlighted on the object popup menu are dependent upon what value type is bound to the variable.

String Value Popup Menu

The illustration below describes the string commands of the Object Browser.



The **Properties** command lets you view all properties of the selected variable as well as modify some.

The **Goto Definition** command invokes the Variable Definition Editor.

The **Edit Value** command lets you edit the value of the selected variable.

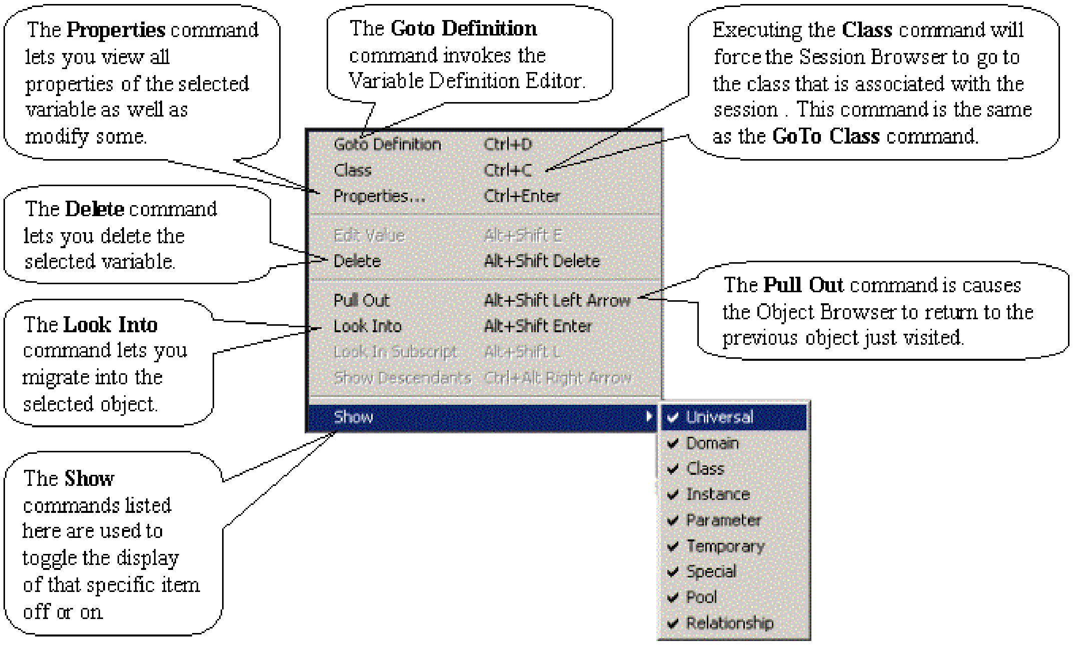
The **Delete** command lets you delete the selected variable.

The **Pull Out** command is causes the Object Browser to return to the previous object just visited.

The **Show** commands listed here are used to toggle the display of that specific item off or on.

Class Value Popup Menu

The illustration below describes the object commands of the Object Browser.



The **Properties** command lets you view all properties of the selected variable as well as modify some.

The **Goto Definition** command invokes the Variable Definition Editor.

Executing the **Class** command will force the Session Browser to go to the class that is associated with the session . This command is the same as the **GoTo Class** command.

The **Delete** command lets you delete the selected variable.

The **Pull Out** command is causes the Object Browser to return to the previous object just visited.

The **Look Into** command lets you migrate into the selected object.

The **Show** commands listed here are used to toggle the display of that specific item off or on

Using the Object Browser

The Object Browser is used to view and modify the internals of an object. It is a static debugging tool. The EsiObjects Interactive Debugger is designed to let you control the execution of a code body and view the state of the object in terms of its symbol table and its execution stack. It is a dynamic tool. The Object Browser lets you view and actually modify the state of a object in a static state, outside of the execution context of a code body.

Invoking the Object Browser

First you must determine what object you want to browse. You must have access to its OID or you can select one of the objects in the Session Browser window or Client Area. **Note: If you want the Xecute Shell to come up in the context of the current session's environment object, click in the Documentation or Output Windows.**

To invoke an Object Browser, follow the instructions below:

1) Execute the Main Menu command **Object|Xecute Shell**. This will bring the Xecute Shell up in the client area of the Main Window.
2) The **ZVIEW** command is used to invoke the Object Browser. It can take an argument that must evaluate out to an OID or it can be argumentless. If it is argumentless, the object context that the Xecute Shell was brought up in will be browsed. Enter the **ZVIEW** command in the Command Line field of the Xecute Shell to invoke it on the object you want to browse.

At this point the Object Browser will display in the client area. Refer to the section The Object Browser Explained for a complete description of the Object Browsers components.

Note: When the Object Browser first displays it will display variables that are available on the stack at that time. However, upon returning to this stack level subsequently, you may not see these variables. This is normal behavior.

Setting Up for Browsing

The Object Browser can be set up to:

- Recycle the current browser every time a new object is migrated to
- Refresh the contents of the display automatically or manually whenever the state of the viewed object has been altered either interactively or via the embedded Xecute and evaluation shell.
- View only selected scoped variables.

Recycling the Object Browser Window

On the **Actions Tool Bar**, the **Recycle** button , when in the depressed state, lets you reuse the current browser window. If it is not depressed, a new browser window will be created when migrating into a new object.

Refreshing the Object Browser Display

Also on the **Actions Tool Bar**, when the **AutoRefresh** button is depressed, the display area of the browser will automatically refresh when the objects state changes. If you do not want it to refresh, simply pop the button out. This lets you refresh the state manually using the **Refresh** button . Simply click on it whenever you want the display refreshed to its current state.

Viewing Selected Symbols

Object can have a large umber of symbols stored in them. Often you are only interested in certain symbols such as Instance variables and no others. It is possible to select those symbols that you want displayed and eliminate those you do not want displayed by using the **Symbols Tool Bar**. This tool bar (and the Show cascading menu on the popup menu) let you select which symbols and values you want displayed. Selecting the symbols you want to see speeds up the display if there are a large number of symbols in an object.

Migrating Object Hierarchies

The Object Browser is capable of migrating links between objects and displaying the target object's internal state. The **Actions Tool Bar** contains all the buttons needed to migrate through an object hierarchy.

On the **Actions Tool Bar**, the **Step Into** button ↴ is used to step into the object that is currently selected in the display area. If the selected item is an <Atomic> type (literal), the Step Into button will not be highlighted. If it is an internal object, the Object Browser will step into that object and it's state will be displayed. Alternatively, to step into an object, simply double click on the selected item in the display window.

Another approach to migrating into an object is to use the popup menu that can be invoked by right clicking on the selected item. This provides all the functionality that the tool bar contains plus more. Since an EsiObjects object can contain primitive M data structures (arrays), often an array will have an OID in a subscript that you may want to look into. There are commands on the popup menu that permit you to migrate to objects pointed to by array subscripts.

Next to the **Step Into** button is the **Step Out** button ↰. Once you have stepped into an object, the Step Out button will highlight indicating that you may step out of the current object, returning to the context of the previous object. Once clicked on, the Object Browser will return to the previous object, redisplaying its state in the display area of the browser.

Viewing and Modifying an Objects State

Once you have migrated to the object you are interested in, there are several approaches to viewing and modifying an objects state. the following sections explain each approach.

Viewing the Definition of a Symbol

When browsing an object, you are often interested in the definition of a symbol. You can access the definition of an object (if it exists) by executing the **Goto Definition** command on the popup menu. Executing this command will bring up the **Variable Definition Editor**. See the section titled **Variable Definition Editor** in this guide for more information.

Using the Xecute and Evaluation Shell

At the bottom of the Object Browser is an embedded Xecute and Evaluation Shell that lets you execute a full line of EsiObjects code or evaluate an EsiObjects expression.

The Shell is operating in the context of the object being browsed. Consequently, you may change any symbol that resides in that object. Additionally, you can evaluate an expression by entering it in the Command field and then clicking on the Evaluate button.

Directly Modifying Symbol Values

Directly modifying values consists of double clicking on the symbol name that has an <Atomic> value associated with it. The browser then goes into edit mode and you can change the value associated with the symbol. To get out of edit mode you must hit the Enter key. Clicking outside of the symbol will not perform this operation.

Note: When editing a value, remember to place quotes around the string if they are required. The compiler will check any symbol you modify. If it incorrect, you will receive an error message.

Transport Tools

Transport Tools Explained

Within the EsiObjects system there are two levels of content that is defined when writing an object-oriented application. They are the:

1) Definitional Content

2) Instance or Legacy Content

Transporting Definitional Content

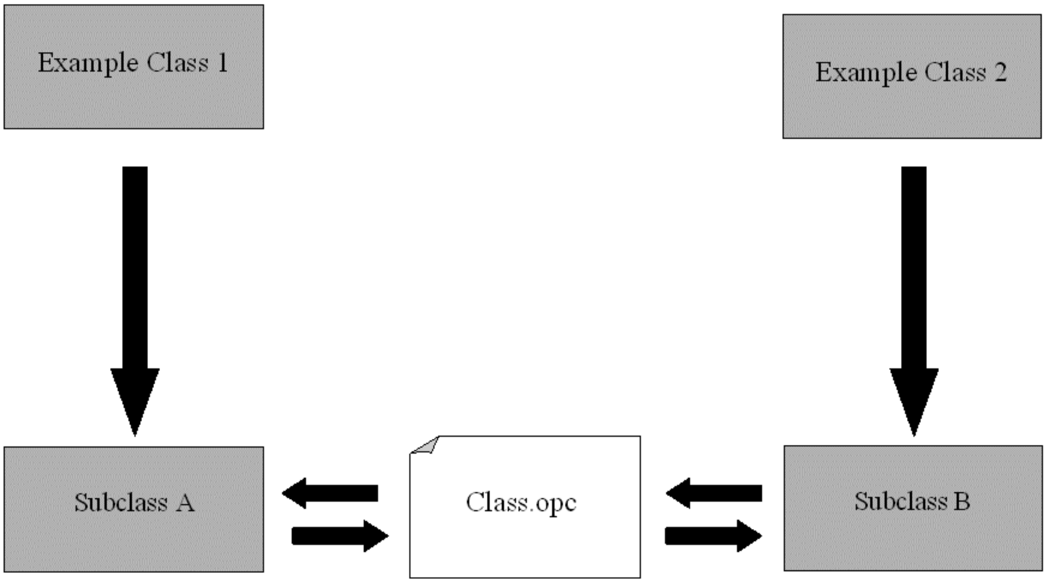
The **Definitional Content** of an EsiObjects application consists of libraries, classes, variables, services and documentation. These are objects that contain definitional information and documentation used to create instances. This information is exported and imported via ASCII text files. Upon import, these definitional objects are mapped to M globals and routines.

EsiObjects contains import and export functionality needed to transfer definitional objects to external file. Exporting means saving the definition of the object to an ASCII text file. Importing means restoring an object definition from an export file.

In EsiObjects, the following definitional objects can be exported: libraries, classes, nested classes, interfaces, method and property source code versions, event source code, relationships and variable definitions. It should be noted that exporting a class library would export the *entire* library including all classes, nested classes and all methods, properties, relationships, events and variable definitions within the classes.

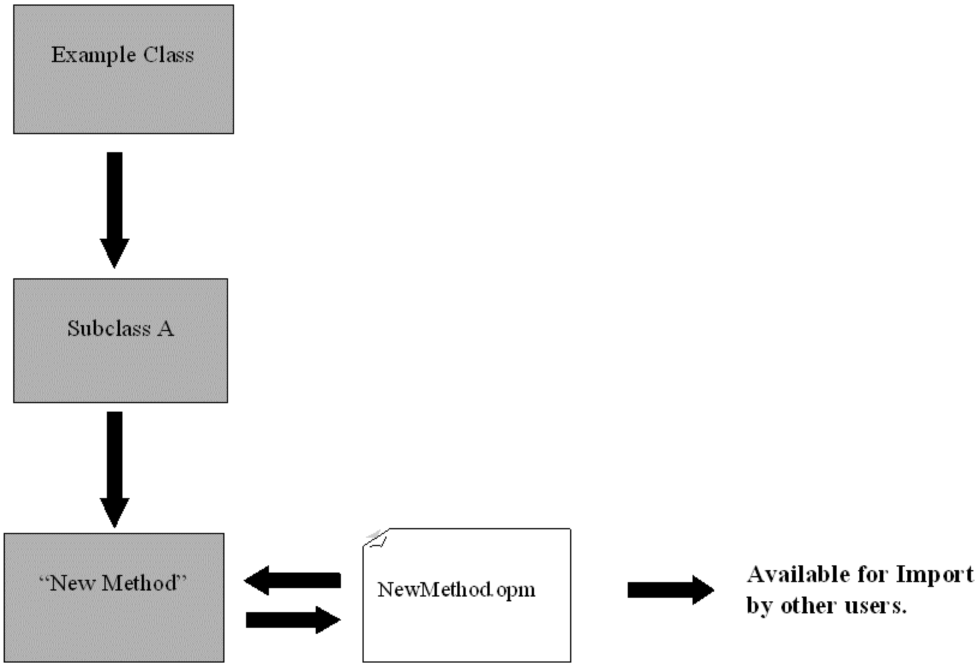
Generic Import

Generic Import is a special type of import. Typically, when you import you will have selected an object and have chosen the Import option to import directly to the object currently selected. For example, if you had an export for Subclass A, which is a subclass of Example Class 1 (see the diagram below), you would select that class in the Session Browser and choose the Import option. After selecting the proper file, Subclass A would be overwritten with the contents of the file. If you were to import the same export file over Subclass B, the system would allow you to do so. Selecting an object and importing will directly import to that object. Note that you could overwrite a class with an export of a completely different class. Of course, you could not import an object definition of a different type. For example, you could not import a method export file to a class.



In the above diagram note that Subclass A has been exported to the text file Class.opc. This file can be imported back into Subclass A. Also, Class.opc can be imported into Subclass B overwriting the definition of that class with the definition stored in Class.opc.

Keep in mind that in many cases you will want to import a definitional object to the same location from which it was exported. For example, let’s say a fellow developer is working on a class and has coded a new method. She exports the method and tells the programming staff to import the file “NewMethod.opm”. Instead of using the Session Browser to find the class, and then find the method (Note: If the method doesn’t exist yet, you would have to create it), before executing the import, members can simply select the **Tools|Generic Import** command from the main menu and select the “NewMethod.opm” file. The system will import the objects to the same location from which it was exported. Even if the class or method does not exist. The system creates any necessary objects along the path of the object before restoring it.



If Generic Import is used, as the user imports the file NewMethod.opm, all necessary objects will be created. A library is the exception to this rule since it knows what M globals to store the objects in and what name prefixes to use when creating an M routine at compile time. However, when exported, libraries can be instructed to retain this information. The Generic Import of a whole library would then know where to store the M globals and routines.

When to use Import vs. Generic Import

Use Import whenever you want to import a definitional object to a specific location or to overwrite one object with a definition of another. By selecting the object in the Session Browser, you can select the Import option (see the Import Option section) and you will import the saved definition over the selected object.

1) Use Generic Import to import the object definition to the same location from which it was exported. This is especially useful if you don't know exactly where the object was exported from but you need to import it to the same location it came from.

You do not need to have a Session Browser open to use Generic Import. It is available through the main menu **Tools|Generic Import** command.

Import/Export File Extensions

Many of the elements of the EsiObjects library structure can be exported to and imported from the host operating system via ASCII files. These files are in a special text format and have the extension OPx where x specifies the type (property, method, etc.).

The following is a list of generated OPx file extensions.

Extension	**Type of File**
.OPS	**S**ave file: Generic - Mix of all types.
.OPB	Code **B**ody
.OPC	**C**lass
.OPD	**D**ocumentation
.OPE	**E**vents
.OPF	**F**olders
.OPI	**I**nterface
.OPL	**L**ibrary
.OPM	**M**ethod
.OPP	**P**roperty
.OPV	Individual **V**ariables
.OPX	Full Variable Table

The extensions are generated automatically based on what object (s) are being exported. For example, if the Export command for a method is selected, the .OPM extension will be generated. If the Export command for a library is selected, the .OPL extension will be generated.

Transporting Instance and Legacy Content

The **Instance Content** of an EsiObjects application consists of the instances of the definitional objects (classes). These instances are mapped to M globals via the object creation process. Traditional M Global Tools exist within EsiObjects for the purpose of transporting these object instances.

The **Legacy Content** consists of those M routines and globals that make up a legacy M application. It may be wrapped using EsiObjects wrapping technology. The Global and Routine Transport tools are used to move legacy globals and routines into and out of the EsiObjects system.

These tools are grouped into two groups:

1) Global Transport Tools
2) Routine Transport and Editing Tools

Global Transport Tools

Two Global Transport Tools exist of exporting and importing M globals. They are the Global Save and Global Restore respectively.

Routine Transport and Editing Tools

Two Routine Transport Tools exist for exporting and importing M routines. They are the Routine Save and Routine Restore respectively. In addition to the transport tools, EsiObjects provides a Routine Editor. It is used to modify routines if required.

Using Definitional Object Transport Tools

EsiObjects provides extensive support for import and export operations. Definitional objects may be imported or exported:

1) *directly* via the Session Browser (library or folder structures) popup menus or main menu **Tools|Generic Import** command

or, *indirectly* via a Folder. See the Folder Operations section for more information on folders.

Direct Import and Export

Export Option

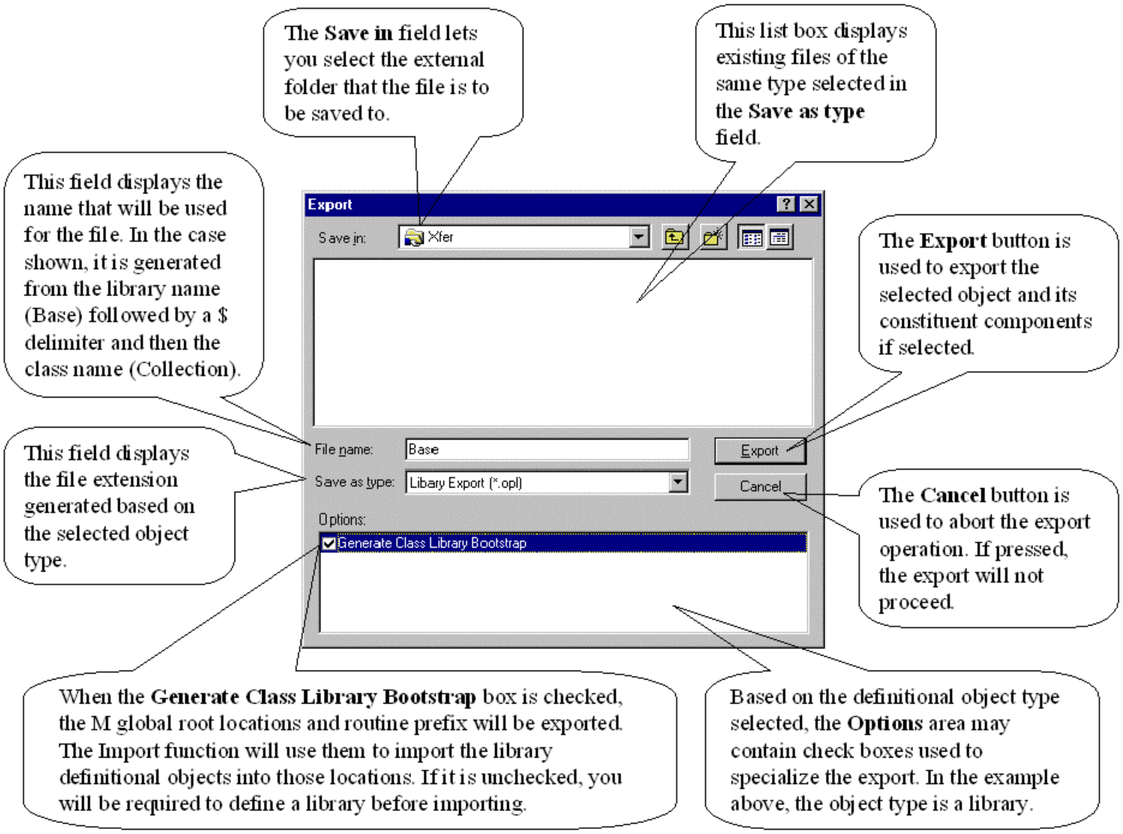
Any objects that are a part of the definitional structure (classes, libraries, methods, properties, property accessors, etc.) can be exported directly from the Session Browser or its descendant applications (property editor, method editor, etc.).

To export a definitional object, follow the instructions below.

1) Select the object you wish to export using the Session Browser. Please note that you may invoke an editor if exporting methods and properties. Within that editor, you may export specific versions.
2) Select the **Export** command from the object's popup menu. If you want to export a specific version of source code, bring up the popup menu by right clicking on the version number.
3) The Export dialog will appear as illustrated below. Fill in all the fields properly based on the type of object being exported.

Note: The check boxes that appear in the Options area are specific to the type of definitional object being exported. Only two types require specializing for export: Libraries and Classes. All other object type exports do not require specialization.

Library Export Dialog



Save in: Xfer

File name: Base

Save as type: Libary Export (*.opl)

☑ Generate Class Library Bootstrap

The **Save in** field lets you select the external folder that the file is to be saved to.

This list box displays existing files of the same type selected in the **Save as type** field.

This field displays the name that will be used for the file. In the case shown, it is generated from the library name (Base) followed by a $ delimiter and then the class name (Collection).

The **Export** button is used to export the selected object and its constituent components if selected.

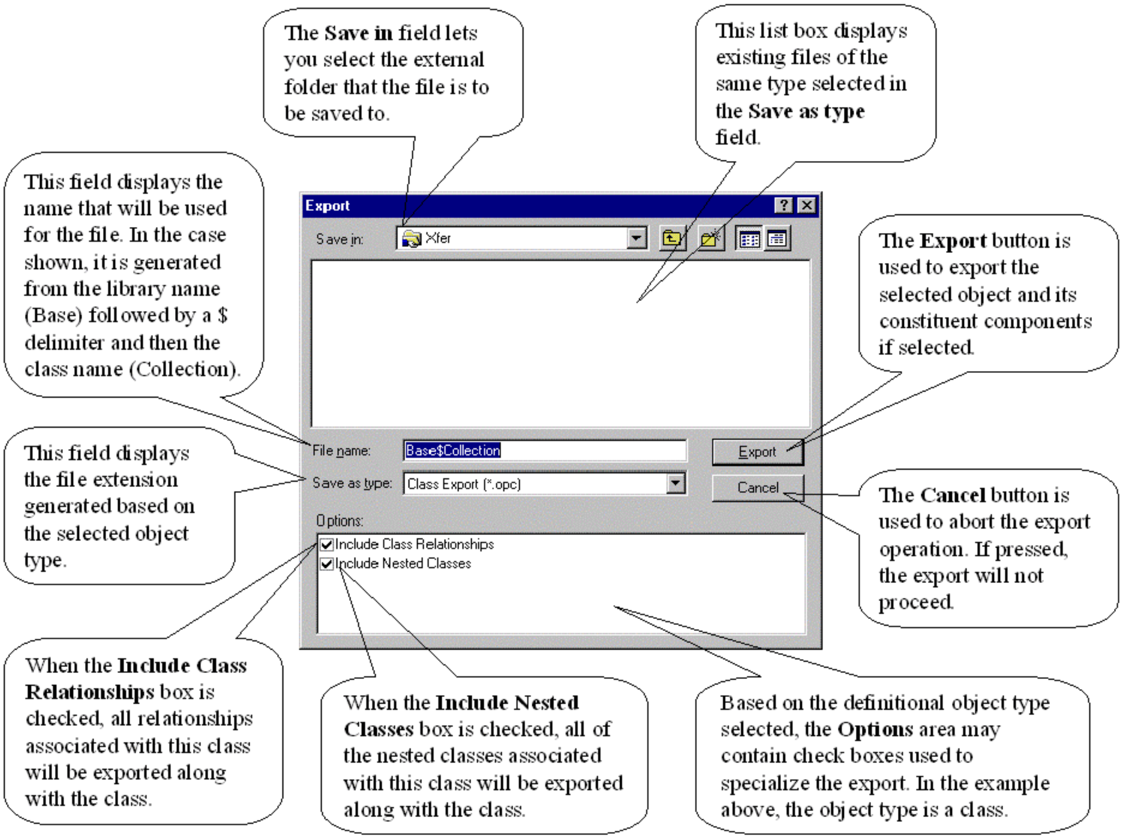
This field displays the file extension generated based on the selected object type.

The **Cancel** button is used to abort the export operation. If pressed, the export will not proceed.

When the **Generate Class Library Bootstrap** box is checked, the M global root locations and routine prefix will be exported. The Import function will use them to import the library definitional objects into those locations. If it is unchecked, you will be required to define a library before importing.

Based on the definitional object type selected, the **Options** area may contain check boxes used to specialize the export. In the example above, the object type is a library.

Class Export Dialog



Save in: Xfer

File name: Base$Collection

Save as type: Class Export (*.opc)

Options:

☑ Include Class Relationships

☑ Include Nested Classes

The **Save in** field lets you select the external folder that the file is to be saved to.

This list box displays existing files of the same type selected in the **Save as type** field.

This field displays the name that will be used for the file. In the case shown, it is generated from the library name (Base) followed by a $ delimiter and then the class name (Collection).

The **Export** button is used to export the selected object and its constituent components if selected.

This field displays the file extension generated based on the selected object type.

The **Cancel** button is used to abort the export operation. If pressed, the export will not proceed.

When the **Include Class Relationships** box is checked, all relationships associated with this class will be exported along with the class.

When the **Include Nested Classes** box is checked, all of the nested classes associated with this class will be exported along with the class.

Based on the definitional object type selected, the **Options** area may contain check boxes used to specialize the export. In the example above, the object type is a class.

7) Once all fields have been filled in properly, click on the **Export** button to export. At this point you may chose to abort the operation by clicking on the **Cancel** button.

A status dialog box will display the progress of the export. Once the status box disappears, the export will be complete. You may click on the **Cancel** button of the status dialog to abort the export at any time.

Import Option

To import a definitional object follow the instruction below.

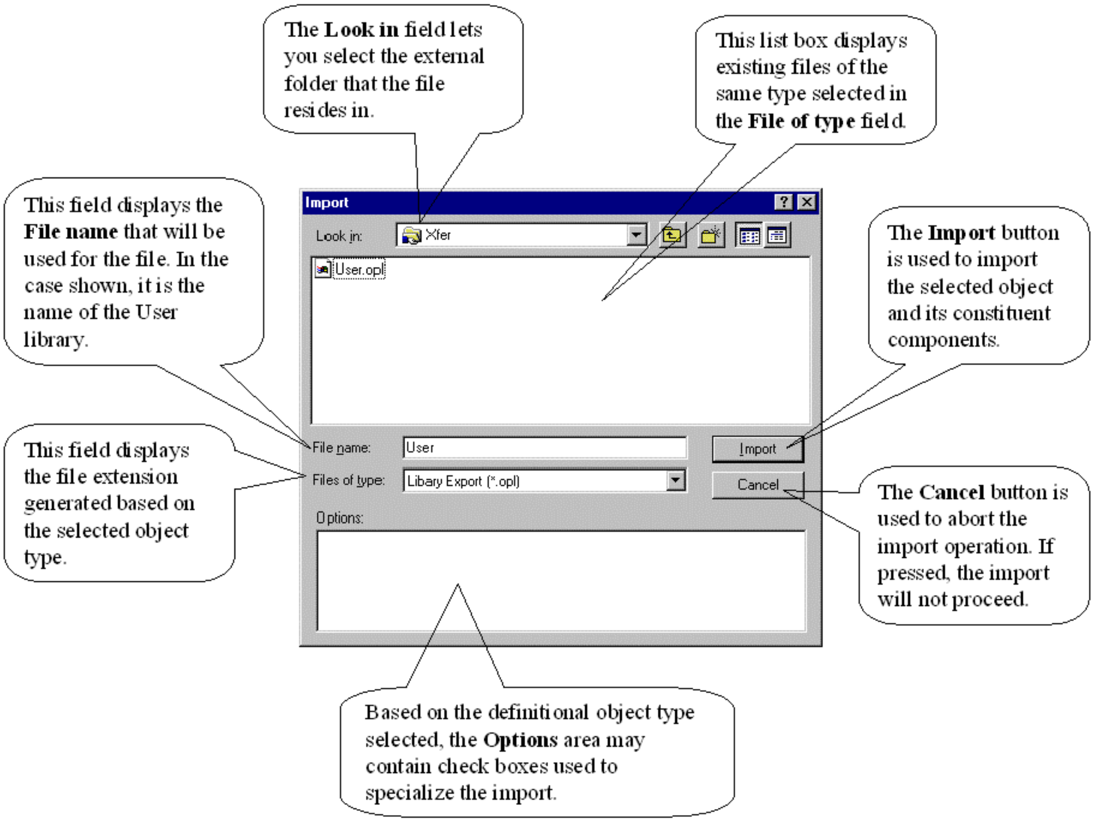
The imported object must already exist to use the Import function. If the object does not yet exist, then you must create it. Alternatively, you may use the **Tools|Generic Import** since it will create the definitional object before the contents are imported. (See the next section labeled Generic Import below).

1) Using the Session Browser, select the correct object type. If the file being imported is a source version, you should bring up the appropriate source editor and select the version you want to import into.

2) Invoke the **Import** command from the object or source version’s popup menu (or the main menu **Object|Import** command).

3) The Import dialog will appear similar to the one shown below. Select the directory that the import file resides in.

4) The export file will appear in the list box (Library import example shown below). Double click on it or select it and click on the **Import** button to start the import.



Look in: Xfer

User.opl

File name: User

Files of type: Libary Export (*.opl)

The **Look in** field lets you select the external folder that the file resides in.

This list box displays existing files of the same type selected in the **File of type** field.

This field displays the **File name** that will be used for the file. In the case shown, it is the name of the User library.

The **Import** button is used to import the selected object and its constituent components.

This field displays the file extension generated based on the selected object type.

The **Cancel** button is used to abort the import operation. If pressed, the import will not proceed.

Based on the definitional object type selected, the **Options** area may contain check boxes used to specialize the import.

Generic Import

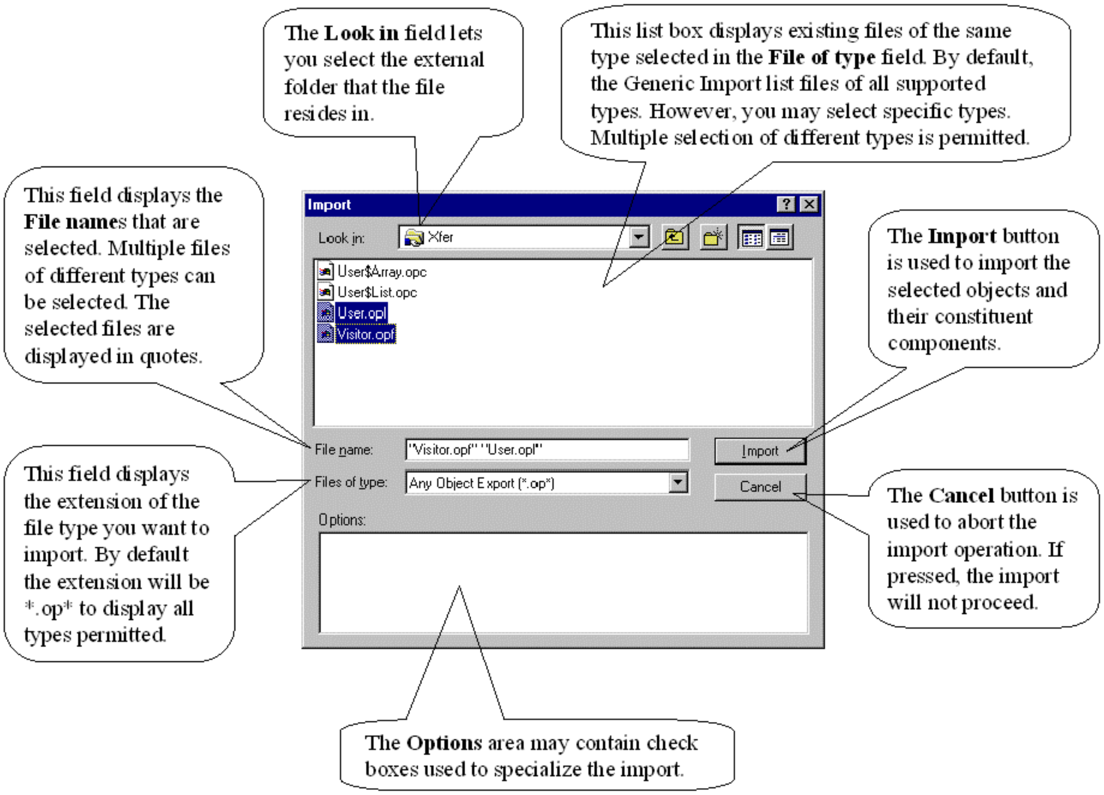
To generically import definitional objects, follow the instruction below.

1) From the main menu, execute the Tool|Generic Import command.

2) The Import dialog box shown below will appear. Using the ‘Look in’ combo box, select the external directory that contains the export file. All supported file types will appear.

3) You may select multiple files by holding the Ctrl key down and clicking on each file name. Alternatively, you may hold the Shift key down and select a range of files.

Note: Classes and any constituent objects contained in the files need not exist in the library structure—they will be automatically created when generically imported. For example, generically importing a property will automatically create the interface and class if they do not exist in the library the property came from. A library will not be recreated under these circumstances. If you want a library recreated automatically, you must export the full library with the **Generate Class Library Bootstrap** box checked in the Export dialog.



The **Look in** field lets you select the external folder that the file resides in.

This list box displays existing files of the same type selected in the **File of type** field. By default, the Generic Import list files of all supported types. However, you may select specific types. Multiple selection of different types is permitted.

This field displays the **File names** that are selected. Multiple files of different types can be selected. The selected files are displayed in quotes.

The **Import** button is used to import the selected objects and their constituent components.

This field displays the extension of the file type you want to import. By default the extension will be *.op* to display all types permitted.

The **Cancel** button is used to abort the import operation. If pressed, the import will not proceed.

The **Options** area may contain check boxes used to specialize the import.

4) Click on the **Import** button to start the import.

5) At this point a dialog box will appear. It will ask you if you want to continue. You have the opportunity to terminate the import at this point. Selecting **Yes** will continue the import. Selecting **No** will terminate the import. If you chose to continue, the Import status dialog box will appear giving you a running status of the import. You have the opportunity to terminate the import at an point by clicking on the **Cancel** button.

Warning: If you terminate the import, your definitional database will be left in an indeterminate state. You will be responsible for rolling it back to it’s original state if that is required.

The AutoLoad Method

If a class with the method 'Factory::AutoLoad' is imported, then this method will be automatically executed after the entire class has been loaded successfully. (This method is not run if the import is canceled before the class has finished loading.)

This is a convenient way of doing specialized processing of a class if required.

Indirect Import and Export

EsiObjects supports Folder structures within the Session Browser. To learn more about Folders, see the Folder Operations and Folder Content Editor sections of this guide.

In a nutshell, folders permit a programmer to store pointers to definitional objects within a library structure. Import and Export operations can be performed on the whole folder or individually on each object pointer stored in the folder.

Export Option

Exporting from the Folder Structure

EsiObjects implements two types of folders: Shared and Private. Shared folders are available within a session. Anyone who connects to the session will be able to use the folder. Private folders are private to each user. Any other user signed onto the session does not see folders private to your session.

Follow these instructions to export the contents of a folder.

1) Select the folder you wish to export using the Session Browser.
2) Select the Export command (or execute the Object|Export command) from the folder's popup menu.
3) The Export dialog will appear as illustrated below for either a Shared or Private folder.

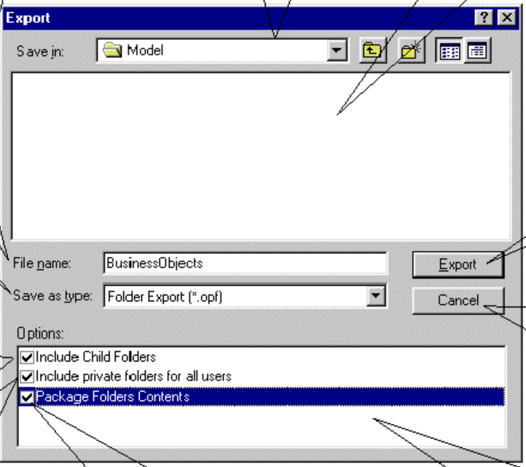
Note: The check boxes that appear in the Options area are specific to the type of folder being exported.

Shared Folders

This field displays the name that will be used for the file. In the case shown, it is generated from the library name (Base) followed by a $ delimiter and then the class name (Collection).

The **Save in** field lets you select the external folder that the file is to be saved to.

This list box displays existing files of the same type selected in the **Save as type** field.



Export

Save in: Model

File name: BusinessObjects

Save as type: Folder Export (*.opf)

Options:

☑ Include Child Folders
☑ Include private folders for all users
☑ Package Folders Contents

This field displays the file extension generated based on the selected object type.

When the **Include Child Folders** checkbox is checked, the export function will export all child folders along with the selected folder

The **Export** button is used to export the selected object and its constituent components if selected.

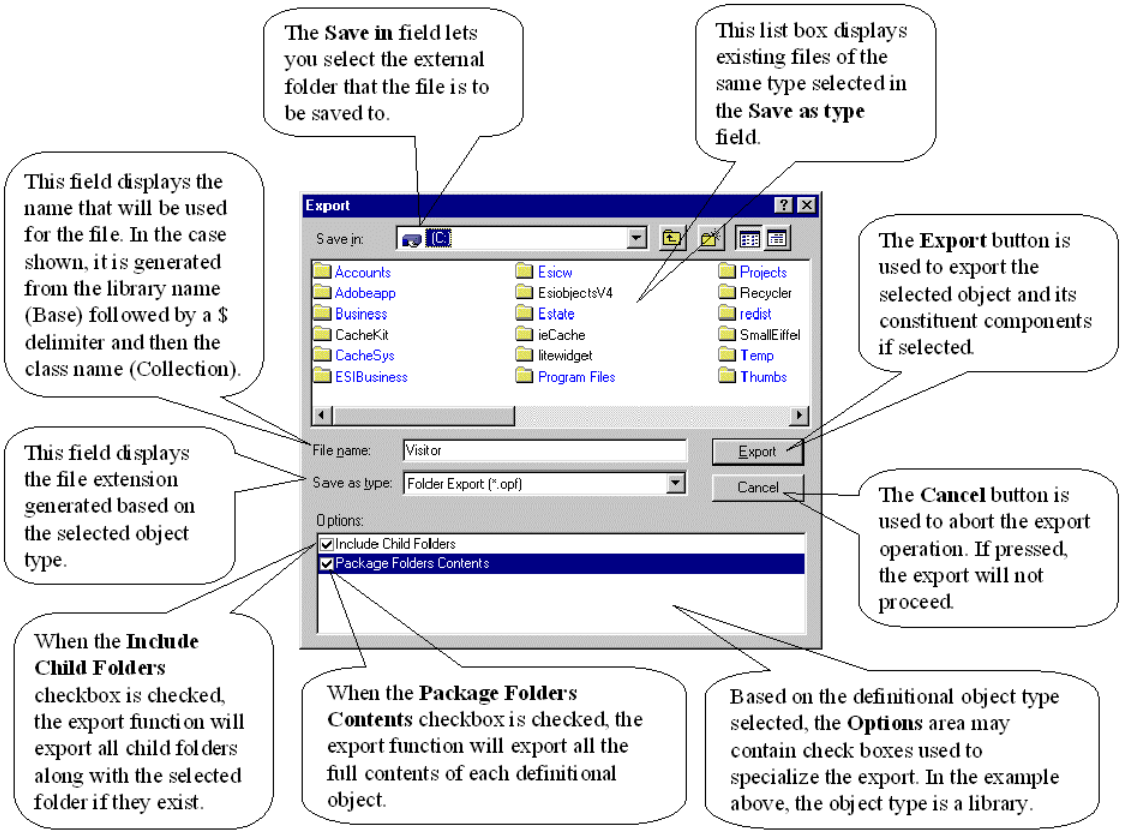
The **Cancel** button is used to abort the export operation. If pressed, the export will not proceed.

When the **Include private folders for all users** checkbox is checked, the export function will export all user's private folders as well as the selected folder..

When the **Package Folders Contents** checkbox is checked, the export function will export all the full contents of each definitional object.

Based on the definitional object type selected, the **Options** area may contain check boxes used to specialize the export. In the example above, the object type is a library.

Private Folders



Save in: (C:)

File name: Visitor

Save as type: Folder Export (*.opf)

☑ Include Child Folders

☑ Package Folders Contents

The **Save in** field lets you select the external folder that the file is to be saved to.

This list box displays existing files of the same type selected in the **Save as type** field.

This field displays the name that will be used for the file. In the case shown, it is generated from the library name (Base) followed by a $ delimiter and then the class name (Collection).

The **Export** button is used to export the selected object and its constituent components if selected.

This field displays the file extension generated based on the selected object type.

The **Cancel** button is used to abort the export operation. If pressed, the export will not proceed.

When the **Include Child Folders** checkbox is checked, the export function will export all child folders along with the selected folder if they exist.

When the **Package Folders Contents** checkbox is checked, the export function will export all the full contents of each definitional object.

Based on the definitional object type selected, the **Options** area may contain check boxes used to specialize the export. In the example above, the object type is a library.

4) Once all fields have been filled in properly, click on the **Export** button to export. At this point you may chose to abort the operation by clicking on the **Cancel** button.

A status dialog box will display the progress of the export. Once the status box disappears, the export will be complete. You may click on the **Cancel** button of the status dialog to abort the export at any time.

Exporting from the Folder Content Editor

Exporting a definitional object from the Folder Content Editor is the same as exporting the object directly from the library structure.

Follow the instructions for exporting in the Direct Import and Export section.

Import Option

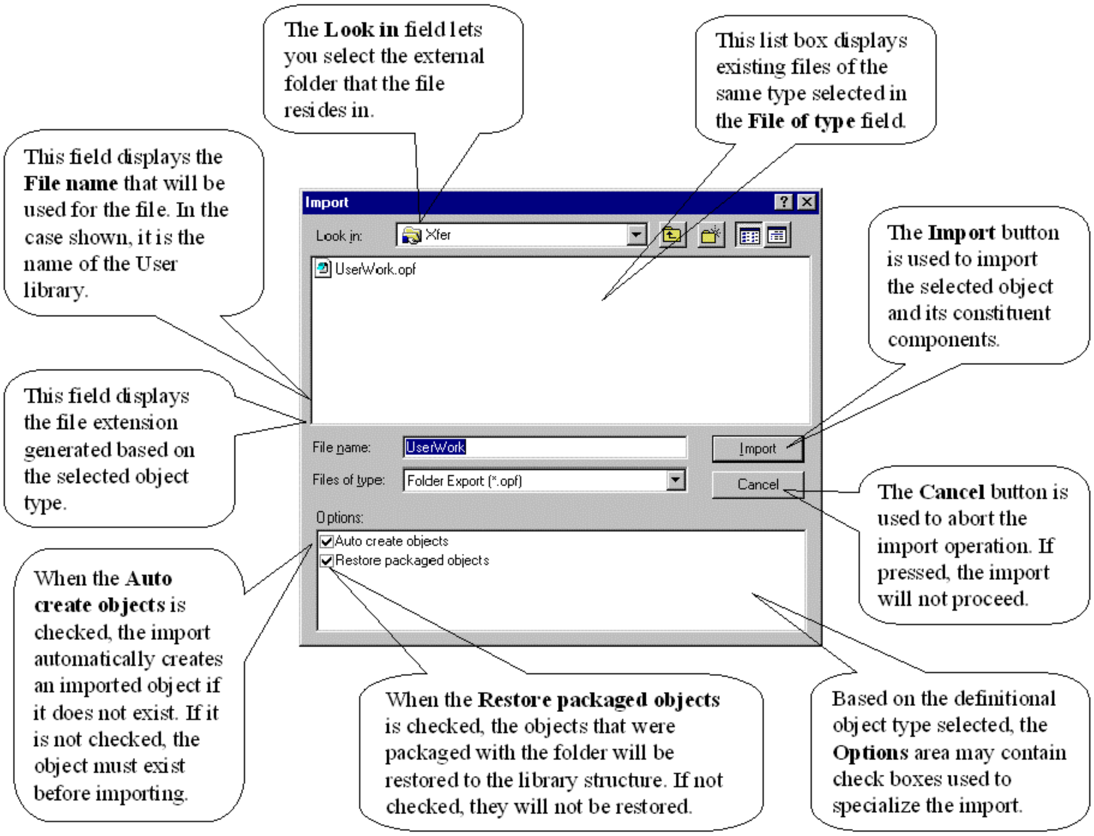
Importing from the Folder Structure

To import a folder follow the instruction below.

1) Using the Session Browser, select the folder you want to import into.

2) Invoke the **Import** command from the folder's popup menu (or the main menu **Object|Import** command).

3) The Import dialog will appear similar to the one shown below. Select the directory that the import file resides in.



Look in: Xfer

UserWork.opf

File name: UserWork

Files of type: Folder Export (*.opf)

☑ Auto create objects

☑ Restore packaged objects

The **Look in** field lets you select the external folder that the file resides in.

This list box displays existing files of the same type selected in the **File of type** field.

This field displays the **File name** that will be used for the file. In the case shown, it is the name of the User library.

The **Import** button is used to import the selected object and its constituent components.

This field displays the file extension generated based on the selected object type.

The **Cancel** button is used to abort the import operation. If pressed, the import will not proceed.

When the **Auto create objects** is checked, the import automatically creates an imported object if it does not exist. If it is not checked, the object must exist before importing.

When the **Restore packaged objects** is checked, the objects that were packaged with the folder will be restored to the library structure. If not checked, they will not be restored.

Based on the definitional object type selected, the **Options** area may contain check boxes used to specialize the import.

4) The exported file will appear in the list box (Folder import example shown below). Double click on it or select it and click on the **Import** button to start the import.

Importing from the Folder Content Editor

Importing a definitional object from the Folder Content Editor is the same as importing the object directly from the library structure.

Follow the instructions for importing in the <u>Direct Import and Export</u> section.

Generic Import

Generically importing a folder is the same as importing an object directly. The only difference is it is a folder that is being imported (file extension .opf) as opposed to a definitional object.

Follow the instructions for generically importing in the <u>Direct Import and Export</u> section.

Using Instance and Legacy Transport Tools

Traditional M Tools have been added to the EsiObjects toolbox to assist you in importing and exporting traditional M routines and globals. They were never designed to provide the full functionality that you would expect in a traditional M development environment. EsiObjects is an object oriented development environment. These tools are primarily available for:

- Importing and exporting legacy M database routines and globals
- Importing and exporting object instance data contained in globals.
- Editing legacy routines.

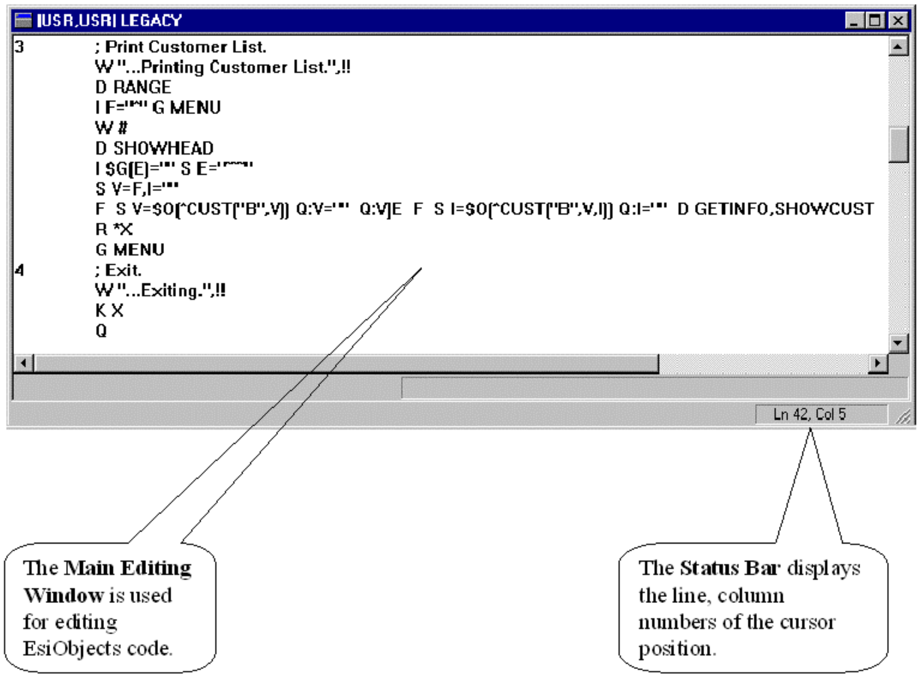
Using Routine Tools

Using the Routine Editor

There are two ways to invoke the EsiObjects Routine Editor:

- To create a new routine, use the following menu path: **Tools | Routines | Editor**.
- To edit an existing routine, **double-click** on that routine's name in the Routine Directory window.

The EsiObjects Routine Editor provides a simple editing window for creating and modifying routine source code. The picture below illustrates the component parts of the Routine Editor.



```
3           ; Print Customer List.
            W "...Printing Customer List.",!!
            D RANGE
            I F="^" G MENU
            W #
            D SHOWHEAD
            I $G(E)="" S E="^~~~"
            S V=F,I=""
            F  S V=$O(^CUST("B",V)) Q:V=""  Q:V]E  F  S I=$O(^CUST("B",V,I)) Q:I=""  D GETINFO,SHOWCUST
            R *X
            G MENU
4           ; Exit.
            W "...Exiting.",!!
            K X
            Q
```

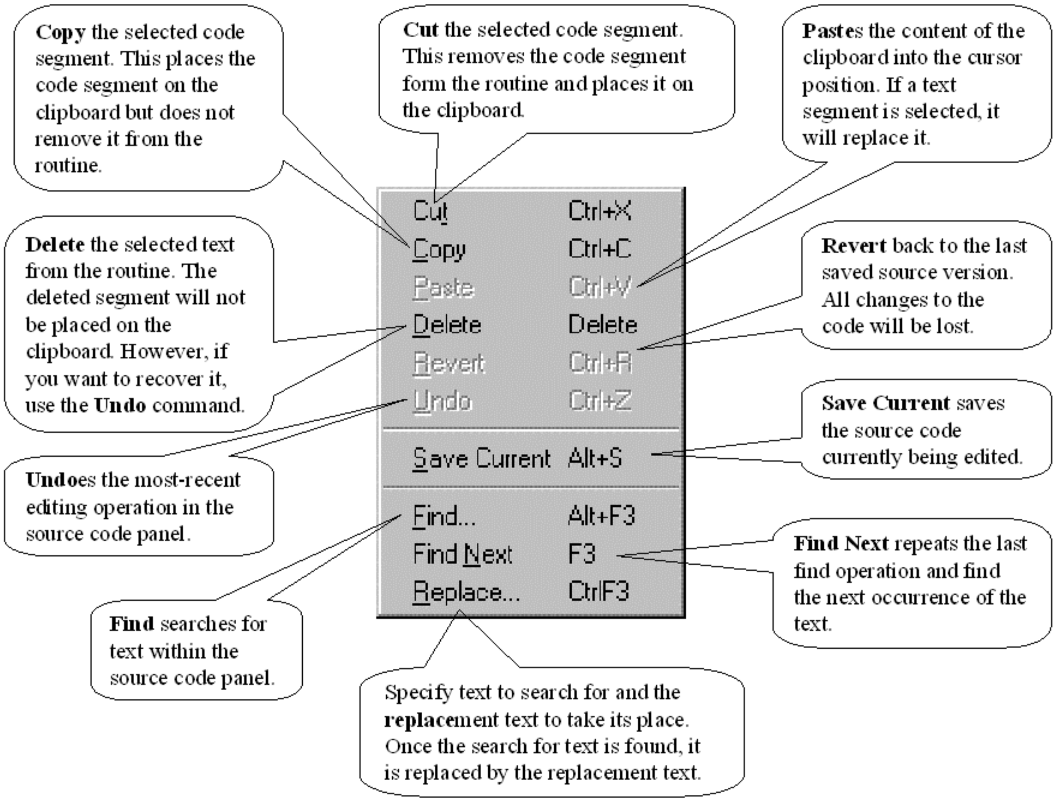
The **Main Editing Window** is used for editing EsiObjects code.

The **Status Bar** displays the line, column numbers of the cursor position.

To **save changes**, you can right click to invoke a popup menu, or simply type **Alt+S**. A variety of other useful editing functions are also available through the popup menu.

Routine Source Code Popup Menu

The Routine Source Code popup menu is invoked by right clicking (or pressing **Shift+F10**) inside the source code pane of the Routine Editor.



Copy the selected code segment. This places the code segment on the clipboard but does not remove it from the routine.

Cut the selected code segment. This removes the code segment form the routine and places it on the clipboard.

Pastes the content of the clipboard into the cursor position. If a text segment is selected, it will replace it.

Delete the selected text from the routine. The deleted segment will not be placed on the clipboard. However, if you want to recover it, use the **Undo** command.

Revert back to the last saved source version. All changes to the code will be lost.

Save Current saves the source code currently being edited.

Undoes the most-recent editing operation in the source code panel.

Find Next repeats the last find operation and find the next occurrence of the text.

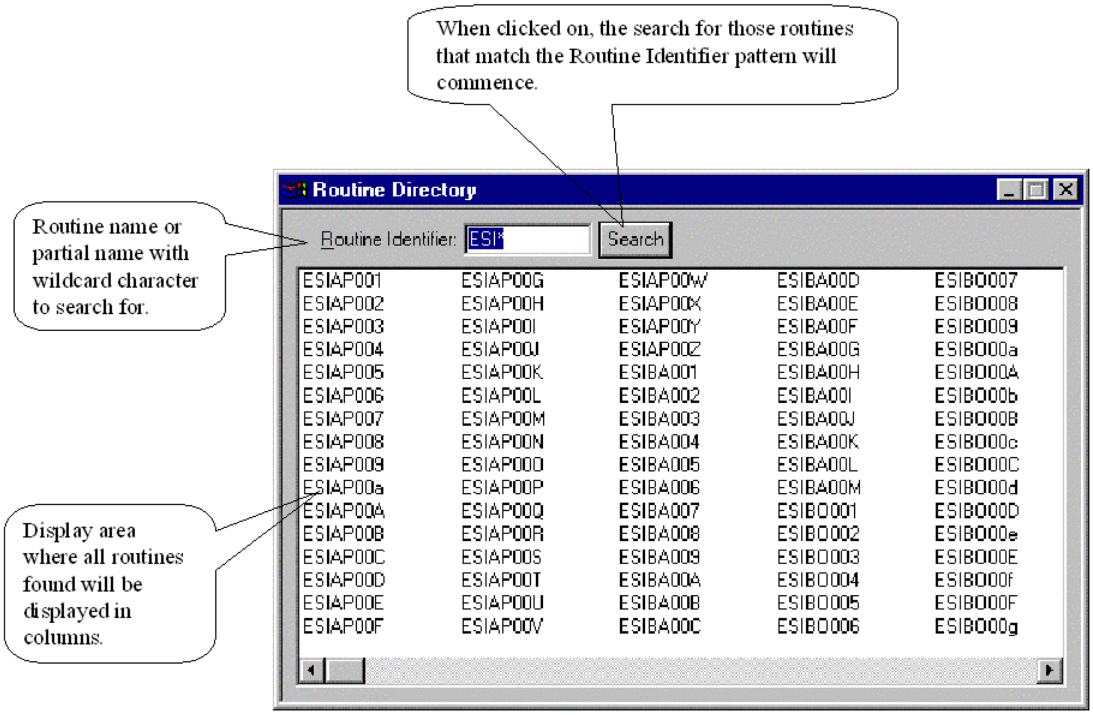
Find searches for text within the source code panel.

Specify text to search for and the **replacement** text to take its place. Once the search for text is found, it is replaced by the replacement text.

Using the Routine Directory

This utility is used to view a routine directory listing:

To view a listing of routines, use the following menu path: **Tools | Routine Tools | Routine Directory**.



When clicked on, the search for those routines that match the Routine Identifier pattern will commence.

Routine name or partial name with wildcard character to search for.

Display area where all routines found will be displayed in columns.

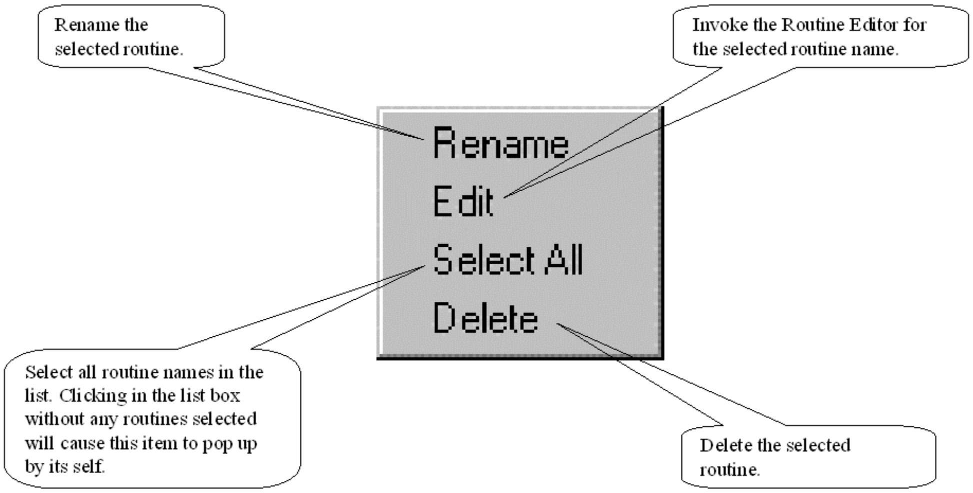
```
ESIAP001   ESIAP00G   ESIAP00W   ESIBA00D   ESIBO007
ESIAP002   ESIAP00H   ESIAP00X   ESIBA00E   ESIBO008
ESIAP003   ESIAP00I   ESIAP00Y   ESIBA00F   ESIBO009
ESIAP004   ESIAP00J   ESIAP00Z   ESIBA00G   ESIBO00a
ESIAP005   ESIAP00K   ESIBA001   ESIBA00H   ESIBO00A
ESIAP006   ESIAP00L   ESIBA002   ESIBA00I   ESIBO00b
ESIAP007   ESIAP00M   ESIBA003   ESIBA00J   ESIBO00B
ESIAP008   ESIAP00N   ESIBA004   ESIBA00K   ESIBO00c
ESIAP009   ESIAP00O   ESIBA005   ESIBA00L   ESIBO00C
ESIAP00a   ESIAP00P   ESIBA006   ESIBA00M   ESIBO00d
ESIAP00A   ESIAP00Q   ESIBA007   ESIBO001   ESIBO00D
ESIAP00B   ESIAP00R   ESIBA008   ESIBO002   ESIBO00e
ESIAP00C   ESIAP00S   ESIBA009   ESIBO003   ESIBO00E
ESIAP00D   ESIAP00T   ESIBA00A   ESIBO004   ESIBO00f
ESIAP00E   ESIAP00U   ESIBA00B   ESIBO005   ESIBO00F
ESIAP00F   ESIAP00V   ESIBA00C   ESIBO006   ESIBO00g
```

Type an identifier to narrow the search, and click the Search button. A listing of the matching routines then appears in the list box below. The following special characters may be used in specifying a search identifier:

Char.	Description
*	Wild Card (anything or nothing can go here.)
%	Single Character (any single character can go here.)

Routine Directory Popup Menu

The Routine Directory popup menu is invoked by right clicking (or pressing **Shift+F10**) inside the routine list pane of the Routine Directory.



Rename the selected routine.

Invoke the Routine Editor for the selected routine name.

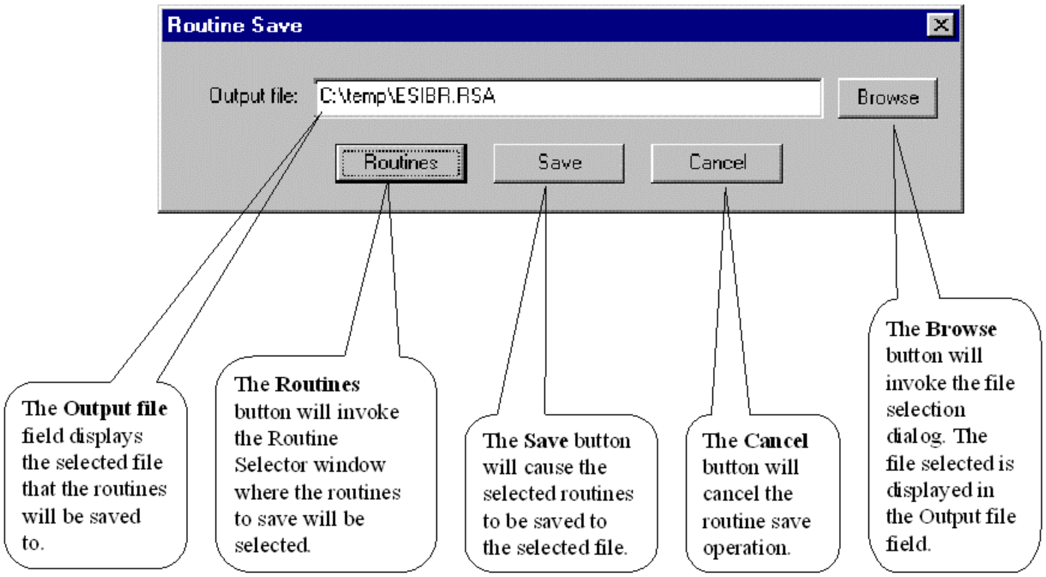
Select all routine names in the list. Clicking in the list box without any routines selected will cause this item to pop up by its self.

Delete the selected routine.

Using the Routine Save

This utility is used to export one or more routines to a formatted file.

To create an export file containing routines, use the following menu path: **Tools | Routine Tools | Routine Save**. A dialog appears, allowing the user to specify an output file name.



Output file: C:\temp\ESIBR.RSA

The **Output file** field displays the selected file that the routines will be saved to.

The **Routines** button will invoke the Routine Selector window where the routines to save will be selected.

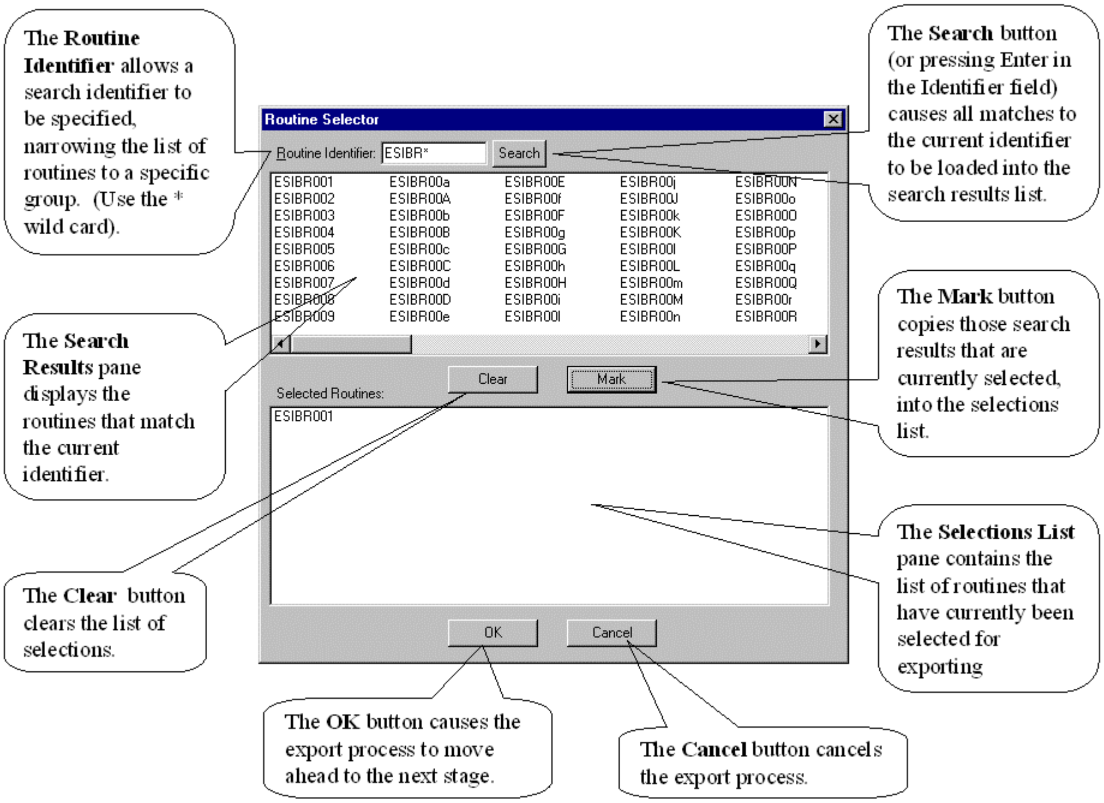
The **Save** button will cause the selected routines to be saved to the selected file.

The **Cancel** button will cancel the routine save operation.

The **Browse** button will invoke the file selection dialog. The file selected is displayed in the Output file field.

Click the **Routines** button.

A Routine Selector window appears like that illustrated below, allowing the user to specify exactly which routines to export.



Routine Identifier: ESIBR*

```
ESIBR001     ESIBR00a     ESIBR00E     ESIBR00j     ESIBR00N
ESIBR002     ESIBR00A     ESIBR00f     ESIBR00J     ESIBR00o
ESIBR003     ESIBR00b     ESIBR00F     ESIBR00k     ESIBR00O
ESIBR004     ESIBR00B     ESIBR00g     ESIBR00K     ESIBR00p
ESIBR005     ESIBR00c     ESIBR00G     ESIBR00l     ESIBR00P
ESIBR006     ESIBR00C     ESIBR00h     ESIBR00L     ESIBR00q
ESIBR007     ESIBR00d     ESIBR00H     ESIBR00m     ESIBR00Q
ESIBR008     ESIBR00D     ESIBR00i     ESIBR00M     ESIBR00r
ESIBR009     ESIBR00e     ESIBR00I     ESIBR00n     ESIBR00R
```

Selected Routines: ESIBR001

The **Routine Identifier** allows a search identifier to be specified, narrowing the list of routines to a specific group. (Use the * wild card).

The **Search** button (or pressing Enter in the Identifier field) causes all matches to the current identifier to be loaded into the search results list.

The **Search Results** pane displays the routines that match the current identifier.

The **Mark** button copies those search results that are currently selected, into the selections list.

The **Clear** button clears the list of selections.

The **Selections List** pane contains the list of routines that have currently been selected for exporting

The **OK** button causes the export process to move ahead to the next stage.

The **Cancel** button cancels the export process.

There are three ways to mark routines using this dialog:

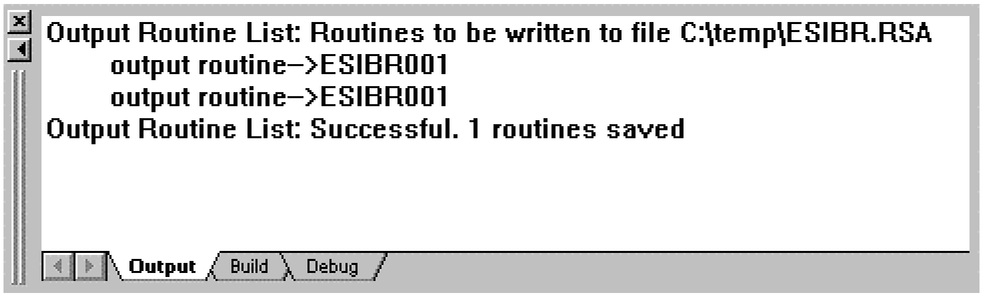
1) **Clicking on Buttons.** The **Mark** button adds selected names to the export list, while the **Clear** button removes all the names from the export list.

2) **Double-Clicking on Names.** Double-clicking on a name in the name list at top, adds that name to the export list. Double-clicking on a name in the export list at bottom, removes that name from the export list.

3) **Popup Menus.** When you have selected a group of names, you can right click on them (or press **Shift+F10**) to invoke a popup menu with useful options.

After the selector window is closed successfully, the earlier file dialog re-appears, allowing the user to specify an output file name.

Clicking the **Save** button causes the routines to be exported to the specified file. The Output Window shown below displays the save progress.



```
Output Routine List: Routines to be written to file C:\temp\ESIBR.RSA
      output routine->ESIBR001
      output routine->ESIBR001
Output Routine List: Successful. 1 routines saved
```

Using Routine Restore

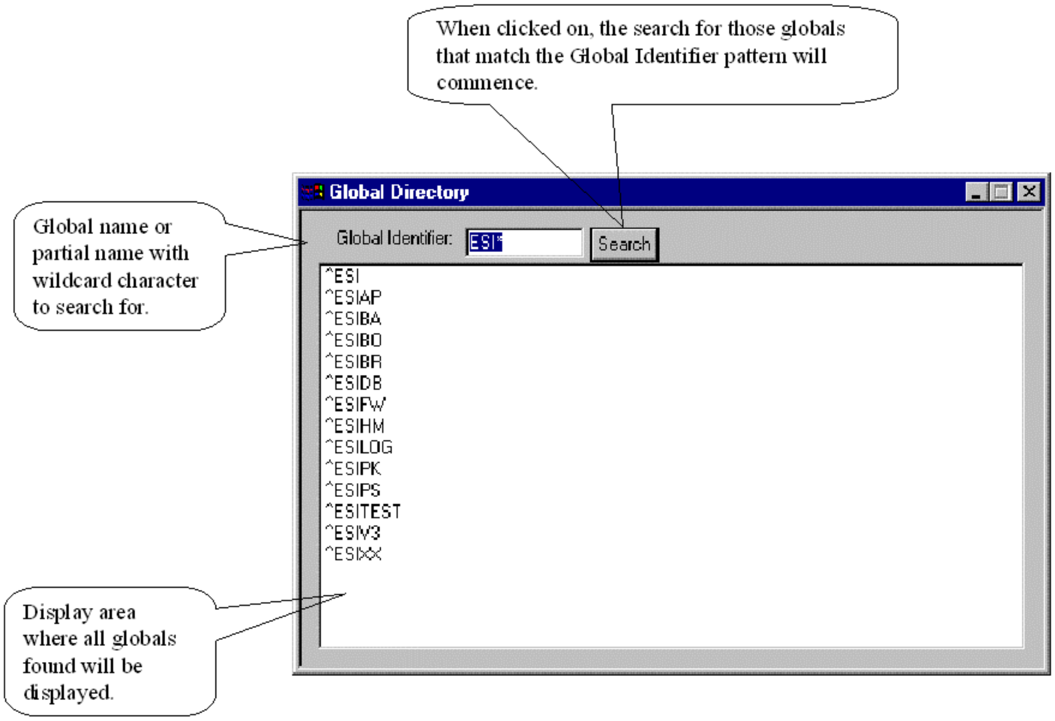
When Routines are restored, you will be prompted via a common Windows file selection dialog box to select a file that contains routines to be restored. Once selected the routines will restore and the status will print in the Output Window.

Note: it is important to be careful ***not*** to accidentally use the **Global Restore** option to import **Routines**, or vice versa.

Using the Global Tools

Using the Global Directory

This utility is used to view a global directory listing. To view a listing of globals, use the following menu path: **Tools | Global Tools | Global Directory**.



Global Identifier: ESI*

```
^ESI
^ESIAP
^ESIBA
^ESIBO
^ESIBR
^ESIDB
^ESIFW
^ESIHM
^ESILOG
^ESIPK
^ESIPS
^ESITEST
^ESIV3
^ESIXX
```

When clicked on, the search for those globals that match the Global Identifier pattern will commence.

Global name or partial name with wildcard character to search for.

Display area where all globals found will be displayed.

Type an identifier to narrow the search, and click the Search button. A listing of the matching globals then appears in the list box below. The following special characters may be used in specifying a search identifier:

Char.	Description
*	Wild Card (anything or nothing can go here.)
%	Single Character (any single character can go here.)

In the global directory, the ^ prefix is optional. In other words, **^ESI*** and **ESI*** will both produce a listing of all globals whose names begin with the three letters, **ESI**.

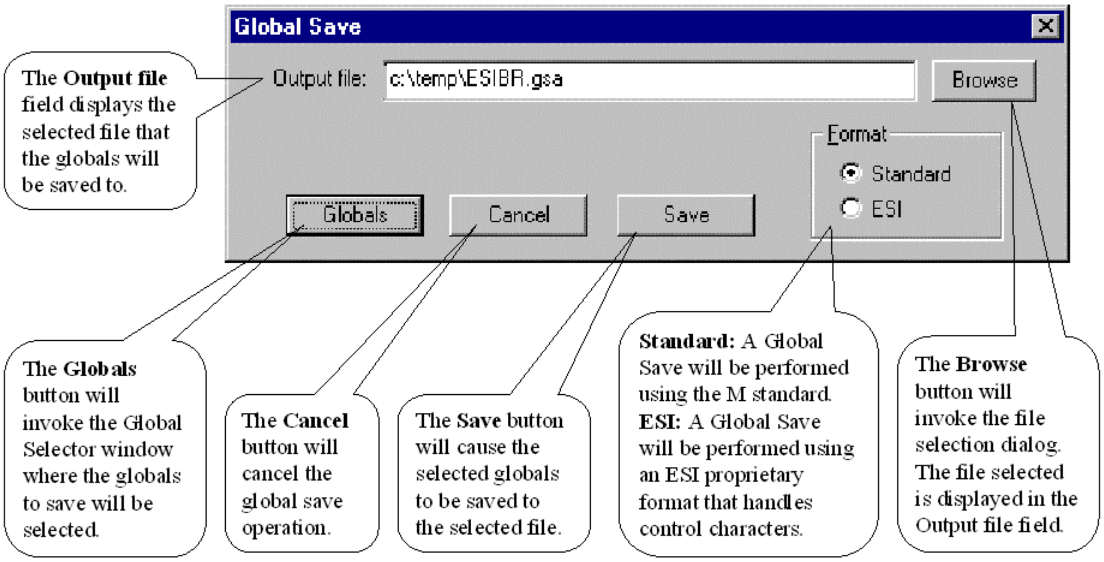
If there are many matching entries, the list will take longer to produce. The following strategies are helpful in reducing the search time:

- Specify a prefix. A* (all routines beginning with A) will run considerably faster than *A (all routines ending with A).
- Narrow the search as much as possible. A search with few matches will run faster than a search with many matches.

Note that EsiObjects does ***not*** allow you to see M routines and globals whose name begins with the % character.

Using the Global Save

This utility is used to export one or more globals to a formatted file. To create an export file containing globals, use the following menu path: **Tools | Global Tools | Global Save**. A dialog appears, allowing the user to specify an output file name.



Global Save

Output file: c:\temp\ESIBR.gsa

Format
● Standard
○ ESI

The **Output file** field displays the selected file that the globals will be saved to.

The **Globals** button will invoke the Global Selector window where the globals to save will be selected.

The **Cancel** button will cancel the global save operation.

The **Save** button will cause the selected globals to be saved to the selected file.

Standard: A Global Save will be performed using the M standard. **ESI:** A Global Save will be performed using an ESI proprietary format that handles control characters.

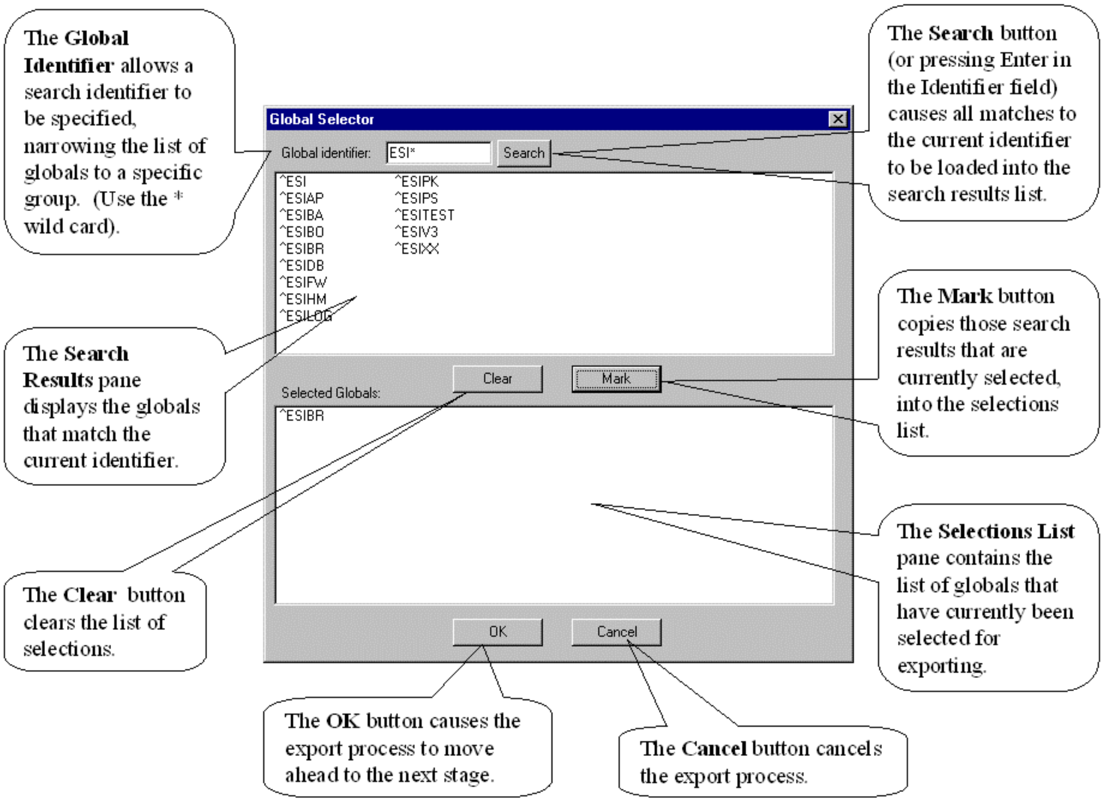
The **Browse** button will invoke the file selection dialog. The file selected is displayed in the Output file field.

There are two choices for the save file format:

1) **Standard:** A Global Save will be performed using the M standard. If a subscript of one of the globals selected or any of the values in the globals contains control characters, then the save operation should use ESI format.

2) **ESI:** A Global Save will be performed using an ESI proprietary format that handles control characters. This format is unknown to M and only understood by EsiObjects.

Click the **Globals** button. A Global Selector window appears, allowing the user to specify exactly which globals to export.



Global identifier: ESI*

Selected Globals: ^ESIBR

The **Global Identifier** allows a search identifier to be specified, narrowing the list of globals to a specific group. (Use the * wild card).

The **Search** button (or pressing Enter in the Identifier field) causes all matches to the current identifier to be loaded into the search results list.

The **Search Results** pane displays the globals that match the current identifier.

The **Mark** button copies those search results that are currently selected, into the selections list.

The **Clear** button clears the list of selections.

The **Selections List** pane contains the list of globals that have currently been selected for exporting.

The **OK** button causes the export process to move ahead to the next stage.

The **Cancel** button cancels the export process.

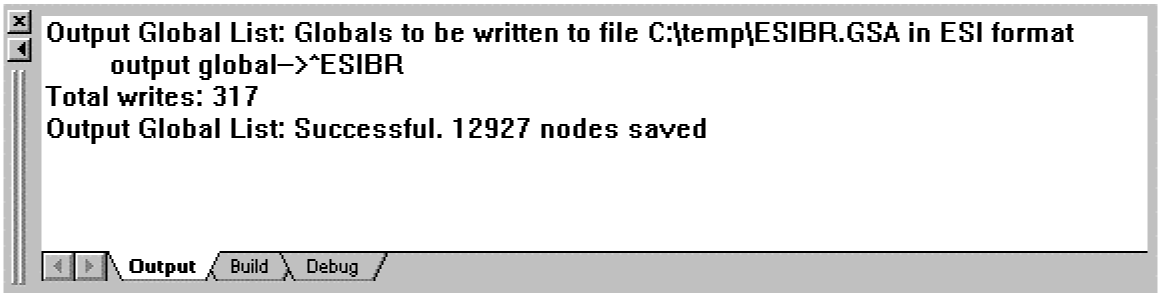
There are three ways to mark globals using this dialog:

1) **Clicking on Buttons.** The **Mark** button adds selected names to the export list, while the **Clear** button removes all the names from the export list.

2) **Double-Clicking on Names.** Double-clicking on a name in the name list at top, adds that name to the export list. Double-clicking on a name in the export list at bottom, removes that name from the export list.

3) **Popup Menus.** When you have selected a group of names, you can right click on them (or press **Shift+F10**) to invoke a popup menu with useful options.

After the selector window is closed successfully, the earlier file dialog re-appears, allowing the user to specify an output file name. Clicking the **Save** button causes the globals to be exported to the specified file. The Output Window shown below displays the save progress.



```
Output Global List: Globals to be written to file C:\temp\ESIBR.GSA in ESI format
      output global–>^ESIBR
Total writes: 317
Output Global List: Successful. 12927 nodes saved
```

Using the Global Restore

When Globals are restored, the user is allowed to specify a file name, containing the globals to be imported. Once the file name has been specified, the process of importing globals can begin.

Note: it is important to be careful ***not*** to accidentally use the **Routine Restore** option to import **Globals**, or vice versa.

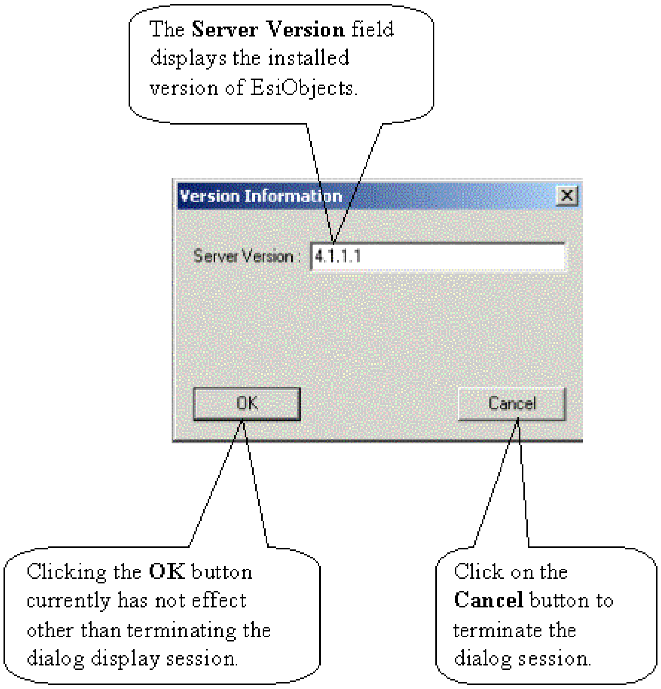
Management Tools

If the developer starts up the EsiObjects Development UI on a client PC with the /Admin (or /ESI) command line qualifiers, access to management functions are permitted through the **Management** main menu option. The Management menu option is an add-in and will only appear if these qualifiers are specified. There are two commands supported at this level.

1. Version, which simply displays the current version of EsiObjects.
2. Security, which provides a command path to system and user level security commands.

Version

The Version command simply displays the installed version of EsiObjects.



Version Information

Server Version : 4.1.1.1

The **Server Version** field displays the installed version of EsiObjects.

Clicking the **OK** button currently has not effect other than terminating the dialog display session.

Click on the **Cancel** button to terminate the dialog session.

Security

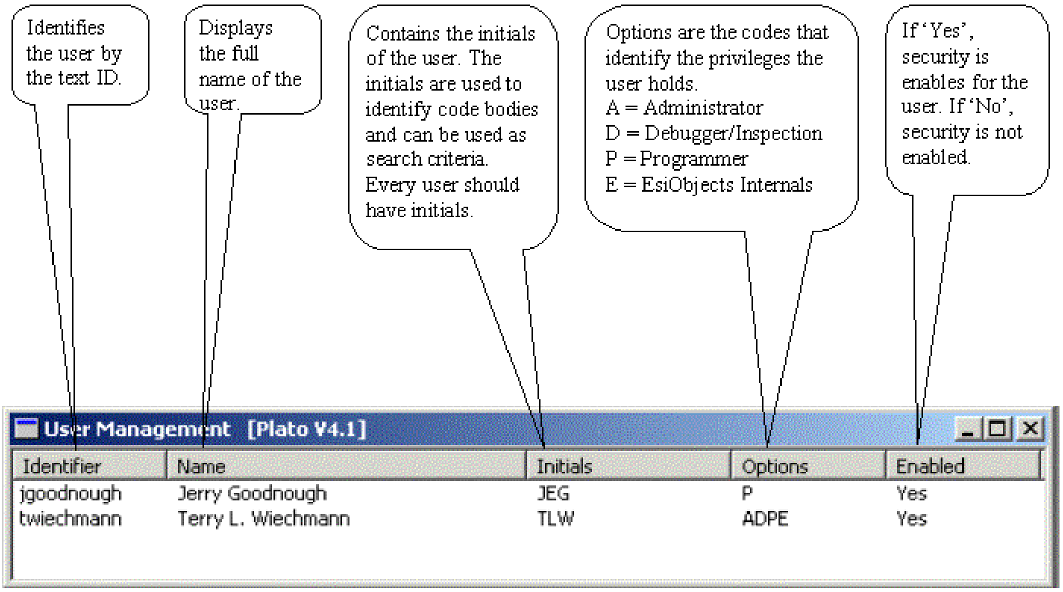
EsiObjects supports a simple username and password security scheme. There are two areas supported by this scheme.

3. **User Management** is where all the commands reside for creating and maintaining the user files on each server (session). The **User Management** command will invoke a window that displays each user in the rows and their attributes as columns. This window permits all the editing functions.
4. **Policies** is where system level security is maintained.

User Management

The User Management command is a graphical tool used to maintain the user's information stored on a particular M server.

The following illustrates the User Management window and its component parts.



Identifies the user by the text ID.

Displays the full name of the user.

Contains the initials of the user. The initials are used to identify code bodies and can be used as search criteria. Every user should have initials.

Options are the codes that identify the privileges the user holds.
A = Administrator
D = Debugger/Inspection
P = Programmer
E = EsiObjects Internals

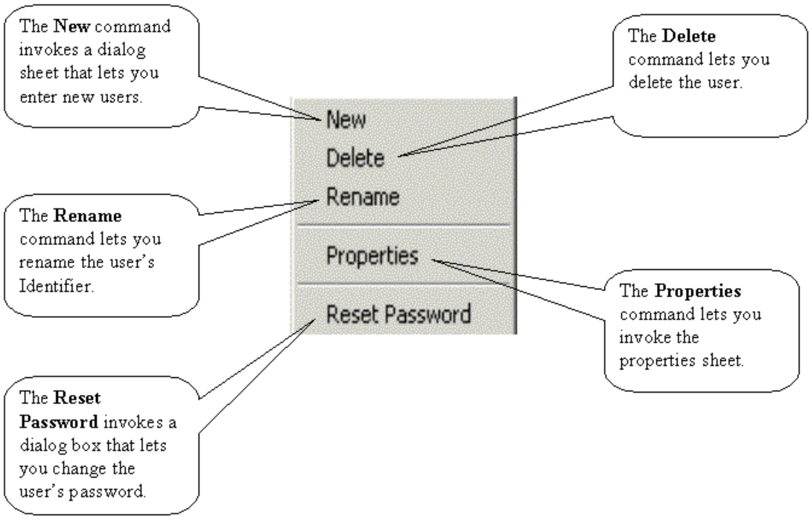
If 'Yes', security is enables for the user. If 'No', security is not enabled.

User Management [Plato V4.1]

Identifier	Name	Initials	Options	Enabled
jgoodnough	Jerry Goodnough	JEG	P	Yes
twiechmann	Terry L. Wiechmann	TLW	ADPE	Yes

Using the Popup Menu Commands

The User Management window supports a popup menu. The menu lets you perform all the functions of the VESOTCMN program. The menu is illustrated below.



The **New** command invokes a dialog sheet that lets you enter new users.

The **Delete** command lets you delete the user.

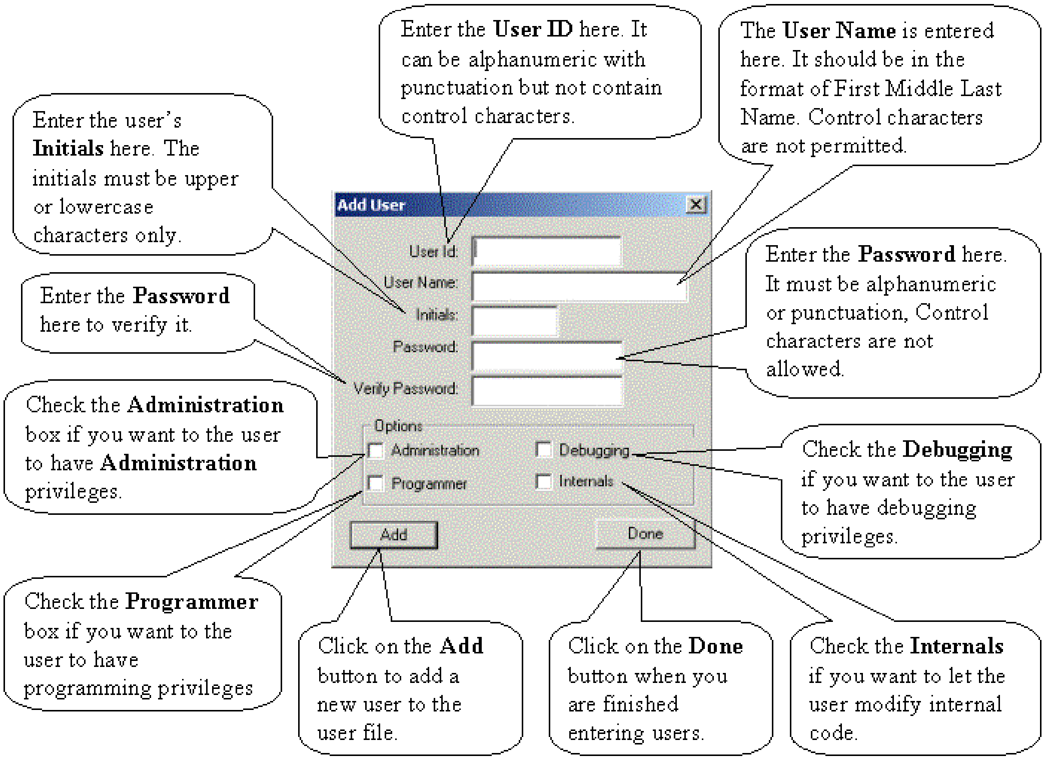
The **Rename** command lets you rename the user's Identifier.

The **Properties** command lets you invoke the properties sheet.

The **Reset Password** invokes a dialog box that lets you change the user's password.

Using the New Command

The New command is used to add new users to the user file. The Add User dialog is illustrated below. All fields are described in the callouts. **When a new user is added to the server, security is, by default, turned on for that user.**



Enter the user's **Initials** here. The initials must be upper or lowercase characters only.

Enter the **User ID** here. It can be alphanumeric with punctuation but not contain control characters.

The **User Name** is entered here. It should be in the format of First Middle Last Name. Control characters are not permitted.

Enter the **Password** here to verify it.

Enter the **Password** here. It must be alphanumeric or punctuation, Control characters are not allowed.

Check the **Administration** box if you want to the user to have **Administration** privileges.

Check the **Debugging** if you want to the user to have debugging privileges.

Check the **Programmer** box if you want to the user to have programming privileges

Click on the **Add** button to add a new user to the user file.

Click on the **Done** button when you are finished entering users.

Check the **Internals** if you want to let the user modify internal code.

Using the Delete Command

The **Delete** command is used to expunge a user from the user file. Selecting the **Delete** command will invoke a dialog that lets to verify the deletion. Deleting a user is final. To put the user back into the system, you must use the New command and reenter all the user information.

Using the Rename Command

The **Rename** command is used to change the user's Identifier only. Selecting this command will change the selected Identifier in the User Management window to rename mode. At that point you can change the identifier.

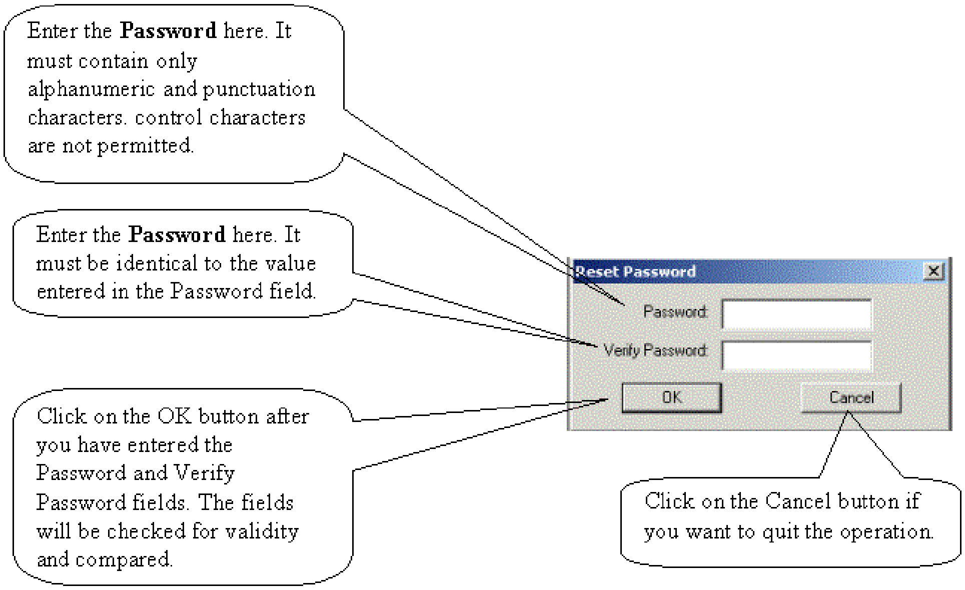
Using the Properties Command

The **Properties** command will invoke the user's property sheet. The sheet is illustrated below.

Picture goes here!

Using the Reset Password Command

The **Reset Password** command lets you change the user's password. When you enter a new password, you must enter the Verify Password field as well. The Password is checked for validity and then compared with the Verify Password entry. If they pass, the password is changed. If not, you will receive an error dialog.



Enter the **Password** here. It must contain only alphanumeric and punctuation characters. control characters are not permitted.

Enter the **Password** here. It must be identical to the value entered in the Password field.

Click on the OK button after you have entered the Password and Verify Password fields. The fields will be checked for validity and compared.

Click on the Cancel button if you want to quit the operation.

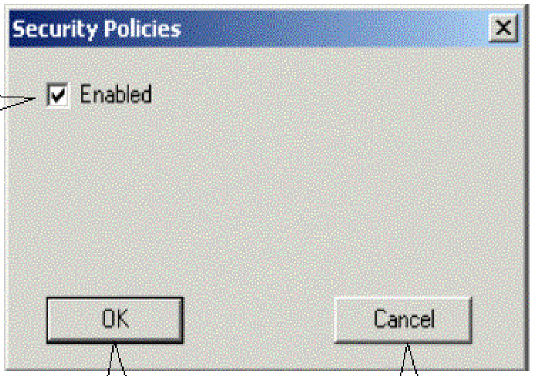
Policies

Security can be enabled and disabled at two levels within the EsiObjects system. First, it can be enabled or disabled at the system level. Second, it can be enabled or disabled for each user.

Selecting the **Policies** command invokes a dialog box that lets you enable or disable security checks for everyone at the system level. When enabled, all users will be required to enter a username (Identifier) and password if security has been enabled at the user level. When disabled, no security checks will be made.

The Security Policies dialog is illustrated below with descriptions.

Check the **Enable** box to enable security on the server. Clear the box to disable security checks.



☑ Enabled

Click on the **OK** button to change the security setting on the server.

Click on the **Cancel** button if you want to avoid making changes.