WEBLOAD
Scripting Guide

Version 10.2

RADVIEW

WebLOAD Scripting Guide

Table of Contents

Chapter 3. Advanced JavaScript Agenda Features 69

Chapter 4. Working with HTTP Protocol 123

Chapter 1

Introduction

Welcome to WebLOAD, the premier performance, scalability, and reliability testing solution for internet applications.

WebLOAD is easy to use and delivers maximum testing performance and value. WebLOAD verifies the scalability and integrity of internet applications by generating a load composed of Virtual Clients that simulate real-world traffic.

This section provides a brief introduction to WebLOAD technical support, including both documentation and online support.

WebLOAD Documentation

WebLOAD is supplied with the following documentation:

WebLOAD™ Installation Guide

Instructions for installing WebLOAD and its add-ons.

WebLOAD™ IDE User Guide

Instructions for recording, editing, and debugging load test Agendas to be executed by WebLOAD to test your Web-based applications.

WebLOAD™ Console User Guide

A guide to using WebLOAD console, RadView's load/scalability testing tool to easily and efficiently test your Web-based applications. This guide also includes a quick start section containing instructions for getting started quickly with WebLOAD using the RadView Software test site.

WebLOAD™ Analytics User Guide

Instructions on how to use WebLOAD Analytics to analyze data and create custom, informative reports after running a WebLOAD test session.

WebRM™ User Guide

Instructions for managing testing resources with the WebLOAD Resource Manager.

WebLOAD™ Scripting Guide

Complete information on scripting and editing JavaScript Agendas for use in WebLOAD and WebLOAD IDE.

WebLOAD™ JavaScript Reference Guide

Complete reference information on all JavaScript objects, variables, and functions used in WebLOAD and WebLOAD IDE test Agendas.

WebLOAD™ Extensibility SDK

Instructions on how to develop extensions to tailor WebLOAD to specific working environments.

The guides are distributed with the WebLOAD software in online help format. The guides are also supplied as Adobe Acrobat files. View and print these files using the Adobe Acrobat Reader. Install the Reader from the Adobe website http://www.adobe.com.

Typographical Conventions

Before you start using this guide, it is important to understand the terms, icons, and typographical conventions used in the documentation.

The following icons appear next to the text to identify special information.

Table 1: Icon Conventions

Icon	Type of Information
	Indicates a note.
	Indicates a feature that is available only as part of a WebLOAD Add-on.

The following kinds of formatting in the text identify special information.

Table 2: Typographical Conventions

Formatting Convention	Type of Information
Special Bold	Items you must select, such as menu options, command buttons, or items in a list.
Emphasis	Use to emphasize the importance of a point or for variable expressions such as parameters.
CAPITALS	Names of keys on the keyboard. for example, SHIFT, CTRL, or ALT.
KEY+KEY	Key combinations for which the user must press and hold down one key and then press another, for example, CTRL+P, or ALT+F4.

Where to Get More Information

This section contains information on how to obtain technical support from RadView worldwide, should you encounter any problems.

Online Help

WebLOAD provides a comprehensive on-line help system with step-by-step instructions for common tasks.

You can press the F1 key on any open dialog box for an explanation of the options or select **Help ➤ Contents** to open the on-line help contents and index.

Technical Support Website

The technical support pages on our website contain:

- FAQ (Frequently Asked / Answered Questions).
- Agenda Center
- Documentation
- RadView's Product Resource Center, where you can find prepared test scripts, product information, and industry related news.
- http://www.radview.com/support/index.asp

Technical Support

For technical support in your use of this product, contact:

North American Headquarters	International Headquarters
e-mail: support@RadView.com Phone: 1-888-RadView (1-888-723-8439) (Toll Free) 908-526-7756 Fax: 908-864-8099	e-mail: support@RadView.com Phone: +972-3-915-7060 Fax: +972-3-915-7683

Note: We encourage you to use e-mail for faster and better service.

When contacting technical support please include in your message the full name of the product, as well as the version and build number.

Chapter 2

Programming your JavaScript Agenda

WebLOAD functions on many different levels. WebLOAD IDE provides a wide range of testing tools and features, ranging from basic tools that are available through a simple graphic user interface, to features that require only a small amount of Agenda 'tweaking', to complex functionality that assumes a sophisticated understanding of programming techniques. Beginning users may take advantage of the basic WebLOAD IDE tool set without ever understanding exactly how these features are implemented. Most WebLOAD testing features and configuration settings are handled automatically by WebLOAD and never require any manual intervention. These GUI-only features are described in the *WebLOAD IDE User Guide*.

Understanding JavaScript Agendas

This chapter focuses on the options available to users who wish to customize their recorded Agendas and are comfortable looking at, and possibly giving a small 'tweak' to the JavaScript code within Agenda files. Major programming skills are not required for the tools in this chapter. A basic understanding of programming logic, an introduction to the internals of Agenda files, and careful reading of the examples provided are enough to get started using the tools documented here.

Note: The topics presented in this chapter include fragments of JavaScript Agenda code. To view the JavaScript code of an Agenda, select JavaScript View from the toolbar or drop-down View menu. This opens a JavaScript View pane that automatically displays the JavaScript code corresponding to any item highlighted in the Agenda Tree.

The WebLOAD IDE tools described in the rest of this manual involve some degree of intervention within the JavaScript code of an Agenda. Users will have an easier time working with these tools if they have a basic understanding of JavaScript Agendas. This chapter therefore begins with a description of the components, structure, and grammar of JavaScript Agendas.

What are JavaScript Agendas?

WebLOAD tests applications by running JavaScript Agendas that simulate the actions of real users. You don't have to be familiar with the JavaScript language to work with WebLOAD and WebLOAD IDE, and test applications. JavaScript Agendas are recorded through WebLOAD IDE. As you execute a typical sequence of activities, WebLOAD IDE records the HTTP protocol level traffic that is generated by the Web browser according to your accesses. By the conclusion of your recording session, a complete JavaScript Agenda file is created.

The basic 'Building Blocks' of a recording session are HTTP requests, which are triggered by user actions. Each time a user navigates to a new URL or submits a form, the browser emits an HTTP method and the resulting HTTP request is recorded. Externally, user activities (at the protocol level) are represented on the WebLOAD IDE desktop by a set of clear, intuitive icons and visual devices arranged in a visual Agenda Tree. Internally, WebLOAD IDE automatically creates JavaScript Agendas that act as scripts, recreating the HTTP traffic created by the actions of the original user during later test sessions.

Most users begin application testing by simply recording and then running a series of basic Agendas, without ever looking into an Agenda's internal Building Blocks to see the actual JavaScript code inside. As their understanding of the WebLOAD IDE tool set grows, many users decide to expand or tailor their original set of Agendas to meet a particular testing need, adding customized features that sometimes require some editing of the JavaScript code within the Agenda itself. WebLOAD IDE offers a wide range of features and options, for users who wish to add more functionality to their Agendas, rather than simply replaying the same set of HTTP methods exactly as originally recorded. Many of these tools are available through the WebLOAD IDE GUI and do not require any additional programming skills. These tools are documented in the *WebLOAD IDE User Guide*. The more complex tools that require some familiarity with JavaScript programming are described in the remaining chapters of this guide, with detailed syntax specifications provided in the *WebLOAD JavaScript Reference Guide*.

Test session Agendas are written in JavaScript. JavaScript is an object-oriented scripting language originally developed by Netscape Communications Corporation and is currently maintained by the Mozilla Foundation. JavaScript is best known for its use in conjunction with HTML to automate World Wide Web pages. However, JavaScript is actually a full-featured programming language that can be used for many purposes besides Web automation. WebLOAD and WebLOAD IDE have chosen JavaScript as the scripting language for test session Agendas. WebLOAD IDE JavaScript Agendas combine the ease and simplicity of WebLOAD's visual, intuitive programming environment with the flexibility and power of JavaScript object-oriented programming.

Agenda Tree Structure

Agenda Tree Nodes

JavaScript Agendas are represented on the WebLOAD IDE desktop by an Agenda Tree. Each item that appears in the Agenda Tree is an Agenda node. As you work in your application during a recording session, WebLOAD IDE adds nodes to the Agenda Tree. Each node in the Agenda Tree represents a single user action at the HTTP protocol level, such as submitting a form or navigating to a new URL. Agenda Tree nodes provide an intuitive, graphic representation of the underlying JavaScript code that actually implements the user activities to be recreated at run time.

The following figure illustrates a typical Agenda Tree fragment:
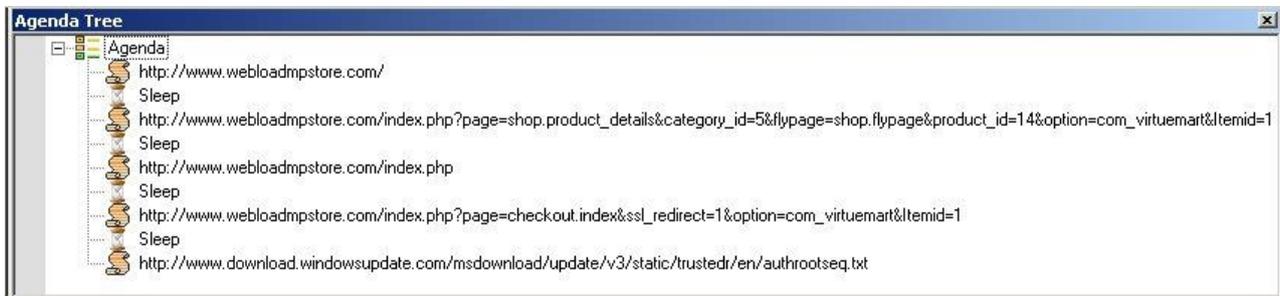
Figure 1: Typical Agenda Tree Fragment

Agenda Tree nodes are arranged sequentially.

The sequential arrangement of icons in the Agenda Tree means that icons appear in the Agenda Tree in the order in which the user actions and HTTP methods occurred when originally recorded. URL nodes in an Agenda Tree will appear in the order in which the browser requested each Web page over the course of a recording session. In cases where the browser emits asynchronous requests (for example, while using AJAX), the requests are recorded and appear in the Agenda Tree in the order that the browser emitted them.

Note: WebLOAD IDE offers users a wide range of tools and features that can add very powerful functionality to a testing Agenda. While users are certainly encouraged to incorporate these tools in their Agenda Tree, users must also be careful when editing URL nodes in the Agenda Tree. Because the order of HTTP activities within an Agenda is so significant, changing the sequence of URLs in the Agenda Tree in effect means changing the sequence of activities, and may destroy the functionality of the Agenda.

Agenda Program Structure

After a user has finished recording a typical session with the system under test, WebLOAD IDE saves a complete record of all user activities at the protocol level and converts the user activity information to a JavaScript Agenda. This Agenda can be run repeatedly, with a variety of testing configurations, until the user is satisfied that the system has been thoroughly tested. The Agenda created by the WebLOAD IDE JavaScript interpreter has the following underlying structure.

Main Script

The main script contains JavaScript code representing the user activities at the protocol level to be simulated during a test session. The main script is required. Without it, WebLOAD and WebLOAD IDE cannot run a test.

The main script is constructed and based on an exact recording of the specific sequence of user activities, such as the following, which are completed during a recording session:

- Web page navigations
- Form submissions
- User think time (sleep)

In addition to the original recording session activities, users may also add other features to the Agenda, such as messages or pause and sleep times. These features are usually added through WebLOAD IDE. WebLOAD executes the main script repeatedly, presenting the results in a series of analysis reports produced at the end of testing sessions.

Initialization and Termination Functions

These are JavaScript functions that WebLOAD or WebLOAD IDE executes once, in a fixed sequence, before or after the main script. These functions are used to prepare for or clean up from a testing session. The following functions are automatically included in the Agenda when needed:

- `InitAgenda()` — Initialize global objects shared by all WebLOAD IDE clients that run the Agenda.
- `InitClient()` — Initialize local objects and variables for individual clients.
- `TerminateClient()` — Free resources of individual clients.
- `TerminateAgenda()` — Free global resources shared by all clients running the Agenda.

- `OnScriptAbort()` — Executes user-defined code to free resources whenever the Agenda stops the execution of a round (ErrorMessage), a session (SevereErrorMessage), or in the event of an error. In addition, OnScriptAbort() is called in the Console whenever the script stops abruptly. This occurs when the end of a scheduled session is in the middle of a round, or when a user manually stops the session.

- `OnErrorTerminateClient()` — Clean up and free resources after a runtime error (per client).

- `OnErrorTerminateAgenda()` — Clean up and free resources after a runtime error (per Agenda).

The initialization and termination functions are not part of the WebLOAD performance test. WebLOAD does not include the operations of these functions in the performance statistics.

Use the initialization and termination functions to create or free objects or to set global variables. Besides these tasks, the functions may contain other JavaScript statements and may call other functions in your Agenda. The termination functions are used both when a test session finishes successfully, and when it is terminated early by an error. See *Agenda Execution Sequence* (on page 11), and *Non-Standard Agenda Execution Sequence* (on page 45), for more information.

Initialization and termination functions may be added directly to the code in an Agenda through the IntelliSense Editor, as described in *Editing the JavaScript Code in an Agenda* (on page 15).

To bring up a list of available functions:

- While working in Java Editing mode, select the node in the Agenda Tree to which you want to add a function, right-click the JavaScript pane, and select **Insert ➤ General ➤ Init/Terminate Functions** from the pop-up menu.

WebLOAD IDE automatically inserts the correct code for the selected function into the Agenda file. You may then add any other necessary commands to the functions without any concerns about mistakes in the function syntax.

Cut	Ctrl+X		
Copy	Ctrl+C		
Paste	Ctrl+V	General ▶	
		Init/Terminate Functions ▶	function InitAgenda() {[<Statement>]*}
Find...	Ctrl+F	Copy/Include Files ▶	function TerminateAgenda() {[<Statement>]*}
Replace...	Ctrl+H	Message Commands ▶	function OnErrorTerminateAgenda() {[<Statement>]*}
Go To...	Ctrl+G	Random Number Commands ▶	function InitClient() {[<Statement>]*}
Go To Object in Tree		Global Variables ▶	function TerminateClient() {[<Statement>]*}
Check Syntax		HTTP Commands ▶	function OnErrorTerminateClient() {[<Statement>]*}
Add WebLOAD IDE Block		HTTP Variables (wlGlobals) ▶	function OnScriptAbort() {[<Statement>]*}
Import JavaScript File...		HTTP Variables (wlHttp) ▶	
Clear JavaScript Editor		Transaction and Verification ▶	
Insert ▶		Dynamic HTML Variables (wlGlobals) ▶	
Insert Variable ▶		Dynamic HTML Variables (wlHttp) ▶	
		Dynamic HTML Functions ▶	
Toggle Breakpoint	F9	Dynamic Response Functions ▶	
Enable Outlining		Dynamic URL Functions ▶	
Line Numbers		SSL Commands (wlGlobals) ▶	
Outlining ▶		SSL Commands (wlHttp) ▶	
		SSL Cipher Functions ▶	
		Certificate Variables (wlGlobals) ▶	
		Certificate Variables (wlHttp) ▶	
		COM Objects ▶	
		Java Objects ▶	

Figure 2: Function Insertion using the Insert Menu

Navigation Functions

When a URL node is selected in the Agenda Tree, the corresponding Agenda code that appears in the JavaScript View pane will usually include basic navigation and validation functions such as `wlHTTP.Get()` and `wlHTTP.Post()`. The `wlHTTP.Get()` and `wlHTTP.Post()` methods store the HTML from the navigated page in the document.wlSource property. The code is refreshed when the Agenda calls `wlHTTP.Get()` or `wlHTTP.Post()` again. The stored code includes any scripts or other data embedded in the HTML, which the Agenda can retrieve and interpret in any desired way.

These functions are almost always visible at the start of the JavaScript code for every URL node in the Agenda Tree. While additional Tree nodes add their own additional corresponding JavaScript code to the Agenda, these functions form the basis for all test session Agendas.

Agenda Execution Sequence

Basic Execution Sequence

WebLOAD executes JavaScript Agendas in a simple, fixed sequence. The normal sequence is as follows.

1. The optional InitAgenda() initialization function runs once for each Load Generator, initializing the global WebLOAD objects, or any other global data or resources, that are shared by all clients in a Load Generator process.

2. The Agenda splits into a separate thread for each client. Load Generator processes may include any number of Virtual Clients, all running the same Agenda. After WebLOAD runs `InitAgenda()`, it runs each client in a separate thread. Each thread can have its own local variables and objects. In essence, each client runs an independent instance or copy of the Agenda.

   For example, suppose you want each WebLOAD client to connect to a different Web page. You can use a local object within each thread to store the Web address and perform the HTTP connection. Because the object is local, there is no confusion between the addresses. Each thread connects to the appropriate address, without any effect on the other threads of the same Agenda.

3. The optional `InitClient()` initialization function runs once for each thread, initializing local WebLOAD objects, or any other variables or resources, which belong to an individual thread of the Agenda. In this way, each thread of an Agenda can work with different data and can operate independently of the other threads.

4. The main script runs repeatedly in a separate loop for each thread. When WebLOAD reaches the end of the main script, it starts again at the beginning. The main script continues to iterate until you stop the WebLOAD test. You can stop the test by issuing a Stop command in the WebLOAD Console, or you can tell the Console to stop the test automatically after a predefined time.

   WebLOAD collects performance statistics while the main script runs. The various statistics that WebLOAD displays (round time, average round time, etc.) apply only to the main script, not to operations in the initialization and termination functions.

5. The optional `TerminateClient()` termination function runs once for each thread. Strictly speaking, most Agendas do not need a `TerminateClient()` function, because JavaScript automatically frees objects and releases most resources when the Agenda terminates. In a complex Agenda, however, it is good programming practice to free local objects and resources explicitly using `TerminateClient()`.

6. The separate threads for each client terminate.

7. The optional `TerminateAgenda()` termination function runs once for each Load Generator process. Most simple Agendas do not need a `TerminateAgenda()` function. In a complex Agenda, including `TerminateAgenda()` is recommended to free global resources.

8. The `OnScriptAbort()`, `OnErrorTerminateClient()`, and `OnErrorTerminateAgenda()` functions clean up and free resources after a runtime error. For more information about handling errors during a test session, see *Error Management* (on page 41).

The following figure illustrates the steps in a normal Agenda execution sequence:
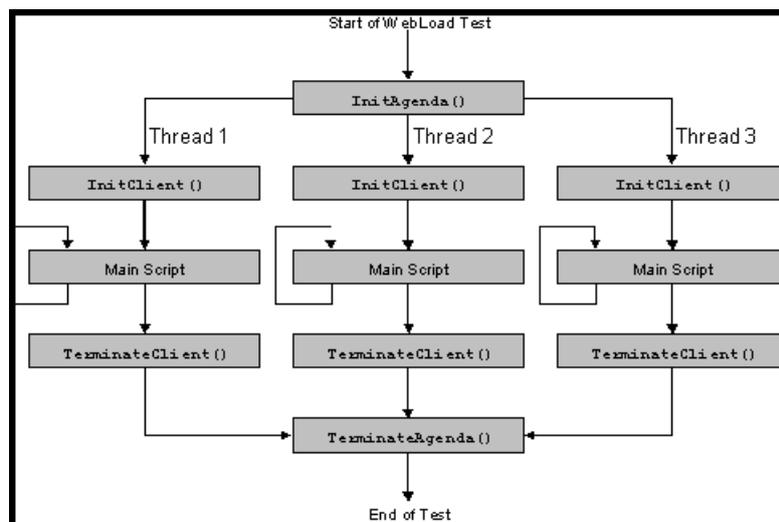
Figure 3: Steps in a Typical Agenda Execution Sequence

EvaluateScript Function

The `EvaluateScript()` function is used to define JavaScript code to be executed at specific points during the execution of the Agenda. The function allows testers to include scripts from an external library and specify the point during Agenda execution at which the script should be executed. You can specify to execute the script at any of the following points within the Agenda:

- WLBeforeInitClient
- WLAfterInitAgenda
- WLBeforeThreadActivation
- WLOnThreadActivation
- WLAfterTerminateClient
- WLBeforeRound
- WLAfterTerminateAgenda

- WLAfterRound

If the script run is successful, the value from the last executed expression statement processed in the script is saved and then parsed when the engine reaches the relevant point. The saved value includes the following information:

- The text of the script.

- The size of the script text, in bytes.

- The name of the file or URL containing the script text.

If the script run is unsuccessful, the value is left undefined.

The following example is used to call a function that is defined in an external file:

```
IncludeFile(filename.js)
EvaluateScript("MyFunction()",WLAfterRound)
```

In this example, the `EvaluateScript` string is parsed, executed, and instructs the JavaScript engine to call the `MyFunction()` function right after each round is completed. At this point WebLOAD returns any syntax or runtime errors.

For more information on the EvaluateScript function, see the *WebLOAD's JavaScript Reference Guide*.

Cleanup at the End of Each Round

The main script retains its variables and objects from one round (iteration) to the next. For example, you can increment a variable in each round, or you can store data in an object in one round and access the data in the next.

There are three exceptions to this rule. At the end of each round:

- Any cookies that the script sets are deleted.
- The SSL Cache is cleared.
- Any open HTTP connections are closed.

Execution Sequence for Scheduled Clients

WebLOAD lets you schedule the number of clients running a single Agenda in a single Load Generator. WebLOAD runs the initialization and termination functions of the Agenda according to your schedule.

For example, suppose you configure a Load Generator to run an Agenda according to the following schedule:

Table 3: Load Generator Schedule—Sample 1

Number of Threads	Scheduled Time
50 threads	For the first 30 minutes.
20 threads	From 30 to 60 minutes.
100 threads	For the remainder of the test.

In this case:

- WebLOAD runs the `InitAgenda()` function when the Load Generator starts to run. WebLOAD then starts the first 50 threads, running `InitClient()` and the main script for each thread.

- After 30 minutes, 30 of the threads stop. WebLOAD runs `TerminateClient()` for these 30 threads. The main script of the other 20 threads continues running.

- At 60 minutes, 80 additional threads start. For each of these 80 threads, WebLOAD runs `InitClient()` and the main script.

- At the end of the test, WebLOAD runs `TerminateClient()` for each of the 100 threads currently running, and then runs `TerminateAgenda()`.

Execution Sequence for Mixed Clients

You can configure a single Load Generator to run more than one Agenda. In that case, WebLOAD splits the time on each thread between the Agendas. WebLOAD runs the initialization and termination functions of each Agenda at the beginning and end of the threads, respectively. It does not run the functions each time the threads switch between Agendas.

For example, suppose you configure a Load Generator to run 100 threads as follows:

Table 4: Load Generator Schedule—Sample 2

Percentage of time	Selected Agenda
50% of the time	`agenda1. wlp`
50% of the time	`agenda2. wlp`

In this case, WebLOAD runs:

- The `InitAgenda()` functions of both Agendas when the Load Generator starts.
- The `InitClient()` functions of both Agendas in each thread when the thread starts.
- The `TerminateClient()` functions of both Agendas, in each thread at the end of the test.
- The `TerminateAgenda()` functions of both Agendas when all the threads have stopped.

Editing the JavaScript Code in an Agenda

WebLOAD is a fully automated testing tool. WebLOAD users are able to record Agendas, add testing tools, run test sessions, and analyze the results, without any programming skills whatsoever. Nevertheless, there are users who wish to manually edit the JavaScript code of a recorded Agenda to add functionality to test session Agendas created with WebLOAD IDE, creating more complex, sophisticated test sessions.

For example, many users design test sessions around a set of basic Agendas created through WebLOAD IDE and then expand or tailor those Agendas to meet a particular testing need. Some of the reasons for editing JavaScript Agendas include:

- Recycling and updating a useful library of test Agendas from earlier versions of WebLOAD.
- Testing websites that work with Java or COM components.
- Creating advanced, specialized verification functions.
- Adding global variables and messages.
- Debugging the system under test.
- Optimization capabilities, to maximize your application's functionality at minimal cost.

This section describes the tools provided by WebLOAD IDE to help you access and edit the JavaScript code within your Agenda.

Accessing the JavaScript Code within the Agenda Tree

WebLOAD IDE provides a complete graphic user interface for creating and editing Agenda files. Additions or changes to an Agenda are usually made through the WebLOAD IDE GUI, working with intuitive icons representing user actions in a graphic Agenda Tree. For greater clarity, the JavaScript code that corresponds to each user action in an Agenda is also visible in the JavaScript View pane on the WebLOAD IDE desktop.

While most people never really work with the JavaScript code within their Agenda, some users do wish to manually edit the JavaScript code underlying their Agenda Tree. For example, some test sessions may involve advanced WebLOAD testing features that can not be completely implemented though the GUI, such as Java or XML objects. Editing the JavaScript code in an Agenda does not necessarily mean editing a huge JavaScript file. Most of the time users only wish to add or edit a specific feature or a small section of the code. WebLOAD IDE provides access to the JavaScript code in an Agenda through JavaScript Object nodes, which are seen on the following levels:

- **JavaScript Object nodes**—individual nodes in the Agenda Tree. Empty JavaScript Object nodes may be dragged from the WebLOAD IDE toolbar and dropped onto the Agenda Tree at any point selected by the user, as described in *Adding JavaScript Object Nodes* (on page 19). Use the IntelliSense Editor, described in *Using the IntelliSense JavaScript Editor* (on page 16), to add lines of code or functions to the JavaScript Object.

- **Imported JavaScript File**—an external JavaScript file that should be incorporated within the body of the current Agenda.

  - While working in JavaScript Editing mode, right-click the JavaScript pane and select **Import JavaScript File** from the WebLOAD IDE menu.

    Often, testers work with a library of pre-existing library files from which they may choose functions that are relevant to the current test session. This modular approach to programming simplifies and speeds up the testing process, and is fully supported and endorsed by WebLOAD.

Using the IntelliSense JavaScript Editor

For those users who wish to manually edit their Agendas, WebLOAD IDE provides three levels of programming assistance:

- An IntelliSense Editor mode for the JavaScript View pane.

  Add new lines of code to your Agenda or edit existing JavaScript functions through the IntelliSense Editor mode of the JavaScript View pane. The IntelliSense Editor helps programmers write the JavaScript code for a new function by formatting new code and prompting with suggestions and descriptions of appropriate code choices and syntax as programs are being written. For example, in the following figure the IntelliSense Editor displays a drop-down list of available properties and objects for the `wlHttp` object being added to the program, with a pop-up box describing the highlighted method in the list.
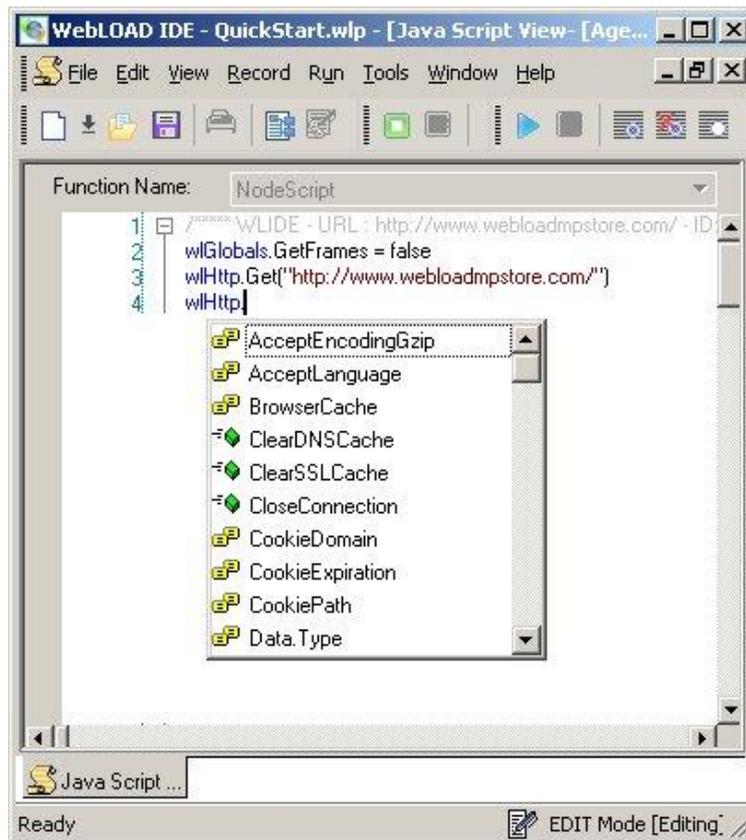
Figure 4: wlHttp Drop-Down List of Available Properties and Objects in IntelliSense Editor

- A selection of the most commonly used programming constructs, can be accessed by right-clicking in the JavaScript Editing Pane, and selecting **Insert ➤ Java Objects** from the pop-up menu.

  Users who choose to program their own JavaScript Object code within their Agenda may take advantage of the WebLOAD IDE GUI to simplify their programming efforts. Manually typing out the code for each command, risks making a mistake, even a trivial typo, and adding invalid code to the Agenda file. Instead, users may bring up a list of available commands and functions for a specific item, by right-clicking in the JavaScript Editing Pane after selecting a specific node from the Agenda Tree, selecting **Insert** from the pop-up menu, and selecting an item illustrated in the following figure to display the item's available commands. WebLOAD IDE automatically inserts the correct code for the selected item into the JavaScript Object currently being edited. The user may then change specific parameter values without any worries about accidental mistakes in the function syntax.
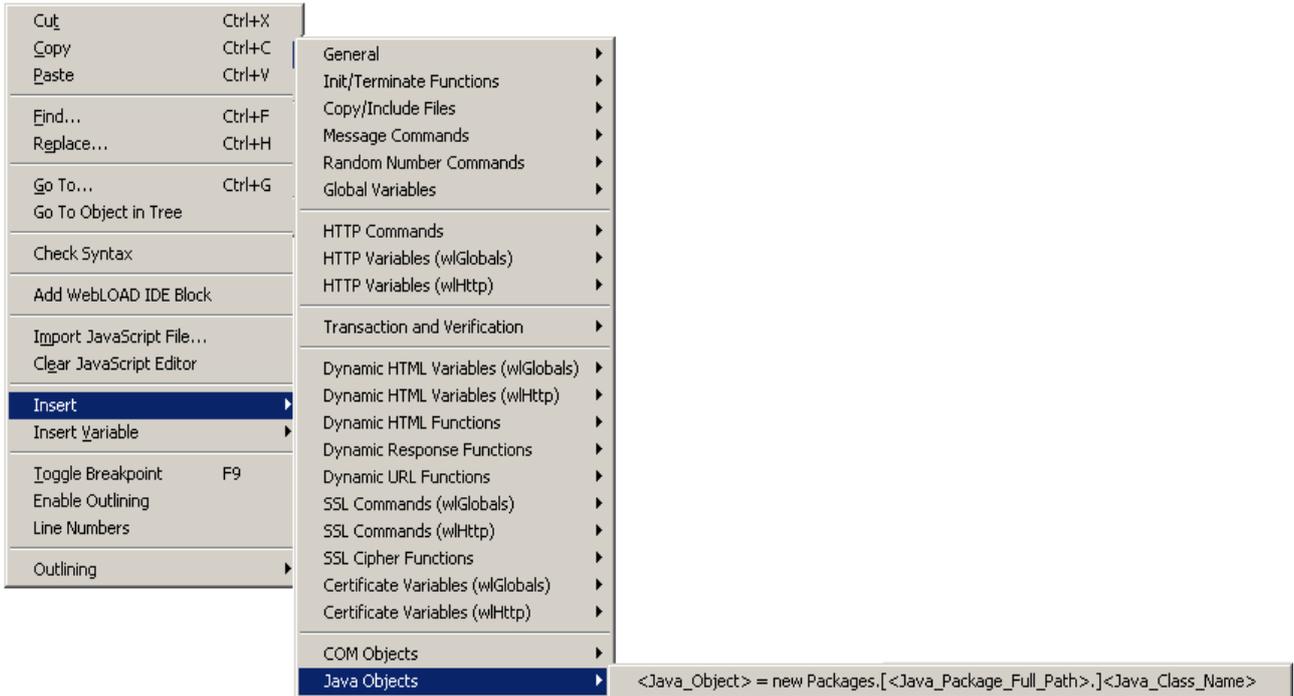
Figure 5: Insert Java Options Menu

In addition to the Insert menu, you may select an item from the Insert Variable menu, to add system and user-defined parameters to the Agenda. This eliminates the need for manual coding.
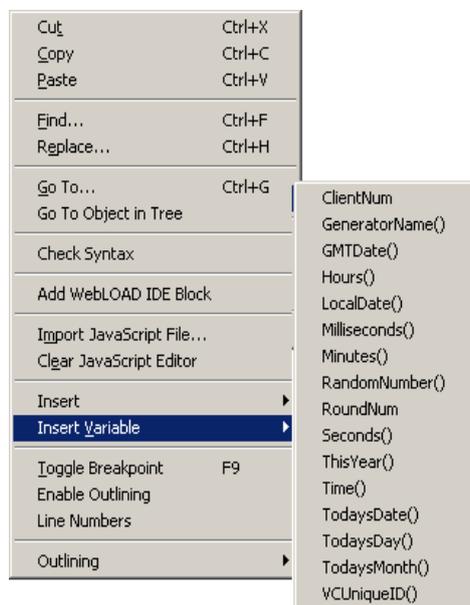
Figure 6: Insert Variable Menu

- A Syntax Checker that checks the syntax of the code in your Agenda file and catches simple syntax errors before you spend any time running a test session.

Select **Tools ➤ CheckSyntax** while standing in the JavaScript View pane of the WebLOAD IDE desktop to check the syntax of the code in your Agenda file.

Agenda code that you wish to write or edit must be part of a JavaScript Object in the Agenda Tree. Adding or converting JavaScript Objects in an Agenda Tree is described in *Accessing the JavaScript Code within the Agenda Tree* (on page 15).

Notes: If you do decide to edit the JavaScript code in an Agenda, be careful not to damage the Agenda structure by changing the sequence or integrity of the Agenda navigation blocks. A test session Agenda is constructed and based on a specific sequence of user activities at the protocol level, such as URL navigations, form submissions, and others. Changing the sequence of code blocks in effect means changing the sequence of user activities, and may destroy the functionality of the test session Agenda. In addition, you should be careful not to change the automatically generated WebLOAD IDE comments, since necessary information for running the Agenda may become lost.

If you use an external text editor to modify the JavaScript code in a JavaScript Agenda that was created through WebLOAD IDE, the changes you made through the external editor will be lost if you open the Agenda in WebLOAD IDE again. Therefore, be sure to do all JavaScript code editing through the IntelliSense Editor in WebLOAD IDE only.

For detailed information about the JavaScript language, please refer to the *WebLOAD JavaScript Reference Guide*. This guide is supplied in Adobe Acrobat format with the WebLOAD software. You may also learn the elements of JavaScript programming from many books on Web publishing. Keep in mind that some specific JavaScript objects relating to Web publishing do not exist in the WebLOAD test environment.

Adding JavaScript Object Nodes

WebLOAD IDE stores user activities at the protocol level as they are completed during recording sessions. These activities are later recreated during subsequent testing sessions. However, the system being tested may also involve user activities that are significant to the system testers, yet cannot be entirely recorded through WebLOAD IDE. Or testers may wish to add additional functionality to their tests that require inserting extra JavaScript functions into the body of their Agenda code. WebLOAD IDE therefore provides a mechanism for including such items within a test session Agenda.

For example, some of the steps involved in access to an external database are executed outside the control of WebLOAD IDE and therefore cannot be completely recorded through WebLOAD IDE alone. Users may add these activities to a test Agenda by adding empty JavaScript Object Nodes to the Agenda Tree and then filling in the code to complete the desired activity. The specific code needed within the JavaScript Object

Node will obviously vary from Node to Node depending on the activity, but adding the generic JavaScript Object Nodes themselves is a standard WebLOAD IDE feature.

To add JavaScript Object Nodes to your test Agenda directly through the WebLOAD IDE GUI:

- Drag the **JavaScript Object** icon from the toolbox and drop it into the Agenda Tree at the preferred spot, as described in the *WebLOAD IDE User Guide*.

  The following figure illustrates a typical JavaScript Object Node highlighted in the Agenda Tree.
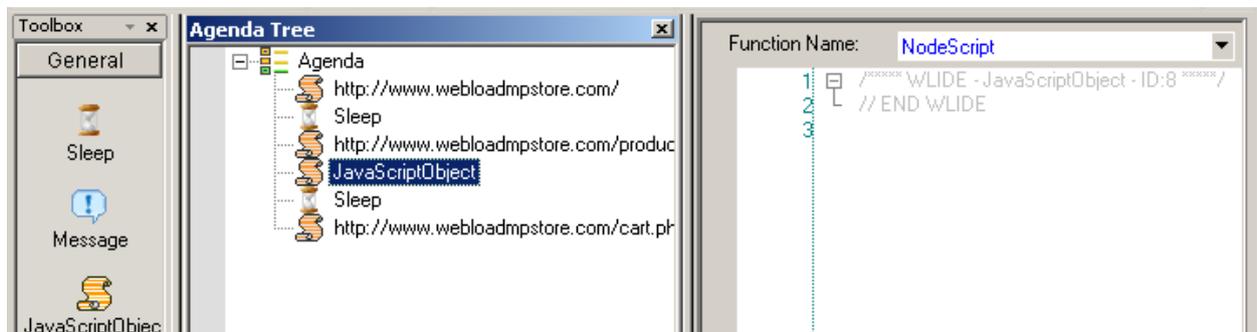
Figure 7: Typical JavaScript Object Node

The specific code needed within the JavaScript Object Node will obviously vary from Node to Node depending on the activity. Some of the activities that may require manual programming within a JavaScript Object Node include:

- Getting data from a database.

- Working with COM, XML, or Java objects.

- Calculating expressions.

- Working with global variables such as a user-defined value to include in messages.

- Working with other protocols.

- Doing specific custom made verifications.

JavaScript syntax specifications for some of these activities are documented in the *WebLOAD JavaScript Reference Guide*.

To add or edit lines of code in a JavaScript Object:

- Click the **JavaScript Object node** in the Agenda Tree.

  The JavaScript object's code appears in the JavaScript View pane. You can edit the code directly in the JavaScript view or by selecting **Edit ➤ Start Java Script Editing** which opens the full JavaScript view screen. This helps programmers write the JavaScript code for a new function by formatting new code and prompting with

suggestions and descriptions of appropriate code choices and syntax as programs are written, as described in *Using the IntelliSense JavaScript Editor* (on page 16).

File Management

JavaScript Agendas work with many different types of files. Users may wish to add to their Agendas programming elements that include the following file types:

- *Including Files* (on page 21)

- *Copying Files* (on page 25)

- *Output Files* (on page 27)

- *Input Files* (on page 30)

Including Files

WebLOAD IDE allows you to include external files within your JavaScript program. This facilitates modular programming, where you may develop different recyclable modules of JavaScript source code to be reused by different test Agendas. Rather than inserting the complete original source code text over and over again into the body of each Agenda, use the `IncludeFile()` command to make all functions defined within the included file available to all including Agendas.

Adding an IncludeFile() Function

`IncludeFile()` functions can be added directly to a JavaScript Object in an Agenda through the IntelliSense Editor, as described in *Editing the JavaScript Code in an Agenda* (on page 15).

To insert an IncludeFile function:

- While working in JavaScript Editing mode, right-click the JavaScript Pane and select **Insert ➤ JavaScript ➤ Copy/Include Files** from the pop-up menu that appears. Select the `IncludeFile()` function from the sub-menu.

WebLOAD IDE automatically inserts the correct code for the selected function into the Agenda file. The user may then edit parameter values without any worries about mistakes in the function syntax.
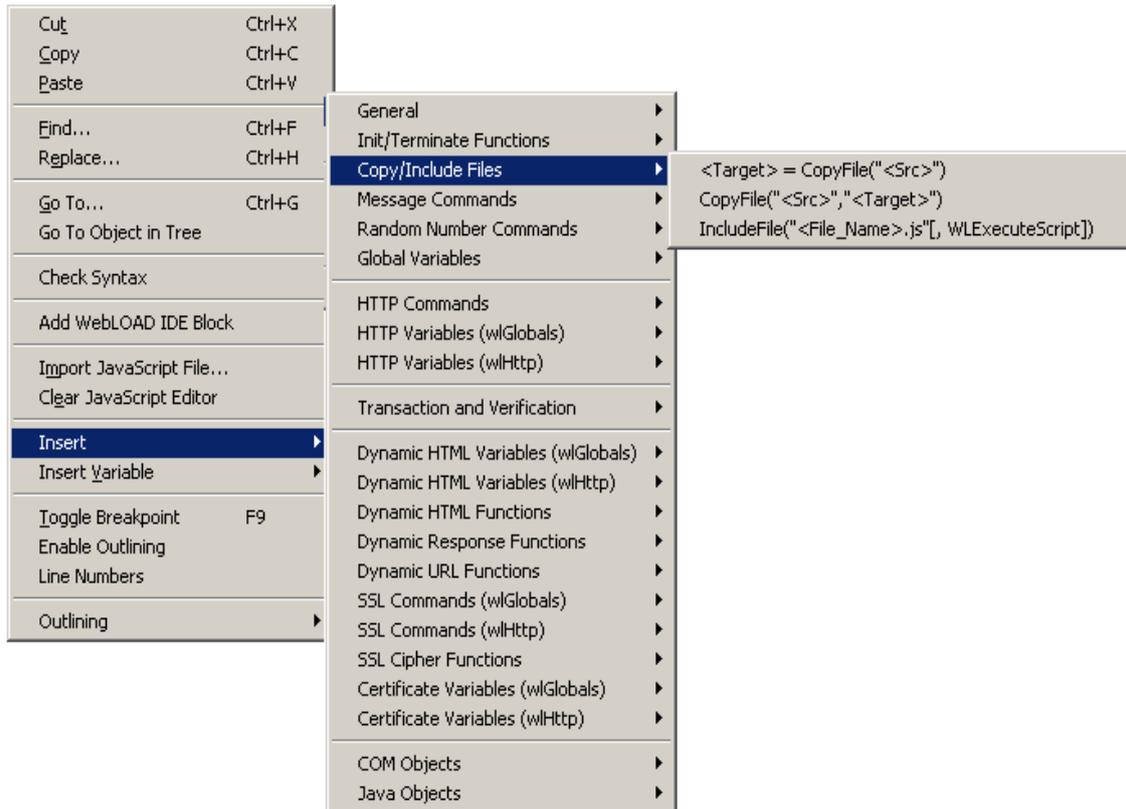
Figure 8: IncludeFile() Function Insertion using Copy/Include Files Menu

For example, to include the external file `MyFunction.js`, located on the Console during WebLOAD testing, your Agenda file should begin with the following command included in a JavaScript Object node:

```
function InitAgenda() {
   ...
   IncludeFile("MyFunction.js")
}
```

See the *WebLOAD JavaScript Reference Guide* for a complete syntax specification.

Determining the Included File Location

WebLOAD assumes that the source file is located in the default directory specified in the File Locations tab in the **Tools ➤ Global Options** dialog box in the WebLOAD Console desktop, illustrated in the following figure:
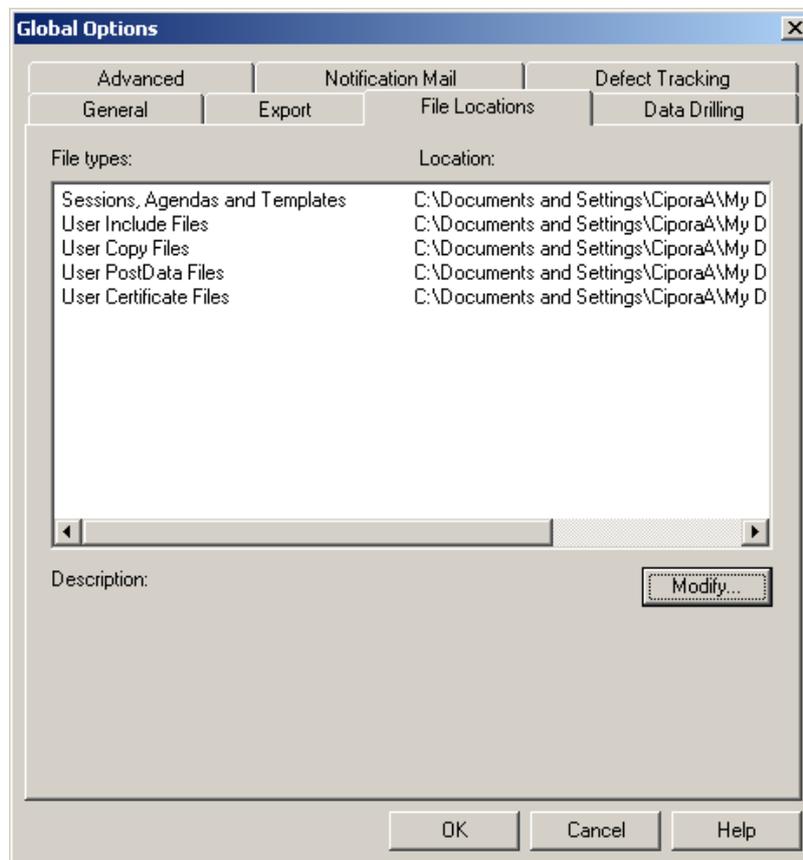
Figure 9: File Locations Tab

Included files by default should be in the User Include Files directory specified in the File Locations dialog box. You may reset the default directory location if necessary by selecting the User Include Files entry and clicking **Modify**. You can select a new file location from the pop-up Browse for Folder dialog box and click **OK**. In general, the system searches for the included file using the following search precedence:

- The load engine first looks for the included file in the default User Include Files directory. If the file is not there, the file request is handed over to WebLOAD, which searches for the file using the following search path order:

  a. If a full path name has been hardcoded into the IncludeFile command, the system searches the specified location. If the file is not found in an explicitly coded directory, the system returns an error code of `File Not Found` and will not search in any other locations.

Note: It is not recommended to hardcode a full path name, since the Agenda will then not be portable between different systems. This is especially important for networks that use both UNIX and Windows systems.

b. Assuming no hardcoded full path name in the Agenda code, the system looks for the file in the default User Include Files directory.

c. If the file is not found, the system looks for the file in the current working directory, the directory from which WebLOAD or WebLOAD IDE was originally executed.

d. Finally, if the file is still not found, the system searches for the file sequentially through all the directories listed in the File Locations tab.

Example: Working with an Included Function

Included files are files whose functions and other contents are included and accessible from the current Agenda file without the file text actually appearing within the Agenda. Use included files to take advantage of an external library of modular functions without adding extra lines of code to your Agenda, improving the clarity of your Agenda code while saving both Agenda space and development time. For example, in the following Agenda fragment, the external file `myjs.js` is included within the `InitAgenda()` initialization function. One of the functions within that included file, `sayHello()`, is later accessed within the main body of the Agenda. The text of the `sayHello()` is embedded in a nearby comment for easy reference.

```
function InitAgenda()
{
   IncludeFile("C:\\Documents and Settings\\uriw\\My
   Documents\\myjs.js")
}
wlGlobals.GetFrames = false
wlHttp.Get("http://www.webloadmpstore.com/general_sample/include
/form.php")


Sleep(4765)


wlHttp.Header["Referer"] = "http://qa50/form.php"
wlHttp.ContentType = "application/x-www-form-urlencoded"
wlHttp.FormData["item"] = "Brushes"
wlHttp.FormData["quantity"] = "55"
var item = wlHttp.FormData["quantity"]
wlHttp.Post("http://www.webloadmpstore.com/general_sample/includ
e/process.php")
// here the outside function is being called
sayHello()
```

The `myjs.js` file contains the following `sayHello()` function, which prints the checkout message to the customer:

```
function sayHello()
{
   InfoMessage("you ordered  " +item + " items")
}
```

Copying Files

WebLOAD enables text and binary data file copying from the Console to a Load Generator automatically during a test session. This is an important feature if your test uses an Agenda that references auxiliary files, such as I/O files, that are found only on the Console. The WebLOAD send option in the Console only sends the Agenda itself and will not send auxiliary files together with an Agenda file to a Load Generator. Use the `CopyFile` command instead to copy any necessary files from a source file on the Console to a destination file on the Load Generator before the main body of the Agenda is executed.

Adding a CopyFile() Function

`CopyFile()` functions can be added directly to a JavaScript Object in an Agenda through the IntelliSense Editor, as described in *Editing the JavaScript Code in an Agenda* (on page 14).

To insert a CopyFile function:

- While working in JavaScript Editing mode, right-click the JavaScript Pane and select **Insert ➤ Copy/Include Files** from the pop-up menu that appears.

    Select the preferred `CopyFile()` function from the sub-menu (see Figure 8).

WebLOAD IDE automatically inserts the correct code for the selected function into the Agenda file. The user may then edit parameter values without any worries about mistakes in the function syntax.

For example, to copy the auxiliary file `src.txt`, located on the Console, to the destination file `dest.txt` on the current Load Generator, your Agenda file should include the following command:

```
function InitAgenda() {
   ...
   CopyFile("src.txt", "dest.txt")
   ...
}
```

You may then access the file as usual in the main body of the Agenda. For example:

```
DataArr = GetLine("dest.txt")
```

See the *JavaScript Reference Guide* for a complete syntax specification.

Determining the Copied File Location

WebLOAD assumes that the source file is located on the Console in the default directory identified in the **File Locations** tab of the **Tools ➤ Global Options** dialog box in the Console desktop, illustrated in the following figure:
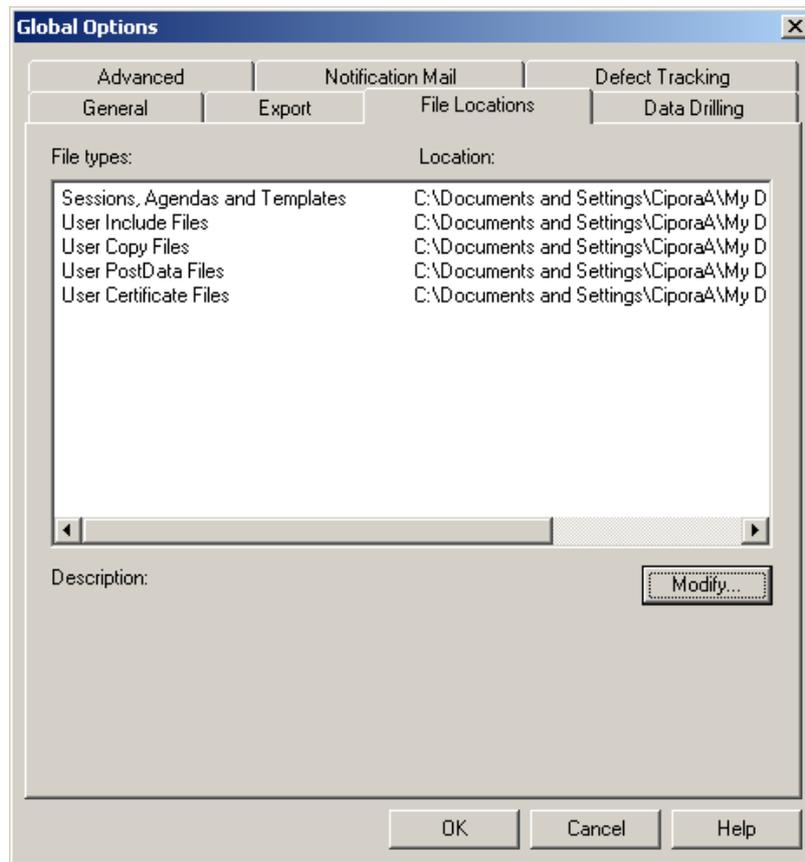
Figure 10: File Locations Dialog Box

Files to be copied by default should be in the User Copy Files directory specified in the File Locations dialog box. You may reset the default directory location if necessary by selecting the User Copy Files entry and clicking **Modify**. You can select a new file location from the pop-up Browse for Folder dialog box and click **OK**. Remember that WebLOAD does not create new directories, so any directories specified as source or target directories must already exist. In general the system searches for the file using the following search precedence:

- The load engine first looks for the file to be copied in the default User Copy Files directory. If the file is not there, the file request is handed over to WebLOAD, which searches for the file using the following search path order:

  a. If a full path name has been hardcoded into the CopyFile command, the system searches the specified location. If the file is not found in an explicitly coded directory, the system returns an error code of File Not Found and will not search in any other locations.

Note: It is not recommended to hardcode a full path name, since the Agenda will then not be portable between different systems. This is especially important for networks that use both UNIX and Windows systems.

  b. Assuming no hardcoded full path name in the Agenda code, the system looks for the file in the current working directory, the directory from which WebLOAD was originally executed.

  c. Finally, if the file is still not found, the system searches for the file sequentially through all the directories listed in the File Locations tab.

Output Files

Writing Agenda Output Messages to a File

Realistic testing requires multiple testing passes using a variety of realistic scenarios. The resulting output data should be stored in output files for later access, verification, study, and analysis. WebLOAD IDE provides the `wlOutputFile` object to simplify saving Agenda output. See the *WebLOAD JavaScript Reference Guide* for a complete syntax specification. This section will provide a brief introduction to the `wlOutputFile` object.

The `wlOutputFile` object lets you write Agenda output messages to an output file. Declaring a new `wlOutputFile` object creates a new, empty output file. If a file of that name already exists, the file will be completely overwritten. Information will not be appended to the end of an existing file. Be sure to choose a unique filename for the new output file if you do not want to overwrite previous Agenda data.

Adding the wlOutputFile Object

The `wlOutputFile` object can be added directly to a JavaScript Object in an Agenda through the IntelliSense Editor, as described in *Editing the JavaScript Code in an Agenda* (on page 14). Users who are programming their own JavaScript Object code within their Agenda may take advantage of the WebLOAD IDE GUI to simplify their programming efforts. Manually typing out the code to create a `wlOutputFile` object, risks making a mistake, and adding invalid code to the Agenda file. Instead, users may bring up a list of available constructor and methods for the `wlOutputFile` object, by right-clicking in the JavaScript Editing Pane, selecting **Insert ➤ General** from the pop-

up menu, and selecting one of the available items. WebLOAD IDE automatically inserts the correct code for the selected command into the JavaScript Object currently being edited. The user may then change the parameters without any worries about mistakes in the object syntax.
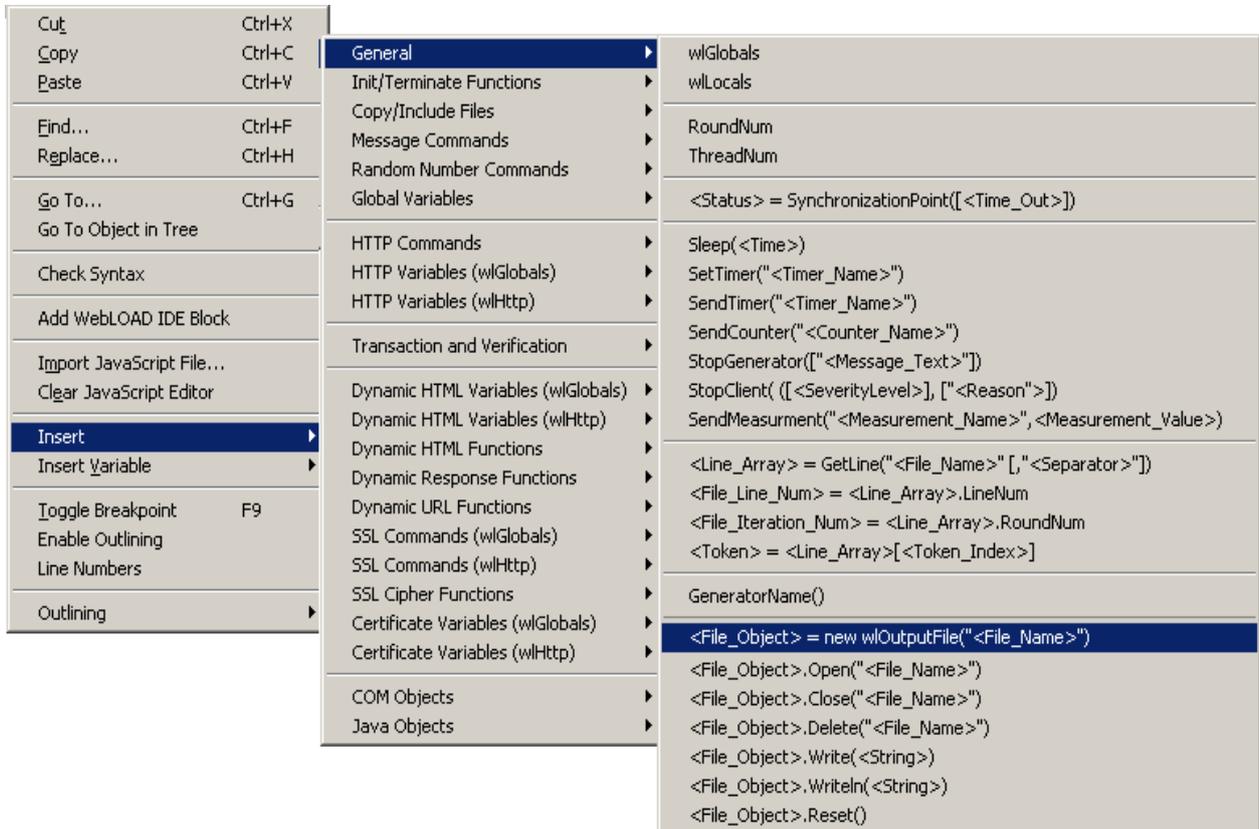
Figure 11: wlOutputFile() Insertion using General Menu

The preceding figure highlights the standard JavaScript syntax to create and work with `wlOutputFile` objects.

For example:

```
MyFileObj = new wlOutputFile("filename")
...
MyFileObj.Write("Happy Birthday")
...
delete MyFileObj
```

Note: The `wlOutputFile` object saves Agenda output messages. To save server response data, use the `Outfile` property described in *Saving Server Output to a File* (on page 29).

wlOutputFile Object Scope Limitations

If you declare a new `wlOutputFile` object in the `InitAgenda()` function of an Agenda, the output file will be shared by all the Agenda threads. There is no way to specify a specific thread writing sequence—each thread will write to the output file in real time as it reaches that line in the Agenda execution.

If you declare a new `wlOutputFile` object in the `InitClient()` function or main body of an Agenda, use the thread number variable as part of the new filename to be sure that each thread will create a unique output file.

If you declare a new `wlOutputFile` object in the main body of an Agenda, and then run your Agenda for multiple iterations, use the `RoundNum` variable as part of the new filename to be sure that each new round will create a unique output file.

Generally, you should only create new `wlOutputFile` objects in the `InitAgenda()` or `InitClient()` functions of an Agenda, not in the main script. If a statement in the main script creates an object, *a new object is created each time the statement is executed*. If WebLOAD repeats the main script many times, a large number of objects may be created and the system may run out of memory.

Saving Server Output to a File

It is often important to save server response data for later study and analysis. WebLOAD and WebLOAD IDE provide the `Outfile` property for this purpose. `Outfile` is a property of the `wlGlobals`, `wlLocals`, and `wlHttp` objects. See the *WebLOAD JavaScript Reference Guide*, for a complete syntax specification.

The following Agenda shows a typical entry to a server output file. The `ClientNum` and `RoundNum` global variables are used to distinguish between Agenda instances in the server output file entries.

```
// Main script actions go here
...
// Write the downloaded server output to a unique file
// for each Agenda instance, identified by ClientNum
// and RoundNum. The output file is named
// <fullpath>\SessionOutput.<ClientNum.RoundNum>.html
wlHttp.Outfile = "c:\\webld\\SessionOutput."
ClientNum + "." + RoundNum + ".html"
```

The `Outfile` property can be added directly to a JavaScript Object in an Agenda through the IntelliSense Editor, as described in *Editing the JavaScript Code in an Agenda* (on page 14).

Input Files

To test an application on a realistic scale, you need large numbers of Virtual Clients using a wide range of realistic input data. Realistic testing also requires multiple testing passes using a variety of realistic scenarios. Working with such extensive input data is only feasible using input files. Input files are a useful tool in application testing, providing all the test input needed without requiring a great deal of manual intervention. Input files may contain a wide range of realistic data, including erroneous data lines that have been deliberately added, such as an invalid date for a date field. This is a useful technique to test server performance when users submit invalid data.

WebLOAD IDE provides the GlobalInputFile Building Block to simplify the use of input files. WebLOAD IDE enables you to insert a file that contains fields which will be used as global variables throughout the entire script. The GlobalInputFile Building Block is the recommended method for managing input data files, as it is a user-friendly GUI approach that does not require any programming skills on the part of the user. However, users who prefer to edit the JavaScript code within their Agenda files may choose to work directly with the internal input file function set, described in this section. For more information on the GlobalInputFile Building Block, refer to *GlobalInputFile* in the *WebLOAD IDE User Guide*.

Working with ASCII Input Files

Input files are text-delimited files containing data that users might submit on a form. WebLOAD uses the `GetLine()` function to read input file data. The input files themselves must already exist in order to be used during a test session. WebLOAD and WebLOAD IDE will not create input files for you.

Input files contain data tokens, text strings that correspond to the input fields on a form. Each input file line should contain the same number of tokens, separated by a delimiter and terminated by newline and linefeed characters. For example, each line of the following file contains three tokens:

* First name

* Last name

* Email address

The delimiter in this file is the default comma (,) character.

```
FirstName,LastName,email@address.com
John,Smith,jsmith@ABC.com
Mary,Todd,mtodd@GHI.com
Betty Sue, ,bettysue@MNO.edu
Tom,Jones,invalid*email
...
```

A token may contain any characters or spaces except the delimiter character. Thus the first two tokens in the fourth line of the example are "Betty Sue" and " " (a string containing a single space character).

Note: You may also deliberately include erroneous data lines in your input file, such as an invalid date for a date field.

Understanding the GetLine Function

The GetLine() function reads and parses data from an ASCII file. The lines of data are stored in a LineArray object. The LineArray object contains both the actual file data and information about the file data, such as the line number on which that data token was found and the round number during which that data token was read from the file.

The GetLine() function reads an input file one line at a time, either sequentially or in a random order:

- If you choose the WLSequential (default) mode, then:
  - The first GetLine() call in the Agenda reads the first line of the file.
  - Each successive call in the Agenda reads the next line of the file.
  - When the last line of the file has been read, the next access loops back to the first line of the file.
- If you choose the WLRandom mode, then:
  - Each successive call in any thread of the Load Generator reads some randomly chosen line of the file.
  - You must include Open(filename, WLRandom) in the Agenda's InitAgenda() function.

You can specify if you want to read a random unique, unused row/value from the file. If another Virtual Client is using this value/row, the Virtual Client is not able to access this value/row and will read another random value/row.

In *Unique* mode, there is a significant performance difference between Sequential and Random. Sequential causes some performance degradation and heavier network communication. Therefore, if working in Unique mode, you should select Random unless the lines **must** be in sequential order. If working in *non-Unique* mode, Sequential is the preferable method in terms of performance, since Random uses more memory.

In this way, a relatively small file can supply an unending stream of test data, and different clients are supplied with different sequences of data.

For example, the following lines read the input data lines stored in MyFavoriteInputFile in a random order. The data tokens on each line are

separated by semicolons (;). The first data token found in the data file is then assigned to a variable in the program.

```
InitAgenda(){
  wlGlobals.TokenNumber = 0
  ...Open(filename, WLRandom)
}
...
MyInputDataArray =
  GetLine("MyFavoriteInputFile", ";")
...
NextDataToken = MyInputDataArray[wlGlobals.TokenNumber++]
```

See the *WebLOAD JavaScript Reference Guide*, for a complete syntax specification.

Adding Input File Commands to your Agenda Code

Input file command lines can be added directly to a JavaScript Object in an Agenda through the IntelliSense Editor, as described in *Editing the JavaScript Code in an Agenda* (on page 14). Users who are programming their own JavaScript Object code within their Agenda may take advantage of the WebLOAD IDE to simplify their programming efforts.

Manually typing out the code for a `GetLine()` function for example, risks making a mistake, and adding invalid code to the Agenda file. Instead, users may bring up a list of input file functions for the `wlOutputFile` object, by right-clicking in the JavaScript Editing Pane, selecting **Insert ➤ General** from the pop-up menu, and selecting one of the available items. WebLOAD IDE automatically inserts the correct code for the selected function into the JavaScript Object currently being edited. The user may then change the specific object names or parameter values without worrying about mistakes in the function syntax.
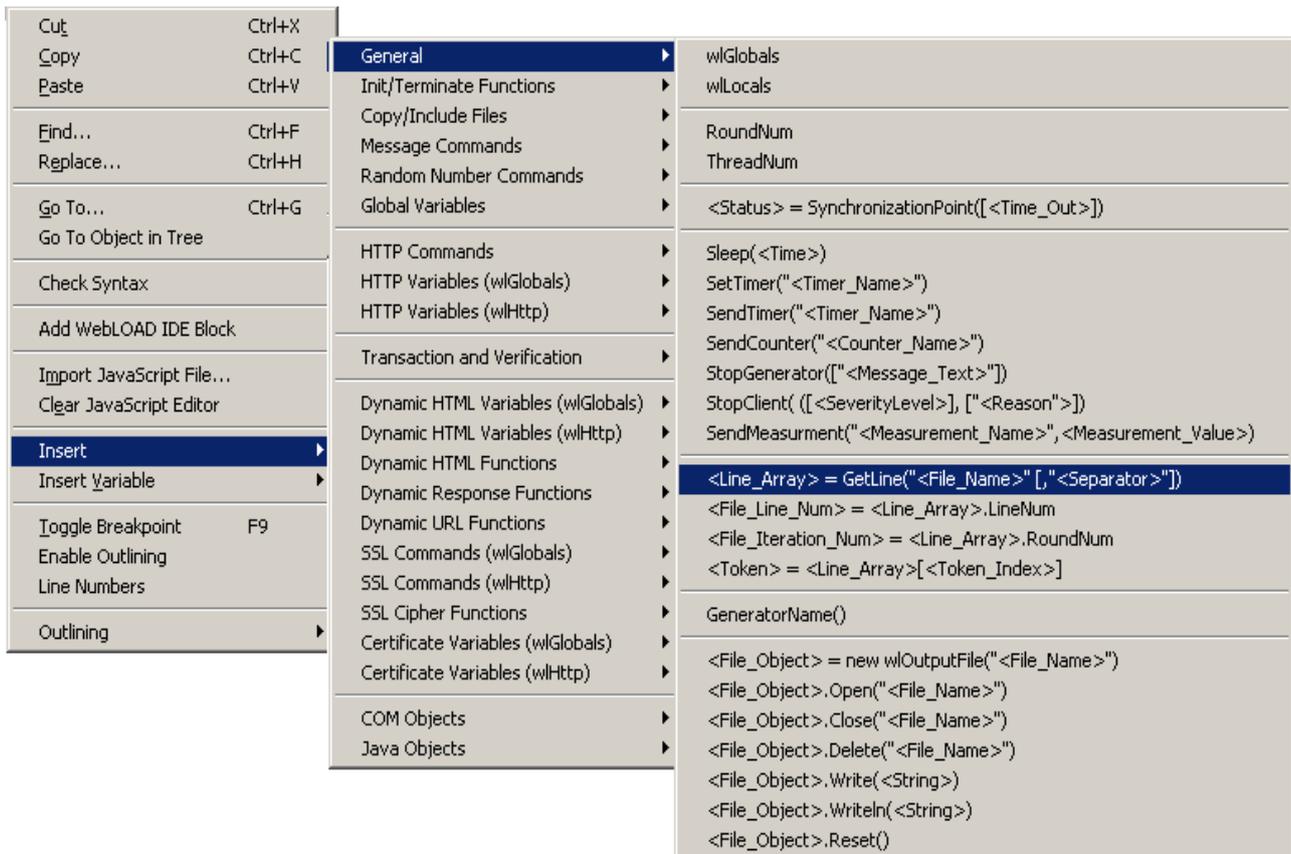
Figure 12: GetLine() Function Insertion using General Menu

JavaScript Objects are described in *Adding JavaScript Object Nodes* (on page 19). The exact JavaScript syntax of input file functions can be found in the *WebLOAD JavaScript Reference Guide*.

Sharing Input File Data

The rules of scope, described in *Rules of Scope for Local and Global Variables* (on page 49), apply to input files as well. One or more line counter variables are used to keep track of which line is to be read next in the file, cycling through the file and starting over at the beginning of the file when the end is reached. Some of the input file access configurations that are possible include:

- Multiple threads of a single Agenda on a single Load Generator may share a single input file. In this case, a single line counter variable keeps track of which lines have been read already in the file, enabling each input file access from any of the participating threads to read the next line in the file.

- A single Load Generator running multiple threads may either use a single input file or read from different files. A separate line counter variable is maintained by the Load Generator for each input file being accessed.

- Multiple Load Generators may all read from the same input file, but each Load Generator maintains its own, independent line counter for that file. A Load

Generator's line counter is incremented each time one of that Load Generator's threads reads the file.

Note: If multiple Load Generators share a single input file, using multiple line counters, it is possible that more than one Load Generator may reach the same line of the input file simultaneously, which may or may not be a problem, depending on the situation. This is illustrated in the following example.

Example: Reading an Input File

In the following example, we use an input file containing the following names and email addresses:

```
FirstName,LastName,email@address.com
John,Smith,jsmith@ABC.com
Mary,Todd,mtodd@GHI.com
Betty Sue,Jackson,bettysue@MNO.edu
Tom,Jones,invalid*email
...
```

The sample Agenda reads the lines of the file, fills in an HTML form with the data, and uploads the form to a website. Because the `GetLine()` function in this example is working in the default `WLSequential` mode, each thread of the Load Generator reads a successive line of the file. When any thread reaches the last line, the file cycles back to the beginning. In that way, the threads can obtain an unending sequence of data from the file.

```
function InitAgenda() {
   //Set the URL of the server that processes the form
   wlGlobals.Url ="http://www.ABCDEF.com/FormProc.exe"
}
//Main Agenda Body
// Read and parse a line of the input file
DataArr = GetLine("c:\\temp\\mydata.txt")
//Enter the data in simulated HTML form fields
wlHttp.FormData["FirstName"] = DataArr[1]
wlHttp.FormData["LastName"] = DataArr[2]
wlHttp.FormData["Email"] = DataArr[3]
//Post the form to the server
wlHttp.Post()
```

Suppose you configure a Load Generator with 20 clients running this Agenda. By chance, it may happen that thread 14 of the Load Generator reaches the `GetLine()` statement first. In that case, thread 14 reads John Smith's name and email address and sends the data to the server. Other threads of the Load Generator read the data for

Mary Todd, Betty Sue Jackson, Donald Gallop, etc. In the second round, thread 14 reads whatever line is next in the sequence.

The threads continue to cycle through the file in this way. When any thread of the Load Generator reaches the last line of the file, the next thread that happens to read the file starts again with the data for John Smith. If the file is large enough (say 50-100 lines), no two threads submit the same data to the server simultaneously, or even in close succession.

Now suppose you configure two different Load Generators running the same Agenda. The two Load Generators access the input file independently. Therefore, it may happen that a thread of the first Load Generator and a thread of the second Load Generator submit the data for John Smith simultaneously or in close succession. In many WebLOAD tests, this behavior is perfectly acceptable. It may be a problem, however, if the server cannot accept simultaneous accesses with the same data (for example if the server doesn't allow John Smith to log in simultaneously from two different computers). In that case, you should create a different input file for the second Load Generator and modify the Agenda to read from the second file.

Security

Security is important for most Web users, whether the access is recreational, personal, or work related. Most Web applications work with some combination of user authentication together with SSL protocol use to ensure application security. Both these security features are handled automatically by WebLOAD. By default, test sessions use the settings that were in effect during the original recording session. Specific security settings may be reset through the WebLOAD IDE or Console GUI dialog boxes.

Security methods and properties can be edited directly within a JavaScript Object in an Agenda through the IntelliSense Editor, as described in *Editing the JavaScript Code in an Agenda* (on page 14). This section provides a general introduction to the internal implementation of the user authentication and SSL security features. See the *WebLOAD JavaScript Reference Guide*, for a complete syntax specification of the security functions introduced here.

Authentication

Users are authenticated through a system of usernames and passwords. Most servers use either a basic user authentication protocol or the Windows NT Challenge Response protocol. WebLOAD supports both protocols, as well as proxy servers that require user authorization. WebLOAD automatically selects the authentication protocol appropriate for the current test session. By default, WebLOAD navigates authorization protocols using the user data saved during the original recording session, but these values may be reset as needed.

For example, many Web applications involve logging in to a site. The original user name and password typed in during a recording session are saved for use during later test sessions.

Testers may reset user authentication values in one of the following ways:

- Through the Authentication tab.
- By manually editing the authentication property set.

These options are described here.

Using the Authentication Tab

Access the Authentication tab by selecting the **Tools ➤ Default Options** menu in the WebLOAD IDE or Console. Select the Authentication tab in the dialog box that opens. You can simply type the relevant values into the input text fields, as illustrated in the following figure.

Note: Passwords will not appear in readable form.
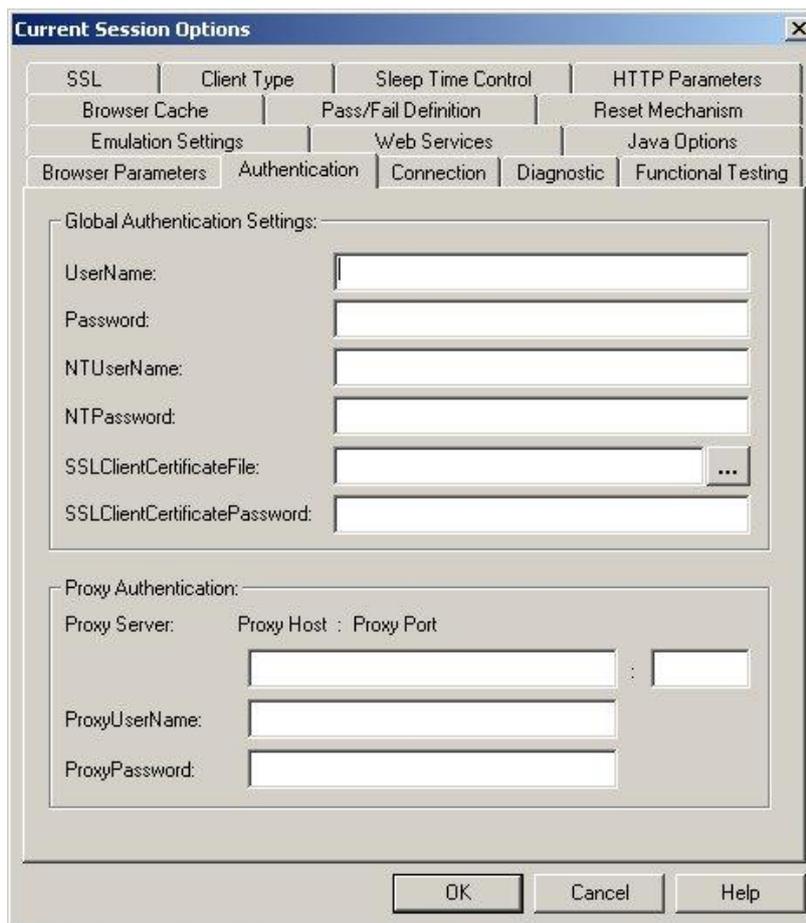
Figure 13: Authentication Tab

Setting User Authentication Values Manually

If you are editing the JavaScript code in your Agenda, you may reset user authentication values manually using the `wlGlobals`, `wlLocals`, or `wlHTTP` property set.

There are three approaches to setting these properties, corresponding to the desired authentication scope:

• Set the authentication properties that correspond to the Authentication tab at the Agenda level using wlGlobals.

• Set the authentication properties for a specific HTTP request using wlHTTP.

• Set the authentication properties that correspond to the Authentication tab at the Virtual Client level using wlLocals.

For example, set the user name and password using either of the following approaches:

```
wlGlobals.UserName = "Bill"
wlGlobals.PassWord = "TopSecret"
```

The authentication related properties include the following:

- NTUserName, NTPassWord
- PassWord
- Proxy, ProxyUserName, ProxyPassWord
- SSLClientCertificateFile, SSLClientCertificatePassWord
- UserName

See the *WebLOAD JavaScript Reference Guide*, for a complete syntax specification of the properties used for user authentication.

Secure Data Transmission through SSL

Web and HTTP communications require a practical, reliable, universally recognized protocol to ensure the secrecy, integrity, and authentication of transmitted information. The Internet Engineering Task Force (IETF) has developed such a standard, commonly referred to as SSL/TLS 1.0.

For a basic introduction to SSL, see:

http://www.cisco.com/en/US/netsol/ns340/ns394/ns50/ns140/networking_solutions_white_paper09186a0080136858.shtml

For more information on the new TLS Protocol Version 1.0, see:

http://www.ipa.go.jp/security/rfc/RFC2246-00EN.html

WebLOAD fully supports the SSL/TLS 1.0 standard, including backward compatibility with earlier versions and standards. See *Appendix A* in the *WebLOAD JavaScript Reference Guide*, for a complete list of the supported SSL protocols and ciphers.

By default, WebLOAD enables use of all cipher versions at the maximum cryptographic strength possible, allowing participating clients and servers to determine the most appropriate connection to other clients and servers through negotiation at connect time. These setting may be changed or reset through dialog boxes on the Console.

For example, reset the SSL bit limit through the SSL tab of the **Tools ➤ Default Options** dialog box on the WebLOAD Console, illustrated in the following figure:
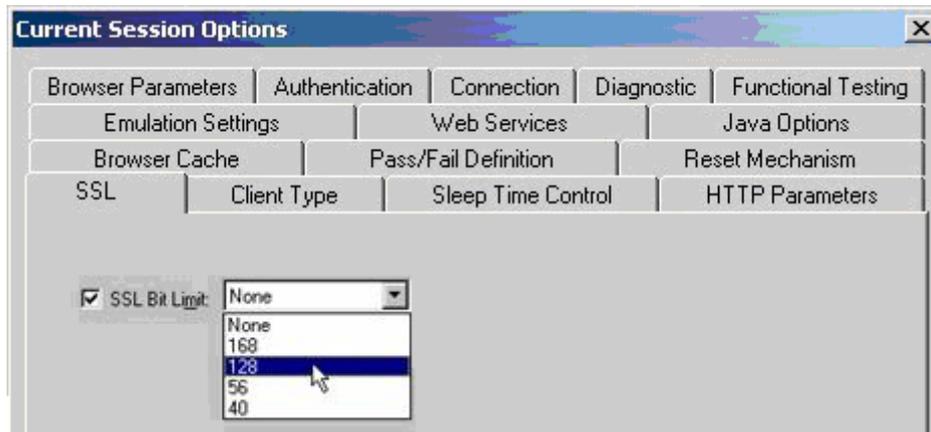

Figure 14: Resetting SSL Bit Limit

By default, WebLOAD handles all SSL activities, setting the appropriate configuration settings automatically. WebLOAD also provides both Cipher Command Suite functions and a complete set of SSL `wlGlobals` properties that allow users to fine-tune their security settings within the Agenda code, identifying, enabling, or disabling specific protocols or cryptography levels, depending on testing preferences.

Note: Changes that are made within Agenda code override anything that has been set through the Console.

These properties and functions work on two levels:

• Higher level functions used for browser emulation.

• Lower level functions for client- or server-specific cipher testing.

See the *WebLOAD JavaScript Reference Guide* for complete syntax specifications for the `wlGlobals` SSL property set and the Cipher Command Suite.

Note: The SSL configuration for your test session is usually set through dialog boxes in the Console. You may change the default session settings within your Agenda, using either the `wlGlobals` properties or the Cipher Command Suite functions. While the SSL property set is also defined for the `wlHttp` and `wlLocals` objects, RadView strongly recommends using these properties with the `wlGlobals` object *only*.

Any changes to an Agenda's SSL property configuration must be made in the Agenda's initialization functions. Configuration changes made in the `InitAgenda()` function will affect all client threads spawned during that Agenda's test session. Configuration changes made in the `InitClient()` function will affect only individual clients. Do not make changes to the SSL property configuration in an Agenda's main body. The results will be undefined for all subsequent transactions.

Browser Emulation

The browser emulation functions emphasize a high-level, categorical approach to cryptographic strength definition. For example, you may wish to test access to foreign browsers who use only earlier versions of SSL, or who are limited to export-only cryptographic strength. This 'smarter' approach of selecting appropriate categories is usually preferable to the low-level approach of enabling or disabling individual protocols.

Browser emulation functionality includes the ability to:

- Enable, disable, and list SSL protocols.

- Define the cryptographic categories to be used in the current test session.

- Limit the strength and complexity of the cipher set in current use by limiting the number of bits used by the cipher.

- Define the SSL version that WebLOAD should use for the current test session.

Server-Specific Cipher Testing

The server-specific cipher testing functions emphasize a low-level, bit-specific approach to cryptographic strength definition. For example, you may wish to enable or disable specific ciphers, to check the limits of certain client site abilities or server configurations.

SSL Cipher Command Suite provides the ability to:

- Enable and disable a specific cipher by name.

- Enable and disable a specific cipher by ID number.

- Obtain a specific cipher's name or ID number.

- Get information about a specific cipher, identified by name or ID number.

- Learn the number of ciphers enabled during the current test session.

The `wlGlobals` SSL property set provides the ability to:

- Support use of SSL client certificates by supplying the certificate filename and password to the SSL server.

- Enable caching of SSL decoding keys received from an SSL (HTTPS) server.

Complete syntax specifications for the SSL Cipher Command Suite and the `wlGlobals` SSL property set are available in the *WebLOAD JavaScript Reference Guide*.

Error Management

Agenda scripts that encounter an error during runtime do not simply fail and die. This would not be helpful to testers who are trying to analyze when, where, and why an error in their application occurs. WebLOAD IDE Agendas incorporate a set of error management routines to provide a robust error logging and recovery mechanism whenever possible.

Error Management Tools

Analyzing Possible Errors

Error management works on multiple levels that must be defined by the testers and tailored to the application being tested. For example, when first designing a test session, testers may wish to ask:

- What is considered an error for my application? How do I define an error? Are there different types of errors for this application that should possibly be handled in different ways?

- How serious a problem is this specific error, or a type of error? When is an error fatal?

- Depending on the source of the error, how do I want to handle this? Do I want to retry the command? Should I simply continue execution? Skip to the next command? Skip to the next navigation block? Stop execution completely?

- How many errors of a specific type are needed to trigger an error condition? Is even one error of this type considered intolerable and fatal to the application? Are up to 10 errors considered reasonable and will not cause damage to the application? Could too many warnings also be a reason to stop the test session?

- Does this application routinely experience a certain error rate that may safely be ignored? Perhaps a rate of up to 5% errors is considered reasonable? Perhaps the application takes a certain amount of time to get started and errors in the first 5 minutes of run time should be ignored?

The answers to these questions should be a part of your test session design. Once you have categorized the various types of errors possible, you should also analyze how you wish to handle these errors, should they occur.

Set the error configuration values through the Pass/Fail Definition and Error Handling tabs of the **Tools ➤ Default or Global Project Options** dialog boxes.

Standard Message Functions

On the most basic level, you may simply insert message functions into your Agenda and have them executed wherever anything unusual or problematic occurs. Insert a message function into your Agenda through the WebLOAD IDE GUI.

Choose from four possible message levels:

- `InfoMessage()` — prints a simple informative message to the Log window. Has no affect on Agenda execution.

- `WarningMessage()` — prints a warning message to the Log window. Has no affect on Agenda execution.

- `ErrorMessage()` — prints an error message to the Log window. Causes the current test round to abort, but the test session continues with the start of the next round.

- `SevereErrorMessage()` — prints an error message to the Log window and stops the current test session.

- `DebugMessage()` — prints a message to the IDE Log window. Has no affect on Agenda execution.

Standard Error Constants

You may also manage errors by checking function return codes, both for built-in and user-defined functions. WebLOAD IDE provides a set of constants that define the error severity level, acting as a useful means for redirecting Agenda behavior. Use the severity level to determine the execution path to be followed in case of error. Less severe errors may be noted and ignored. More severe failures may cause the whole test to be aborted. All failures are logged and displayed in the both the WebLOAD IDE and Console Log Windows. Refer to Appendix A in the *WebLOAD IDE User* Guide for more information on return codes and error levels.

WebLOAD and WebLOAD IDE include the following error levels:

- `WLSuccess` — the activity terminated successfully.

- `WLMinorError` — this specific activity failed, but the test session may continue as usual. The Agenda displays a warning message in the Log window and continues execution from the next statement.

- `WLError` — this specific activity failed and the current test round was aborted. The Agenda displays an error message in the Log window and begins a new round.

- `WLSevereError` — this specific activity failed and the test session must be stopped completely. The Agenda displays an error message in the Log window and the Load Generator on which the error occurred is stopped.

Note: These built-in message functions and error levels provide a single, standard set of error management options. After an error occurs, testers may choose to continue with the current round, stop the current round, or stop the whole test session. But not every error falls into these three broad categories. Sometimes an error is serious enough to invalidate the current activity, yet the current round may still be able to continue execution. Very often the set of activities on a specific Web page is not successful, but that does not have any affect on the activities recorded on subsequent Web pages.

Variable Activity Blocks

WebLOAD IDE offers an additional level of error management, which can be manually added to an Agenda. This is done by the `try()/catch()` function pair. Users may add `try()/catch()` pairs around their own logical blocks of code, through the IntelliSense Editor.

The `try()/catch()` pair of error handling functions allows users to catch errors, skip a single set of corrupted activities, and still continue with the current round. Use these functions to delimit a specific logical activity unit and define an alternate error-handling routine to be executed in case an error occurs within that block of code. The `try()/catch()` functions limit the scope of an error to individual logical component branches of your Agenda tree.

Limited scope for an error has two advantages:

- Pinpointing the source of the error, enabling easier debugging and recovery.
- Isolating the error to a specific Agenda branch, in many cases allowing test execution to continue with the rest of the Agenda Tree despite the error.

Navigation blocks and other logical activity blocks are useful for error management, especially when running "hands-free" test sessions. For example, the user can define the default testing behavior to be that if a non-fatal error is encountered during a test session, WebLOAD IDE should throw the error, skip to the next navigation block, and continue with the test session.

The **Pass/Fail Definition** tab of the **Tools ➤ Default or Global Project Options** dialog box is used to define the default WebLOAD IDE behavior in case of error.

Note: At each testing level, WebLOAD IDE provides a choice of error handling options. Depending on the context, you may choose to repeat the action, skip to the next line, skip to the next navigation block, or stop execution completely.

The wlException Object

Agenda scripts that encounter an error during runtime do not simply fail and die. This would not be helpful to testers who are trying to analyze when, where, and why an error in their application occurs. WebLOAD Agendas incorporate a set of error management routines to provide a robust error logging and recovery mechanism whenever possible. The `wlException` object is part of the WebLOAD error management protocol.

WebLOAD users have a variety of options for error recovery during a test session. The built-in error flags provide the simplest set of options; an informative message, a simple warning, stop the current round and skip to the beginning of the next round, or stop the test session completely. Users may also use `try()/catch()` commands to enclose logical blocks of code within a round. This provides the option of catching any minor errors that occur within the enclosed block and continuing with the next logical block of code within the current round, rather than skipping the rest of the round completely.

Users may add their own `try()/catch()` pairs to an Agenda, delimiting their own logical code blocks and defining their own alternate set of activities to be executed in case an error occurs within that block. If an error is caught while the Agenda is in the middle of executing the code within a protected logical code block (by `try()`), WebLOAD will detour to a user-defined error function (the `catch()` block) and then continue execution with the next navigation block in the Agenda.

`wlException` objects store information about errors that have occurred, including informative message strings and error severity levels. Users writing error recovery functions to handle the errors caught within a `try()/catch()` pair may utilize the `wlException` object. Use the `wlException` methods to perhaps send error messages to the Log Window or trigger a system error of the specified severity level.

Example

The following code fragment illustrates a typical error-handling routine:

```
try{
   ...
   //do a lot of things
   ...
   //error occurs here
   ...
}
```

```
catch(e){
  myException = new wlException(e,"we have a problem")

  //things to do in case of error
  if (myException.GetSeverity() == WLError) {
        // Do one set of Error activities
            myException.ReportLog()
            throw myException
  }
  else {
        // Do a different set of Severe Error activities
            throw myException
  }
}
```

Non-Standard Agenda Execution Sequence

Agenda Execution Sequence (on page 11) describes the sequence of activities during a normal, standard test session. However, many test sessions do not simply follow a normal execution sequence. Test sessions are designed to catch application errors, and these errors often interrupt the test session. This section describes what happens within your Agenda if it is interrupted in mid-session.

Minor Error Management—Continue the Test Session as Usual

In the event of a minor error, WebLOAD and WebLOAD IDE record the error in a log and continues processing. This occurs when you call an `InfoMessage()` or `WarningMessage()` function or use the `WLMinorError` constant.

For example, one of the most common runtime errors is the failure of an HTTP Get, Post, or Head command. This can occur, for example, if the HTTP server is temporarily unavailable. In most cases, an HTTP failure is a minor error and should not stop the test, although it can affect the performance statistics. You may optionally include additional error-handling functions in your Agenda.

Standard Error Management—Stopping a Single Round

In the event of a standard error, WebLOAD records the error in a log and stops the current round. This may occur under the following circumstances:

- If you call an `ErrorMessage()` function or use the `WLError` constant.
- If the user issues a Stop or Pause command in the WebLOAD Console that happens to catch a thread in mid-round.

- If a thread encounters a non-severe error such as a failed HTTP connection.

In these circumstances, WebLOAD does the following:

- Sends an error message to the Log window.

- Stops the main script in the current round.

- Runs the `OnScriptAbort()` function, if it exists in your Agenda, in the thread where the error occurred, to free objects and resources.

- Continues with the next round of the thread, at the beginning of the main script.

Non-severe errors affect only the thread where the error occurred. They have no effect on other threads of the Load Generator, and they do not run the `OnError...` functions.

Severe Error Management—Stopping a Test Session

In the event of a severe runtime error, WebLOAD sends a severe error message to the Log window and stops the whole test session. This may occur under the following circumstances:

- If you call a `SevereErrorMessage()` function or use the `WLSevereError` constant.

- If a thread encounters a severe error such as an overflow or an illegal operation, or any other error that the user has specified should trigger a severe error condition.

When stopping an entire test session, WebLOAD uses the following execution sequence.

- In the thread where the error occurred, WebLOAD runs the `OnScriptAbort()` and `OnErrorTerminateClient()` functions. These functions are used for error handling that is specific to individual threads, such as freeing local objects and resources.

- In the other threads of the Load Generator, WebLOAD runs the `OnScriptAbort()` and `TerminateClient()` functions.

- The separate threads terminate.

- WebLOAD runs the `OnErrorTerminateAgenda()` function instead of `TerminateAgenda()`. This function is used to free global resources, or for other types of error handling at the global level.

Like the other initialization and termination functions, the error-handling functions are optional. Omit them from your Agenda if they are not needed.

In the event of a severe runtime error, WebLOAD IDE sends a severe error message to the Log window and stops the whole test session. This may occur under the following circumstances:

- If you call a `SevereErrorMessage()` function or use the `WLSevereError` constant.

- If the Agenda encounters a severe error such as an overflow or an illegal operation, or any other error that the user has specified should trigger a severe error condition. A severe error is also triggered if WebLOAD IDE cannot find a specified object on the Web page, for example if the object was deleted or the current Web page does not match the expected Web page accessed during the recording session.

When stopping an entire test session, WebLOAD IDE uses the following execution sequence.

- First WebLOAD IDE runs the `OnScriptAbort()` function, used for error handling such as freeing local objects and resources.

- Then WebLOAD IDE runs the `OnErrorTerminateAgenda()` function instead of `TerminateAgenda()`. This function is used to free global resources, or for other types of error handling at the global level.

Like the other initialization and termination functions, the error-handling functions are optional. Omit them from your Agenda if they are not needed.

Defining a Standard or Severe Error

By default, WebLOAD and WebLOAD IDE only stop an Agenda if a severe error occurs, such as an overflow or an illegal operation. However, users may redefine when exactly WebLOAD should halt a test session through the WebLOAD IDE GUI through the Pass/Fail Definition tab of the **Tools ➤ Default / Current Options** dialog box. Options include:

- Failing on the first severe error occurrence.

- Specifying an absolute error threshold of failure if more than a specified number of errors or warnings occur.

- Specifying a relative error threshold of failure if more than a specified percentage of errors or warnings occur.

- Tracking numbers of errors only after a certain amount of time has elapsed from the start of the test session. This slight delay allows the test session to establish and stabilize an initial Web connection that may have no connection to the actual application being tested, preventing premature (and meaningless) session failures.

Execution Sequence after Stop and Abort

Stop an Agenda by issuing a Stop command through the Console or WebLOAD IDE GUI. Abort an Agenda during its debugging by issuing an Abort command through the WebLOAD IDE. An Abort command cannot be issued through the Console. Stop and Abort commands halt the Load Generator.

After you issue the Stop command in the Console or IDE, WebLOAD:

* Stops the main script in each thread.

* Runs the `OnScriptAbort()` function in any thread that you happen to stop in mid-round. `OnScriptAbort()` does not run in a thread that you happen to stop at the end of a round.

* Runs the `TerminateClient()` function in each thread.

* Stops the separate threads.

* Runs the `TerminateAgenda()` function of the Agenda.

Note: The Agenda doesn't necessarily stop immediately when you issue the Stop command in the Console. The Agenda checks for the command at the end of each round and during certain operations that may take a long time, such as HTTP Get and Post commands. For this reason, it is recommended that you do not program a very long loop in an Agenda. Depending on the operations in the loop, the WebLOAD Console may have no way to interrupt the loop.

After you issue the Abort command, WebLOAD IDE:

* Stops the main script in each thread.

* Runs the `OnScriptAbort()` function in any thread that you happen to stop in mid-round. `OnScriptAbort()` does not run in a thread that you happen to stop at the end of a round.

See the *WebLOAD IDE User Guide* for more information on using these commands.

Note: If you stop or abort WebLOAD IDE during execution, a message dialog appears, telling the user that WebLOAD IDE is collecting data in preparation for stopping the test session. While the window pops up immediately, it may stay open for a while if there is extensive clean-up work necessary before the current session may be shut down.

The Agenda doesn't necessarily stop immediately when you issue a Stop or Abort command. The Agenda only checks for the command at the end of each round and during certain operations that may take a long time, such as certain HTTP transactions. For this reason, it is recommended that you do not program a very long loop in an Agenda. Depending on the operations in the loop, WebLOAD IDE may have no way to interrupt the loop.

Execution Sequence after Stopping a Virtual Client

If an error condition occurs, you can stop a Virtual Client using `StopClient()` in your Agenda. This function stops the execution of the Virtual Client running the script from which `StopClient()` was called. After `StopClient()` is called, this Virtual Client cannot be resumed. This function does not affect any other Virtual Client.

When using `StopClient()`, you can specify the error level to display and a reason why the Virtual Client was terminated.

After you issue the `StopClient()` command, WebLOAD does the following on the current (calling) thread only:

- Stops the main script in the current thread.

- Runs the `OnScriptAbort()` function in case the current thread is stopped in mid-round. `OnScriptAbort()` does not run in a thread that you happen to stop at the end of a round.

- Runs the `TerminateClient()` function in the current thread.

See the *WebLOAD IDE User Guide* for more information on using these commands.

See the *WebLOAD JavaScript Reference Guide* for more information on using the `StopClient()` function.

Rules of Scope for Local and Global Variables

To maximize testing options, WebLOAD offers the ability to run multiple Agendas, in multiple threads, often across multiple Load Generators and machines. This may involve setting variables or flags or accessing data across multiple threads, Agendas, Load Generators, or machines. Such multiple accesses must be handled carefully.

This section describes the scope rules for all variables, both built-in configuration properties and user-defined shared variables, explaining how Agendas can juggle multiple configuration settings and share data either locally or globally. This information supplements the usual JavaScript scope rules, which apply within a single thread.

To understand the scope rules for user-defined variables, we need to distinguish between the following testing contexts:

- *Limited Context* (on page 50), limited to a specific browser action within an Agenda

- *Local Context* (on page 52), local to a single thread of a single Agenda

- *Global Context* (on page 53), which may be global to the following extent:
  - Shared by all threads of a single Agenda running on a single Load Generator.
  - Shared by all threads of multiple Agendas, including a mix of Agendas, running on a single Load Generator.
  - Global to all threads of a single Agenda, running on multiple Load Generators, potentially on multiple machines, system-wide.
  - Global to all threads of multiple Agendas, including a mix of Agendas, running on multiple Load Generators, potentially on multiple machines, system-wide.

Each of these testing contexts is described in the following sections.

Note: The HTTP configuration properties documented in this section may be edited through the IntelliSense Editor, as described in *Editing the JavaScript Code in an Agenda* (on page 15). However, remember that the majority of application tests rarely require anything more than perhaps a few user-defined global variables or configuration settings, which can be created and set through the WebLOAD IDE or Console dialog boxes. Manual intervention in individual configuration values within the JavaScript code of an Agenda is not usually recommended.

Limited Context

Limited variables are only visible or accessible within the bounds of a specific function or code block. Users may also define configuration properties that are only applicable to a specific browser action, managed through the `wlHttp` object.

For example, to limit a variable `LimitedX` to a specific function only, use the IntelliSense Editor to create a function within a JavaScript Object and assign a value to a `LimitedX` variable within that function. The `LimitedX` variable will only be meaningful within the context of the function in which it appears.

This is illustrated in the following code fragment:

```
//Context limited to specific function only
function MyFavoriteFunction() {
  var limitedX = 20        //Scope of limitedX is the
      //current function only
}
//Main Agenda Body
y = limitedX + 2      //Error, limitedX is undefined here
```

Working with the wlHttp Object

Configuration properties that are limited to a single transaction are managed through the `wlHttp` object, described in this section.

In WebLOAD Agendas, HTTP transactions are performed using `wlHttp` objects. You do not need to declare or create a `wlHttp` object before using it in your Agenda. WebLOAD automatically creates a single `wlHttp` object for each thread of an Agenda.

`wlHttp` objects include a complete set of HTTP configuration properties. HTTP configuration properties are also included with the more general `wlGlobals` and `wlLocals` objects. However, the configuration values assigned in a `wlHttp` object are the definitive configuration values for the transaction immediately following the property value assignment, overriding both the local defaults assigned in `wlLocals` and the global defaults assigned in `wlGlobals`. WebLOAD uses the `wlLocals` or `wlGlobals` defaults only if you do not assign values to the corresponding properties in the `wlHttp` object.

For example, one of the properties of `wlHttp` is the FormData, which stores information that the user entered during the session. In the main body of the following Agenda, the user defines `FormData` value and then completes the browser activity. For the specific command that immediately follows the `wlHttp` property assignment, the `wlHttp FormData` value will override any default local or global `FormData` value.

```
//Main Agenda Body
wlHttp.FormData["login"] = "demo"
wlHttp.FormData["password"] = "demo"
```

Erasing and Preserving the HTTP Configuration

By default, `wlHttp` properties are cleared automatically after every HTTP transaction. This lets you assign different values for each connection (for example, a different URL, user name, or form data), without having to explicitly delete your previous connection data. In general it is better to assign special `wlHttp` properties at the specific point where they are needed, in the main script and not in `InitClient()`, so they will be reassigned as needed in every round.

The decision of whether or not to erase `wlHttp` values is based on the value of the `wlHttp.Erase` property, which by default is set to true. You may optionally set the `Erase` property value to false. If `Erase` is set to false, the `wlHttp` property data will not be deleted automatically. This may be convenient if you know you will always need the same information, and do not wish to reassign the same values over and over again for each transaction. However, it could also leave you with unexpected, unintended `wlHttp` property values. See the *WebLOAD JavaScript Reference Guide*, for a complete syntax specification.

A better way to preserve the HTTP configuration is to define it using the `wlGlobals` and `wlLocals` objects, rather than `wlHttp`. The properties of `wlGlobals` and `wlLocals` are not erased unless you change them yourself. However, the recommended way to set configuration values is through the Default or Current Project Options dialog boxes under the Tools menu in the WebLOAD IDE or Console desktop.

Local Context

The local Agenda thread context is local to each individual thread of an Agenda. Local objects may not be accessed in the global context. Different threads may not access or alter the value of the same local variable, since it is local to a single thread. Different threads may only share variable values through global variables.

Local configuration properties are managed through the `wlLocals` object. Local values for both user-defined variables and configuration properties are initialized in the `InitClient()` function of an Agenda. Read or assign local values using JavaScript Object nodes added to the main script, as described in *Editing the JavaScript Code in an Agenda* (on page 15).

Working with the wlLocals Object

Configuration properties that are local to a single Agenda thread are managed through the `wlLocals` object, described in this section. You do not need to declare or create a `wlLocals` object before using it in your Agenda. WebLOAD automatically creates a single `wlLocals` object for each thread of an Agenda.

`wlLocals` is a WebLOAD-supplied local object, which sets the local default configuration for HTTP commands (overriding any global defaults set in `wlGlobals`, but overridden in turn by any values set in `wlHttp`). For example, one of the properties of `wlLocals` is the URL to which the object connects. If you set a different value of `wlLocals.Url` in each thread of an Agenda, then each thread can connect to a different URL, as the following Agenda illustrates:

```
function InitClient() {//Local context
   //Set the URL for each thread
   if (ClientNum == 0)
        wlLocals.Url = "http://www.ABCDEF.com"
   else
        wlLocals.Url = "http://www.GHIJKL.com"
}
//Thread 0 connects to www.ABCDEF.com
//All other threads connect to www.GHIJKL.com
wlHttp.Get()
```

Working with User-Defined Variables (Local Context)

Within the local context, you can define any variables that you wish. The variables are local to a single thread. By default, the scope of a local variable is the entire local thread context. For example, you can define a variable `localX` in the `InitClient()` function and use it in the main script to store different values depending on the Agenda thread or round number.

The corresponding JavaScript code may look similar to this:

```
//Local context
function InitClient() {
  //Assign a different value to the local copy
  //of localX in thread 0 only
  if (ClientNum == 0)
        {localX = 20}
  else
        {localX = 10}
}
//Main Agenda Body
//Access the local values of localX & y
y = localX + 2          //y = 22 in thread 0
  //y = 12 in all other threads
```

You can also limit the scope to a single function by defining the variable using a `var` statement, as described in *Limited Context* (on page 50).

Global Context

Global variables are universally accessible and shared by all Agenda components in all threads of an Agenda, within the initialization and termination functions as well as the main script body. Global variables and configuration properties are defined within the `InitAgenda()` initialization function of an Agenda. Once a global variable has been declared you may read and assign values to that variable at any subsequent point in your Agenda.

For example, it may be convenient to define as a global variable a message text or a URL string that you expect to use and reuse frequently within your Agenda. Or you may need a global counter that should be incremented each time any Agenda thread reaches a certain point in the test session. You could increment the variable value using JavaScript Object nodes added to the main script, as described in *Editing the JavaScript Code in an Agenda* (on page 15). Add an InfoMessage node to the Agenda Tree to check on how the global variable value changes over the course of a test session, as illustrated in the following figure:
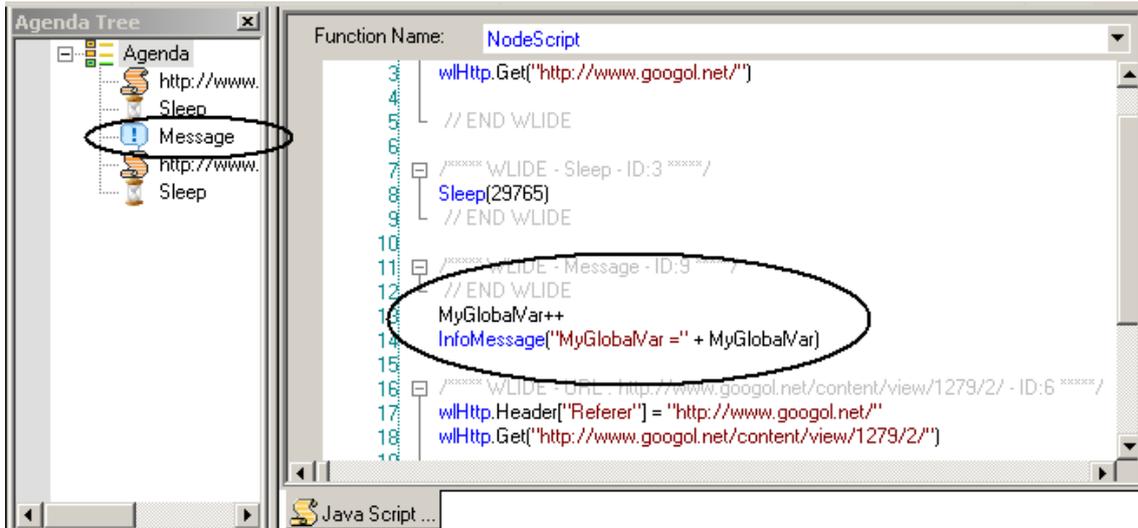
Figure 15: InfoMessage Node Addition to Agenda Tree

In the preceding figure, the Agenda contains nodes for a:

- JavaScript Object

- Short sleep period

- Message to the Log Window

The JavaScript View pane displays the corresponding JavaScript code, beginning with the navigation to the new Web page and a short sleep period. The code that corresponds to the InfoMessage node in the Agenda Tree is circled. This node is used to increment and then print the value of a user-defined global variable.

Working with the wlGlobals Object

Configuration properties that are global to all threads of a single Agenda are managed through the `wlGlobals` object, described in this section. The global Agenda context includes global variables and settings that are shared by all threads of a single Agenda, running on a single Load Generator.

Globally shared values are managed through the following objects:

- `wlGlobals`

- `wlGeneratorGlobal`

- `wlSystemGlobal`

This section describes the `wlGlobals` object properties used for global HTTP configuration settings, and reviews HTTP command search-order precedence. `wlGlobals` object properties used for user-defined global variables are described in the *WebLOAD IDE User Guide*.

WebLOAD provides a global object called `wlGlobals`. The `wlGlobals` object stores the default configuration properties for HTTP connections, such as:

- The URL
- user name and password
- proxy server
- HTML parsing behavior

WebLOAD creates exactly one `wlGlobals` object for each Agenda. You can access `wlGlobals` properties and methods anywhere in an Agenda, in both the global and local contexts.

Initialize and set the properties of `wlGlobals` in the `InitAgenda()` function of your Agenda. The values are then set globally and shared by each Agenda's set of threads. For example, set a property such as `wlGlobals.DisableSleep` in `InitAgenda()` for it to automatically have the same value for all rounds. The value may be overridden by resetting the `wlHttp.DisableSleep` value, as illustrated in the following Agenda:

Note: Remember that HTTP configuration settings and sleep configuration preferences are usually set directly through a dialog box in the WebLOAD IDE or Console desktop, as described in the *WebLOAD IDE User Guide* and the *WebLOAD Console User Guide*. The JavaScript coding example here is only intended to illustrate the balance between global defaults and local overrides.

```
// Initialization function—wlGlobals context
function InitAgenda() {
  // By default, do not include 'sleep pauses' in this Agenda
  wlGlobals.DisableSleep = true
}
// Main Script Body—wlHttp context
if (RoundNum<20)
  wlHttp.DisableSleep = false
...
```

WebLOAD executes the `InitAgenda()` method when test execution first starts, and decides by default to disable all sleep pauses over the course of Agenda execution:

```
wlGlobals.DisableSleep = true
```

For rounds 0-19, the main script overrides the global default and allows pausing during Agenda execution:

```
wlHttp.DisableSleep = false
```

After the first 20 rounds of execution are completed, there is no local value assigned, so the Agenda will again, by default, no longer allow sleep pauses.

`wlGlobals` object commands may be added directly to the code in a JavaScript Object within an Agenda through the IntelliSense Editor, as described in *Editing the JavaScript Code in an Agenda* (on page 15). Users who are programming their own JavaScript Object code within their Agenda may take advantage of the WebLOAD IDE GUI to simplify their programming efforts. Manually typing out the code to create a `wlGlobals` object method or property, risks making a mistake, and adding invalid code to the Agenda file. Instead, users may bring up a list of available objects for the `wlGlobals` object, by right-clicking in the JavaScript Editing Pane, selecting **Insert ▶ General** from the pop-up menu, and selecting one of the available items.
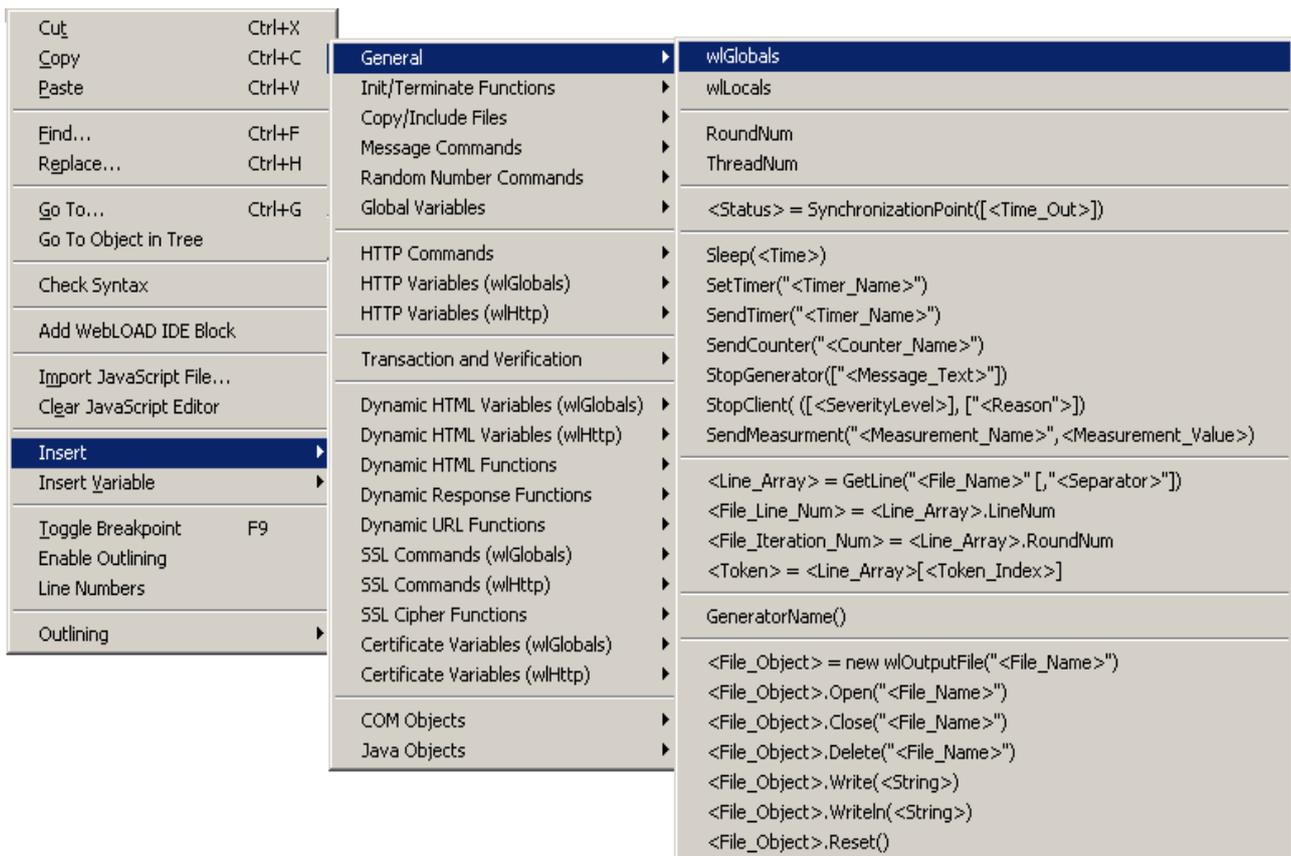

Figure 16: wlGlobals Insertion using General Menu

Select `wlGlobals` from the list and WebLOAD IDE automatically inserts the correct code for a `wlGlobals` object into the Agenda code currently being edited. The IntelliSense Editor then helps programmers write the JavaScript code for `wlGlobals` properties and methods by bringing up a list of appropriate choices together with pop-up boxes that describe each item on the list, as illustrated in the following figure:
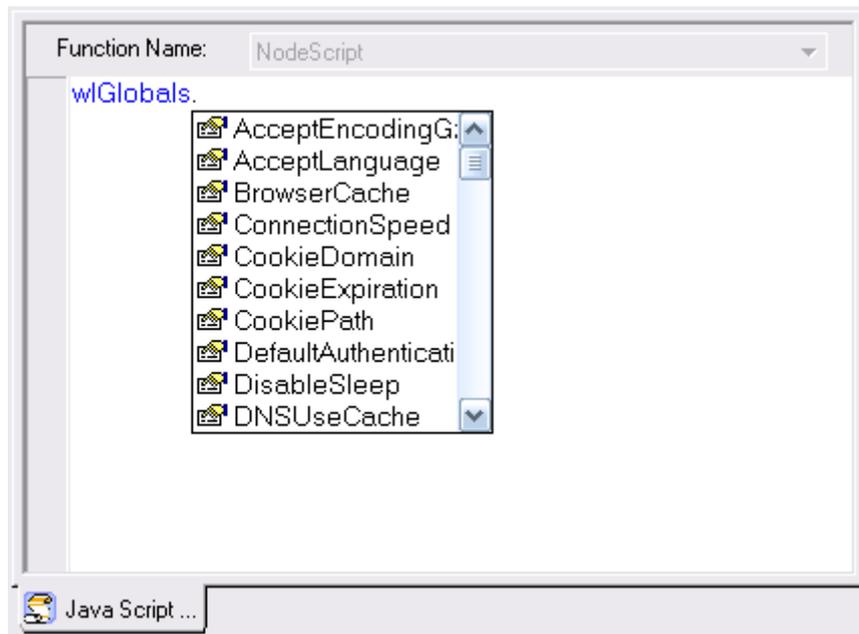
Figure 17: wlGlobals Selection from List

Working with the IntelliSense Editor is described in *Editing the JavaScript Code in an Agenda* (on page 15). See the description of `wlGlobals` objects in the *WebLOAD JavaScript Reference Guide* for a complete syntax specification.

Variables Defined through the wlGeneratorGlobal Object with WLCurrentAgenda Flag

For users who are comfortable working inside the JavaScript code of their Agendas, WebLOAD provides a global object called `wlGeneratorGlobal`. The `wlGeneratorGlobal` object, when used with the `WLCurrentAgenda` scope flag, stores variable values that you wish to share between all threads of a single Agenda, running on a single Load Generator. WebLOAD creates exactly one `wlGeneratorGlobal` object for each Agenda. You can access `wlGeneratorGlobal` properties and methods anywhere in the Agenda, in both the global and local contexts.

`wlGeneratorGlobal` includes the following methods:

- `Set("SharedVarName", value, WLCurrentAgenda)` — assigns a number, Boolean, or string value to the specified shared variable. If the variable does not exist, WebLOAD will create a new variable.

- `Get("SharedVarName", WLCurrentAgenda)` — returns the current value for the specified shared variable. The calling Agenda thread waits for the requested value to be returned before continuing execution.

- `Add("SharedIntVarName", number, WLCurrentAgenda)` — increments the specified shared number variable by the specified amount. If the variable has not

been declared or Set previously, the Add function both declares and sets the variable to the specified value.

Variables Defined through the wlGeneratorGlobal Object with WLAllAgendas Flag

The shared multiple Agenda context includes global variables and settings that are shared by all threads of a multiple Agendas. The wlGeneratorGlobal object, when used with the WLAllAgendas scope flag, stores variable values that you wish to share between all threads of one or more Agendas, including a mix of Agendas, part of one or more spawned processes, running on a single Load Generator. You can access wlGeneratorGlobal properties and methods anywhere in the Agenda, in both the global and local contexts.

wlGeneratorGlobal includes the methods with the WLAllAgendas scope flag instead of the WLCurrentAgenda flag:

- Set("SharedVarName", value, WLAllAgendas)
- Get("SharedVarName", WLAllAgendas)
- Add("SharedIntVarName", number, WLAllAgendas)

wlGeneratorGlobal Example

wlGeneratorGlobal properties and methods can be accessed from anywhere within an Agenda. They do not have to be set or initialized in any special place, and are not limited in where they may appear. Simply use the variables as necessary within your Agenda code.

For example:

```
function InitAgenda() {
  B = new Number
  B = 0
  wlGeneratorGlobal.Set("B", 1, WLAllAgendas)
  wlGeneratorGlobal.Set("C", 4, WLCurrentAgenda)
}
function InitClient() {
  wlGeneratorGlobal.Add("B", 2, WLAllAgendas)
  wlGeneratorGlobal.Add("C", 5, WLCurrentAgenda)
  InfoMessage("C—agenda1 = " +
        wlGeneratorGlobal.Get("C", WLCurrentAgenda))
}
// Main Agenda Body
B = wlGeneratorGlobal.Get("B", WLAllAgendas)
```

```
if (RoundNum==1){
   InfoMessage("B—agenda1 = " + B)
}
Sleep(1000)
wlGeneratorGlobal.Add("B", 10 , WLAllAgendas)
InfoMessage("B2—agenda2 = " +
   wlGeneratorGlobal.Get("B", WLAllAgendas))
wlGeneratorGlobal.Add("C", 25, WLCurrentAgenda)
InfoMessage("C—agenda1 = " +
   wlGeneratorGlobal.Get("C", WLCurrentAgenda))
...
```

Note: In the preceding example, the globally shared value of "B" is assigned to a local variable B. If you plan to use many InfoMessage commands, it is more efficient to assign the value to a local variable, rather than using a new `wlGeneratorGlobal.Get` call each time.

See *Editing the JavaScript Code in an Agenda* (on page 15), for an introduction to editing your JavaScript code with the IntelliSense Editor. See the description of the `wlGeneratorGlobal` object in the *WebLOAD JavaScript Reference Guide* for a complete syntax specification.

Variables Defined through the wlSystemGlobal Object with WLCurrentAgenda Flag

The global Agenda context includes global variables and settings that are shared by all threads of a single Agenda, system-wide. The `wlSystemGlobal` object, when used with the `WLCurrentAgenda` scope flag, stores variable values that you wish to share between all threads of a single Agenda, part of one or more spawned processes, potentially running on multiple Load Generators, and multiple machines. You can access `wlSystemGlobal` properties and methods anywhere in the Agenda, in both the global and local contexts.

`wlSystemGlobal` includes essentially the same methods described in *Variables Defined through the wlGeneratorGlobal Object with WLAllAgendas Flag* (on page 57), applied here to global variables:

- `Set("GlobalVarName", value, WLCurrentAgenda)` —assigns a number, Boolean, or string value to the specified global Agenda variable. If the variable does not exist, WebLOAD will create a new variable.

- `Get("GlobalVarName", WLCurrentAgenda)` —returns the current value for the specified global variable. The calling Agenda thread waits for the requested value to be returned before continuing execution.

- `Add("GlobalIntVarName", number, WLCurrentAgenda)` —increments the specified global number variable by the specified amount. If the variable has not

been declared or `Set` previously, the `Add` function declares and sets the variable to the specified value.

Variables Defined through the wlSystemGlobal Object with WLAllAgendas Flag

The global multiple Agenda context provides full, system wide global variables, shared by all elements of a test session. The `wlSystemGlobal` object, when used with the `WLAllAgendas` flag, stores variable values that you wish to share between all threads of all Agendas, part of one or more spawned processes, on all Load Generators and machines, system-wide. You can access `wlSystemGlobal` properties and methods anywhere in the Agenda, in both the global and local contexts.

`wlSystemGlobal` includes essentially the same methods described in *Variables Defined through the wlSystemGlobal Object with WLCurrentAgenda Flag* (on page 59), with the `WLAllAgendas` scope flag in stead of the `WLCurrentAgenda` flag:

- `Set("GlobalVarName", value, WLAllAgendas)`

- `Get("GlobalVarName", WLAllAgendas)`

- `Add("GlobalIntVarName", number, WLAllAgendas)`

wlSystemGlobal Example

`wlSystemGlobal` properties and methods can be accessed from anywhere within an Agenda. They do not have to be set or initialized in any special place, and are not limited in where they may appear. Simply use the variables as necessary within your Agenda code.

For example:

```
function InitAgenda() {
<snip>
}
function InitClient() {
  wlSystemGlobal.Set("E", -2, WLCurrentAgenda)
  wlSystemGlobal.Set("D", 0, WLAllAgendas)
  wlSystemGlobal.Set("S","http://www.yahoo.com",WLAllAgendas)
}
// Main Agenda Body
wlSystemGlobal.Add("D", 100, WLAllAgendas)
if (RoundNum==1){
  InfoMessage("D—agenda4 = " +
        wlSystemGlobal.Get("D", WLAllAgendas))
wlSystemGlobal.Add("E", 0.5, WLCurrentAgenda)
```

```
if (RoundNum==1){
  InfoMessage("E—agenda3 = " +
        wlSystemGlobal.Get("E", WLCurrentAgenda))
}
Sleep(1000)
```

See *Editing the JavaScript Code in an Agenda* (on page 15), for an introduction to editing your JavaScript code with the IntelliSense Editor. See the description of `wlSystemGlobal` objects in the *WebLOAD JavaScript Reference Guide* for a complete syntax specification.

Example: Using a Combination of Global Variables

The following sample Agenda combines the different global variable options introduced in the previous sections:

```
function InitAgenda() {
  wlGeneratorGlobal.Set("C", 4, WLCurrentAgenda)
  wlGeneratorGlobal.Set("B", 1, WLAllAgendas)
  <snip>
}
function InitClient() {
  wlGeneratorGlobal.Add("C", 5, WLCurrentAgenda)
  InfoMessage("C - agenda1 = " +
        wlGeneratorGlobal.Get("C", WLCurrentAgenda))
  wlGeneratorGlobal.Add("B", -2, WLAllAgendas)
  wlSystemGlobal.Set("E", +1, WLCurrentAgenda)
  wlSystemGlobal.Set("D", 0, WLAllAgendas)
  wlSystemGlobal.Set("S","http://www.yahoo.com",WLAllAgendas)
}
// Main Agenda Body
B = wlGeneratorGlobal.Get( "B", WLAllAgendas)
wlSystemGlobal.Add("E", -1, WLCurrentAgenda)
wlSystemGlobal.Add("D", 100, WLAllAgendas)
if (RoundNum==1){
  InfoMessage("B - agenda1 = " + B)
  InfoMessage("E - agenda1 = " +
        wlSystemGlobal.Get( "E", WLCurrentAgenda))
    InfoMessage("D - agenda1 = " +
            wlSystemGlobal.Get( "D", WLAllAgendas))
}
wlHttp.Get(wlSystemGlobal.Get("S", WLAllAgendas))
Sleep(1000)
```

Search Order Precedence

Before WebLOAD begins a browser action, it searches for configuration parameters in the following order of precedence:

1. `wlHttp` properties (configuration property values limited to a specific browser object)

2. `wlLocals` properties (configuration property values that are local to a specific Agenda thread)

3. `wlGlobals` properties (configuration property values that are global defaults, set at Agenda initialization)

For example:

1. WebLOAD decides whether or not it will use recorded sleep times based on the value of `wlHttp.DisableSleep`.

2. If `wlHttp.DisableSleep` is not set, WebLOAD then searches for a value in the local default `wlLocals.DisableSleep`.

3. Finally, if both `wlHttp.DisableSleep` and `wlLocals.DisableSleep` are both not set, WebLOAD will use the global default value found in `wlGlobals.DisableSleep`, set at Agenda initialization.

The same order of precedence applies to all the `wlGlobals`, `wlLocals`, and `wlHttp` properties. You can set global and local defaults for the `UserName`, `Url`, or other configuration properties. When WebLOAD executes a transaction, it uses any configuration property values that you have specified for that specific connection. If you have not assigned a value for the connection, WebLOAD searches for the local and global defaults, in that order.

Note: The recommended way to set configuration values is through the Default or Current Project Options dialog boxes under the Tools menu in the WebLOAD IDE or Console desktop. Explicitly setting these values for a specific browser action, which is possible only if you set these properties within your Agenda code, does give you a much more focused and sophisticated power, but it is also more risky, triggering unexpected side effects or complications. Configuration property settings within an Agenda will *always* override the configuration property values set globally through the WebLOAD GUI. Keep these points in mind when deciding how to set your system configuration to get the most out of your Agenda test session.

Global Sharing Considerations

A WebLOAD test session measures an application's performance. The testing tool, your Agenda, must consume minimal system resources to have as small an impact as possible on the performance of the application being tested. To minimize system

overhead when running your test Agenda, WebLOAD recommends using the minimal level of global sharing required to run your test correctly.

Variables that are accessible within a single Agenda only have the smallest impact on system performance, as noted in *wlGeneratorGlobal Example* (on page 58). For maximum efficiency and simplicity, WebLOAD strongly recommends creating and managing user-defined variables through the WebLOAD IDE GUI, as described in the *WebLOAD IDE User Guide*.

Values shared between multiple Agendas, but within a single Load Generator, are the next most efficient choice. Global values shared system wide should only be included in your Agenda if it is truly necessary to check values or synchronize data shared across multiple Load Generators or machines.

To optimize performance times during a test, multiple access requests from multiple sources are accommodated sequentially. Agenda execution is not held for the `Set()` or `Add()` methods, which are queued and processed sequentially. Only the `Get()` method actually holds an Agenda while waiting for the requested value to be returned.

Over the course of a typical test session, multiple Agenda threads may all be accessing, and potentially changing, the values of the same global variables. For example, two threads may each get `GlobalVarX`, currently set to 10. The two threads may then wish to increment `GlobalVarX` by 1, setting it to a new value of 11. These two `Get()` and two `Set()` commands are processed sequentially, independent of each other. Since each Agenda got an initial value of 10, each Agenda will independently set the value to 11. The final value will be set to 11, even though the variable was actually incremented twice, and should now equal 12. To avoid this potential inconsistency, WebLOAD recommends using the `Add()` command, which gets, changes, and resets a global value in a single action, ensuring /that each thread always accesses the most current value for a global variable and avoiding potential conflicts between threads.

Identification Variables and Functions

For performance statistics to be meaningful, testers must be able to identify the exact point being measured. WebLOAD therefore provides the following identification variables and functions:

- Two variables, `ClientNum` and `RoundNum`, identify the client and round number of the current Agenda instance.

- The `GeneratorName()` function identifies the current Load Generator.

- The `GetOperatingSystem()` function identifies the operating system of the current Load Generator.

- The `VCUniqueID()` function identifies the current Virtual Client instance.

An example is provided at the end of this section to illustrate common use of these variables and functions. Use these variables and function to support the WebLOAD measurement features and obtain meaningful performance statistics.

These identification variables and functions are usually accessed and inserted into Agenda files directly through the WebLOAD IDE GUI.

To open the Variables Window:

1. Start debugging. Click **Run** ▶ or **Step Into** 🔁.

2. In the main window, click **View ➤ Debug Windows ➤ Variables**,

   -Or-

   Click the **Variables Window** 🔲 toolbar button.
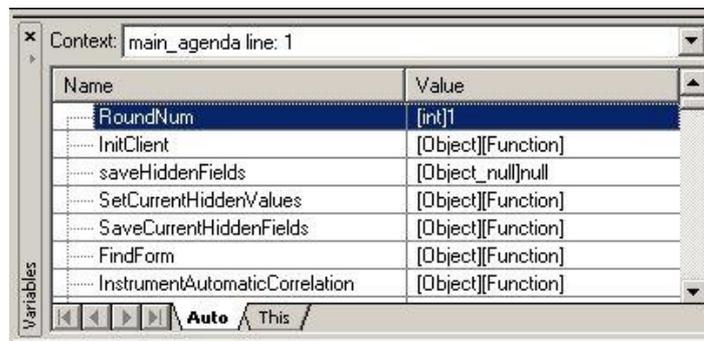
Figure 18: Variables Window

The window includes two tabs:

- **Auto** tab: Displays variables used in the current statement and in the previous statement. It also displays return values when you step over or out of a function.

- **This** tab: Displays the object pointed to by **this**.

Each tab contains a grid with fields for the variable name and value. The debugger automatically fills in these fields. You cannot add variables or expressions to the Variables window (you must use the Watch window), but you can expand or collapse the variables shown. You can expand an array, object, or structure variable in the Variables window if it has a plus sign (+) box in the Name field. If an array, object, or structure variable has a minus sign (-) box in the Name field, the variable is already fully expanded.

In addition to the tabs, the Variables window has a **Context** box that displays the current scope of the variables displayed. To view variables in a different scope, select the scope from the drop-down list box.

Viewing the Value of a Variable

You can view the value of a variable in the Variables window.

To view a variable in the Variables window:

1. Start debugging. Click **Run** ▶ or **Step Into** 🖫.

2. In the main window, click **View ➤ Debug Windows ➤ Variables**,

   -Or-

   Click the **Variables Window** 🖼 toolbar button.

3. Click the **Auto** or **This** tab, according to the type of variables you want to see.

ClientNum

This is the serial number of the client in the WebLOAD test configuration. `ClientNum` is a read-only local variable. Each client in a Load Generator has a unique `ClientNum`. However, two clients in two different Load Generators may have the same `ClientNum`.

Note: `ClientNum` is not unique system wide. Use *VCUniqueID()* (on page 68), to obtain an ID number which is unique system-wide.

Add `RoundNum` directly to the code in any JavaScript Object in your Agenda. Work through the IntelliSense Editor, described in *Using the IntelliSense JavaScript Editor* (on page 16).

For example, if there are N clients in a Load Generator, the clients are numbered `0`, `1, 2, ...,` `N-1`. You can access `ClientNum` anywhere in the local context of the Agenda (`InitClient()`, main script, `TerminateClient()`, etc.). `ClientNum` does not exist in the global context (`InitAgenda()`, `TerminateAgenda()`, etc.).

If you mix Agendas within a single Load Generator, instances of two or more Agendas may run simultaneously on each client. Instances on the same client have the same `ClientNum` value.

`ClientNum` reports only the main client number. It does not report any extra threads spawned by a client to download nested images and frames. See WebLOAD Actions, Objects, and Functions in the *WebLoad JavaScript Reference Guide* for more information about spawning threads.

Note: Earlier versions of WebLOAD referred to this value as `ClientNum`. The variable name `ClientNum` will still be recognized for backward compatibility.

RoundNum

The number of times that WebLOAD has executed the main script of a client during the WebLOAD test, including the current execution. `RoundNum` is a read-only local variable, reporting the number of rounds for the specific WebLOAD client, no matter how many other clients may be running the same Agenda.

`RoundNum` does not exist in the global context of an Agenda (`InitAgenda()`, etc.). In the local context:

- In `InitClient()`, RoundNum = `0`.

- In the main script, RoundNum = `1, 2, 3, ....`

- In `TerminateClient()`, `OnScriptAbort()`, or `OnErrorTerminateClient()`, RoundNum keeps its value from the final round.

The WebLOAD clients do not necessarily remain in synchronization. The `RoundNum` may differ for different clients running the same Agenda.

If a thread stops and restarts for any reason, the `RoundNum` continues from its value before the interruption. This can occur, for example, after you issue a Pause command from the WebLOAD Console.

If you mix Agendas in a single Load Generator, WebLOAD maintains an independent round counter for each Agenda. For example, if `agenda1` has executed twice and `agenda2` has executed three times on a particular thread, the `RoundNum` of `agenda1` is `2` and the `RoundNum` of `agenda2` is `3`.

Add `RoundNum` directly to the code in an Agenda through the IntelliSense Editor, described in *Editing the JavaScript Code in an Agenda* (on page 15). Select a function by right-clicking the JavaScript Editing Pane, and selecting **Insert ➤ General** from the pop-up menu. WebLOAD IDE automatically inserts the correct code for the selected function into the Agenda file. The user may then edit parameter values without any worries about mistakes in the function syntax.
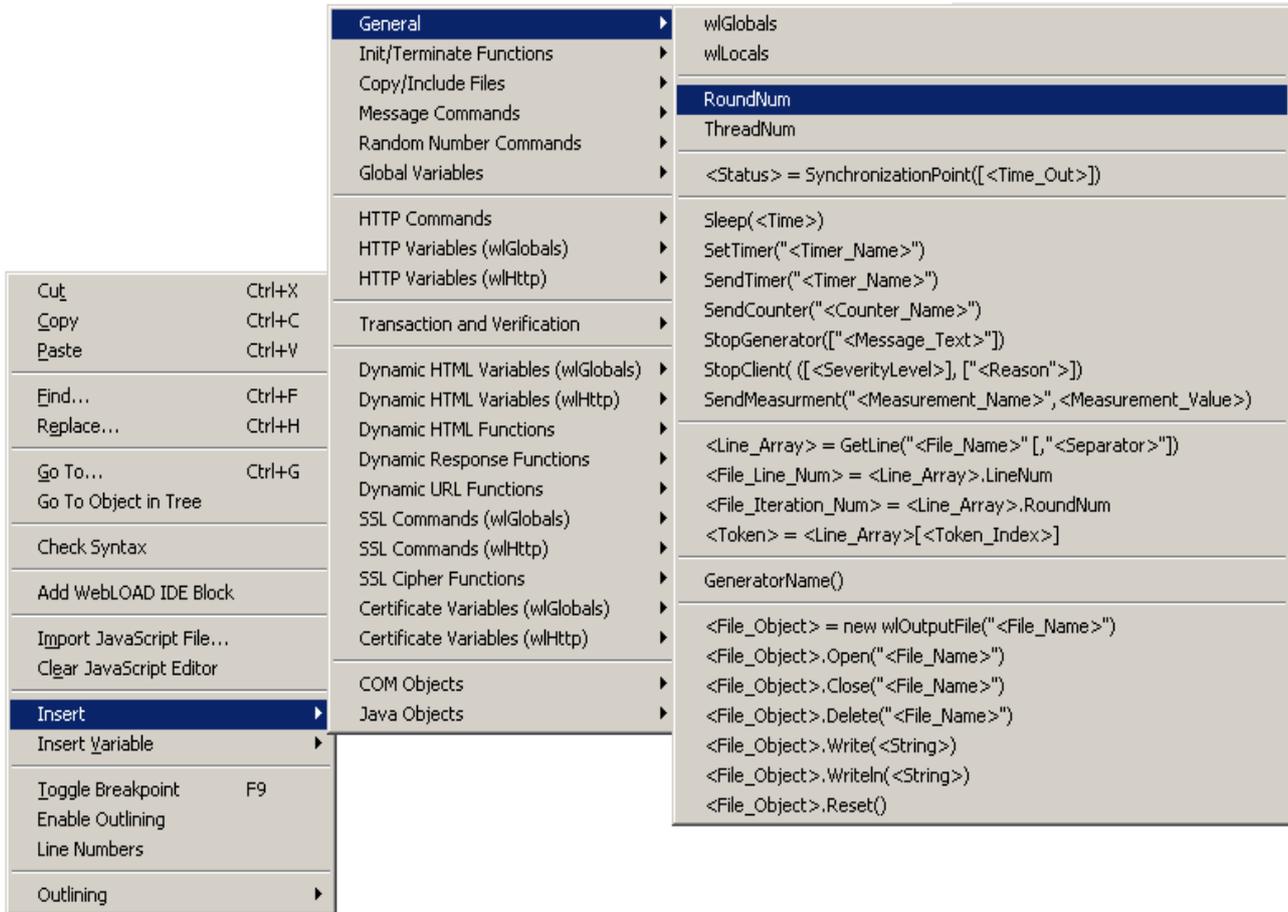
Figure 19: Adding RoundNum using the Copy/Include Files Menu

Example: identifying a client and round

For example, suppose your Agenda submits data to a server on an HTML form. You want to label one of the form fields so you can tell which WebLOAD client submitted the data, and in which round of the main script.

You can do this using a combination of the `ClientNum` and `RoundNum` variables. Together, these variables uniquely identify the WebLOAD client and round. For example, you can submit a string such as the following in a form field:

```
"C" + ClientNum.toString() + "R" + RoundNum.toString()
```

GeneratorName()

This returns a unique identification string for the current Load Generator. `GeneratorName()` provides a unique identification for the current Load Generator instance, even with multiple spawned processes running simultaneously. The identification string is composed of a combination of the current Load Generator name, computer name, and other internal markers.

GetOperatingSystem()

This returns a string that identifies the operating system running on the current Load Generator. If an operating system is available in more than one version, returns the name of the operating system followed by an identifying version number.

For example:

- If the Load Generator is working with a Windows platform, possible return values include:
    - Windows 95
    - Windows 98
    - Windows NT/2000
    - Windows XP
    - Windows OtherVersionNumber
- If the Load Generator is working with a Solaris platform, this function would return the string 'Solaris' followed by the version name and release number, such as SunOS2.
- If the Load Generator is working with a Linux platform, this function would return the string 'Linux' followed by the version name and release number, such as RedHat1.

VCUniqueID()

This returns a unique identification string for the current Virtual Client instance.

VCUniqueID() provides an identification for the current Virtual Client instance which is unique system-wide, across multiple Load Generators, even with multiple spawned processes running simultaneously. Compare this to *ClientNum* (on page 65), which provides an identification number that is only unique within a single Load Generator. The identification string is composed of a combination of the current thread number, round number, and other internal markers.

Chapter 3

Advanced JavaScript Agenda Features

Previous chapters discussed the basic set of Agenda tools used to test typical Internet applications. However, to evaluate and test your Internet application most effectively, it is not enough to measure the total transaction turn-around time or throughput of the website. A website is often only the front-end for applications that reside behind the scenes, for back-end components that do the actual work for an application. Testing the components that stand behind your Internet application is essential for comprehensive Internet application testing. A detailed breakdown and analysis of the results of each user activity helps pinpoint the source of any potential bottlenecks or other problems, and expedites the work required for any corrections or improvements.

WebLOAD fully supports component based testing. WebLOAD IDE Agendas do not simply measure the time required for transactions to a website to be completed. WebLOAD JavaScript Agenda objects can also directly access the back-end applications that compose your Internet application.

WebLOAD provides a single, uniform environment for all your testing needs. WebLOAD JavaScript is the single scripting language that is able to work seamlessly with COM, Java, and XML DOM, accessing the applications that use these technologies through a variety of protocols, or your own proprietary protocol, in addition to standard Web-site testing, adding power to and increasing the universal applicability of your testing Agendas.

This chapter introduces various component-based testing features.

Syntax specifications for the WebLOAD objects that appear in the examples in this chapter are provided in the *WebLOAD JavaScript Reference Guide*.

Working with the XML DOM

XML is a meta-language developed by W3C to organize and transfer data in a generic, universally recognized manner. XML relies on a simple, logical structure that is both easy to learn and works quickly and reliably, focusing on transferring hard data only, with no formatting or presentation information. XML acts as a gateway between autonomous, heterogeneous, component-based systems. This allows users to connect

or link to any platform. XML provides an elegant solution for Web masters who wish to reach a wide range of clients, working on any number of diverse systems.

WebLOAD provides full support for work with the XML Document Object Model. Using XML DOM objects, WebLOAD Agendas are able to both *access* XML site information, and *generate new XML data* to send back to the server for processing, taking advantage of all the additional benefits that XML provides.

WebLOAD supports:

- Working with existing XML Data Islands. Data Islands are XML documents embedded within HTML documents. WebLOAD IDE, like the IE Browser, produces an XML DOM object for each Data Island.

- Creating new XML DOM objects via the XML Object Constructor. WebLOAD supports XML Native Browsing through use of the XML Object Constructor.

- Parsing and manipulating any XML data, using the XML Parser Object.

Both WebLOAD and the IE Browser use the MSXML parser to create XML DOM objects. Since WebLOAD XML DOM objects and Browser XML DOM objects are created by the same MSXML parser, the XML DOM objects that are produced for both WebLOAD and the IE Browser are identical.

When working through the IE Browser, XML DOM objects are accessed through the `all` collection. When working through WebLOAD, XML DOM objects are accessed through the `wlXmls` collection. Since a WebLOAD XML DOM object is identical to an IE Browser XML DOM object, the WebLOAD XML DOM uses the same Document Interface (programming methods and properties) found in the IE Browser XML DOM.

The next few sections of this guide explain basic XML concepts and usage. Look in the *WebLOAD JavaScript Reference Guide* for a description of the WebLOAD `wlXmls` collection syntax and for a complete list of the WebLOAD-supported XML DOM Interfaces.

For more general XML support and cross platform capabilities, WebLOAD offers the XMLParserObject as another alternative for accessing XML data. The XMLParserObject is based on the open source Xerces XML parser. In addition to the multi-platform support, using this object will result in lower memory consumption and increased performance during load testing.

The following sections contain examples using the XMLParserObject. For more information about the XMLParserObject please refer to the *WebLOAD JavaScript Reference Guide. A*dditional information about the XMLParserObject's underlying Xerces component can be found at <u>http://xml.apache.org/xerces-c/</u>.

WebLOAD XML DOM Objects

WebLOAD XML DOM objects produced from HTML documents may be used on two different levels:

- The simplest approach is to use the XML DOM object to work directly with text strings through the standard HTML properties id, src, and innerHTML. These properties refer to the text strings found within an HTML document.

- On a more sophisticated level, programmers may use the same XML DOM object to work with a full set of XML DOM Document Interface properties and methods, as listed in *Appendix B* of the *WebLOAD JavaScript Reference Guide*. The following figure illustrates these options:
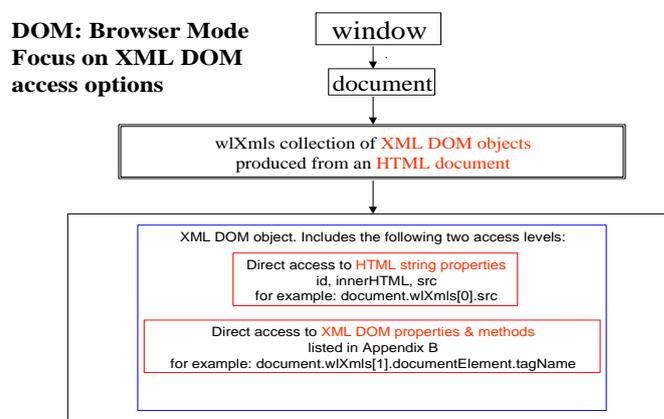
Figure 20: XML DOM Object Options

See the *WebLOAD JavaScript Reference Guide* for a complete description of the `wlXmls` collection.

Data Islands

Data Islands are XML documents embedded within HTML documents. Data Islands found on HTML documents are located either between a set of `<xml>` and `</xml>` tags or between a set of `<script>` and `</script>` tags. Data Islands consist of either a complete body of in-line XML code or a reference to XML code found in another location, with the location specified in a `src=location` statement.

WebLOAD, like the Internet Explorer browser, produces an XML DOM object for each Data Island. No matter how a Data Island is specified, either within a set of `<xml>` tags or a set of `<script>` tags, either as in-line XML code or through a reference to another location, WebLOAD produces a full XML DOM object for each Data Island. These XML DOM objects are accessible through the `wlXmls` collection.

Data Island XML DOM objects expose both aspects of each XML DOM object, (as described in *WebLOAD XML DOM Objects* (on page 71)):

- The standard HTML properties id, src, and innerHTML. These properties refer to text strings found within the HTML document.

- The XML Document Interface. The interface provides access to the properties and methods of the XML DOM object.

The following figure illustrates the various Data Islands options.

Note: Not all these Data Islands would be found in a single `wlXmls` collection if they have been taken from different sources. They appear here in a single figure to illustrate the various Data Island possibilities.
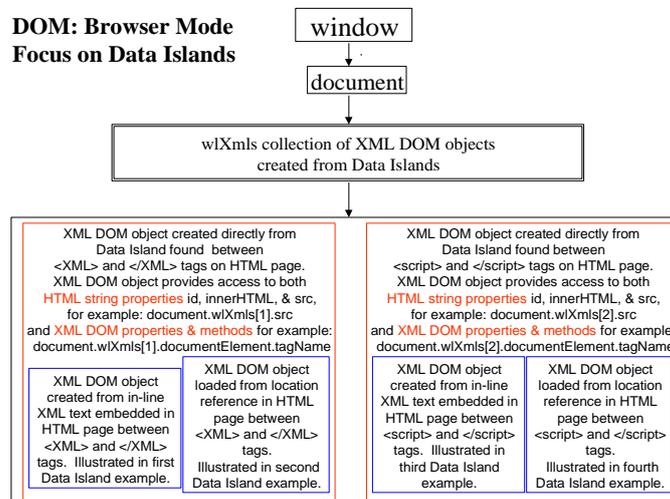
Figure 21: Data Island Possibilities

Data Islands are accessible through WebLOAD Agendas using the `wlXmls` collection of XML DOM objects corresponding to all the Data Islands found in an HTML document. Each Data Island produces a different XML DOM object.

For example, when working with an HTML document that includes two Data Islands, use the following code to assign the XML object that references the first Data Island to another object:

```
XMLDataIsland1 = document.wlXmls[0]
To access the second Data Island use:
XMLDataIsland2 = document.wlXmls[1]
```

If a Data Island is identified by name through an ID, you could also access the Data Island through the id property. For example, if the first Data Island begins:

```
<xml id="xmlBookstoreDatabase">
```

Then you could access the Data Island using any of the following:

```
MyBookstore = document.wlXmls.xmlBookstoreDatabase
MyBookstore = document.wlXmls["xmlBookstoreDatabase"]
MyBookstore = document.wlXmls[0]
```
The following HTML code fragments illustrate these Data Islands options.

Data Island—In-Line Code

In this code fragment, in-line XML data is embedded in the HTML page. WebLOAD uses this data to create the XML DOM object.

```
<html>
   <head>
        ...
   </head>
   <body>
        ...
        <!—in-line XML data>
        <xml id="xmldoc_1">
                <?xml version="1.0" standalone = true?>
                Programmer's Guide
                    <author>Mark Twain</author>
                    <title>Tom Sawyer</title>
                    <price>$11.00</price>
                </book>
                <foo>
                    ...
                </foo>
        </xml>
        ...
   </body>
</html>
```

Data Island—Reference to Another Source

In this code fragment, a reference to XML data found at another location is found in the HTML page. WebLOAD creates an XML DOM object from the source found at that location *only*.

Note: If a Data Island includes an `src=location` statement, then any additional XML data found embedded within the Data Island on the HTML page will be ignored.

```
<html>
   <head>
```

```
        ...
    </head>
    <body>
        ...
        <!—reference to another source>
        <xml id="xmldoc_2" src="http://demo/bookstore.xml">
        </xml>
        ...
    </body>
</html>
```

Data Island—Scripted In-Line

XML DOM objects created from scripts within HTML documents may also be based on either in-line XML data or references to other sources.

For example:

```
<SCRIPT language="xml" id="xmlscript">
    <?xml version="1.0"?>
    <!—XML data in-line within a script element.>
    <bookstore>
        Programmer's Guide
            <author>Mark Twain</author>
            <title>Tom Sawyer</title>
            <price>$11.00</price>
        </book>
        Programmer's Guide
            <author>Oscar Wilde</author>
            <title>The Giant And His Garden</title>
            <price>$8.00</price>
        </book>
    </bookstore>
</SCRIPT>
```

Data Island—Scripted Reference to Another Source

The preceding code illustrates XML data embedded in a script. Scripts may also include references to other sources:

```
<SCRIPT language="xml" id="xmlscript" src="book.xml">
</SCRIPT>
```

Example: Using Data Islands in an Agenda

The following example illustrates Data Island usage. Assume you are working with a Web Bookstore site that includes the following inventory database code fragment:

```
<HTML>
   <HEAD>
        <TITLE> </TITLE>
   </HEAD>
   <BODY>
        <h3>Html file with embedded XML</h3>
        <P>Here begins the XML Data Island</P>
            <xml id="WebStudents">
        <wclass>
        <!-- My students who attended my web programming class -
        ->
        <student id="1">
            <name>Linda Jones</name>
            <legacySkill>Access, VB5.0</legacySkill>
        </student>
        <student id="2">
            <name>Adam Davidson</name>
            <legacySkill>Cobol, MainFrame</legacySkill>
        </student>
        <student id="3">
            <name>Charles Boyer</name>
            <legacySkill>HTML, Photoshop</legacySkill>
        </student>
        <student id="4">
            <name>Charles Mann</name>
            <legacySkill>Cobol, MainFrame</legacySkill>
        </student>
        </wclass>
        </xml>   <P>Here ends the XML Data Island</P>
   </BODY>
</HTML>
```

Working with HTML Properties

When accessing this website, your Agenda may use the standard HTML properties id and innerHTML to print out text strings showing the information found within the XML tags, as follows:

JavaScript Agenda Code:

```
function InitAgenda()
{
  wlGlobals.Parse = true
  wlGlobals.ParseXML = true
}


wlHttp.SaveSource = true


wlHttp.Get("http://www.webloadmpstore.com/xmlsamples/sample2.htm
l")


var XMLstudents = document.wlXmls[0]
InfoMessage("ID : " +XMLstudents.id)
InfoMessage("HTML text : " +XMLstudents.innerHTML)
```

Output Text:

Running this Agenda produces the following output, essentially a text copy of the Data Island fields:

```
ID : WebStudents
HTML text :
  <wclass>
  <!-- My students who attended my web programming class -->
      <student id="1">
            <name>Linda Jones</name>
            <legacySkill>Access, VB5.0</legacySkill>
      </student>
      <student id="2">
            <name>Adam Davidson</name>
            <legacySkill>Cobol, MainFrame</legacySkill>
      </student>
      <student id="3">
            <name>Charles Boyer</name>
            <legacySkill>HTML, Photoshop</legacySkill>
      </student>
      <student id="4">
            <name>Charles Mann</name>
            <legacySkill>Cobol, MainFrame</legacySkill>
      </student>
  </wclass>
```

Working with XML DOM Properties

Your Agenda may also access the complete XML DOM Document Interface for the XML DOM object created from this Data Island. For example, to see a printout of the complete XML DOM structure for the `xmlBookSite` Data Island described on the preceding pages, use the following recursive function. See *Appendix B* in the *WebLOAD JavaScript Reference Guide*, for a description of the XML DOM properties used in this Agenda fragment.

JavaScript Agenda Code:

```
function InitAgenda()
{
   wlGlobals.Parse = true
   wlGlobals.ParseXML = true
}


wlHttp.SaveSource = true


wlHttp.Get("http://www.webloadmpstore.com/xmlsamples/sample2.htm
l")


var XMLstudents = document.wlXmls[0]
var XMLstudentDoc =  XMLstudents.XMLDocument
var XMLstudentElement = XMLstudentDoc.documentElement


InfoMessage("Printing XML under the root")
InfoMessage(XMLstudentElement.xml)
InfoMessage("Element structure of the DOM")
printChildren(XMLstudentElement)


// function that will print all the fields and their attributes
function printChildren(element)
{
   switch(element.nodeTypeString)
   {
        case "element":
             InfoMessage(element.nodeName)
             for(var i=0; i<element.childNodes.length; i++)
             {
                  printChildren(element.childNodes.item(i))
             }
```

```
                break
        case "text":
                InfoMessage("\"" + element.nodeValue + "\"")
                break
    }
}


InfoMessage("End of test")
```

Output Text:

Running this Agenda produces the following output:

```
Printing XML under the root
<wclass>
  <!-- My students who attended my web programming class -->
  <student id="1">
        <name>Linda Jones</name>
        <legacySkill>Access, VB5.0</legacySkill>
  </student>

  <student id="2">
        <name>Adam Davidson</name>
        <legacySkill>Cobol, MainFrame</legacySkill>
  </student>

  <student id="3">
        <name>Charles Boyer</name>
        <legacySkill>HTML, Photoshop</legacySkill>
  </student>

    <student id="4">
        <name>Charles Mann</name>
        <legacySkill>Cobol, MainFrame</legacySkill>
  </student>
</wclass>

Element structure of the DOM
  wclass
  student
  name
  "Linda Jones"
```

```
legacySkill
"Access, VB5.0"
student
name
"Adam Davidson"
legacySkill
"Cobol, MainFrame"
student
name
"Charles Boyer"
legacySkill
"HTML, Photoshop"
student
name
"Charles Mann"
legacySkill
"Cobol, MainFrame"
End of test
```

JavaScript Agenda Code Using the XMLParser Object:

A similar Agenda, with the same functionality, can be written using the XMLParserObject, as shown in the code snippet below:

```
xmlObject = new XMLParserObject();
doc = xmlObject.
parseURI("http://www.webloadmpstore.com/xmlsamples/sample2.html"
)
doc1 = doc.getDocumentElement()
//used to present only the xml
domNode = doc1.getElementsByTagName("xml").item(0);
InfoMessage("Printing XML under the root")
printChildren(domNode)
function printChildren(element)
{
   switch(element.nodeTypeString)
   {
        case "element":
             InfoMessage(element.nodeName)
             for(var i=0; i<element.childNodes.length; i++)
             {
                  printChildren(element.childNodes.item(i))
             }
```

```
                break
        case "text":
                InfoMessage("\"" + element.nodeValue + "\"")
                break
    }
}


InfoMessage("End of test")
```

Changing Bookstore Data

After you have accessed the website and created an XML DOM object from the XML data found on the site, you may wish to change some of the data and post the new version back to the website. To change the author of the first book to 'S. B. David Lee Eddings':

```
function InitAgenda()
{
    wlGlobals.Parse = true
    wlGlobals.ParseXML = true
}


wlHttp.SaveSource = true
wlHttp.Get
("http://www.webloadmpstore.com/xmlsamples/sample1.xml")
//Getting the XML file
var xmlBookListDoc = document.wlXmls[0]
var xmlBookListIsland = xmlBookListDoc.XMLDocument
var xmlBookListElement = xmlBookListIsland.documentElement
InfoMessage("Printing xml under the root: " +
xmlBookListElement.xml)


//First access the node of the first book
var FirstBook = xmlBookListElement.childNodes.item(0)


//Next access the node of the author of the first book
var AuthorFirstBook = FirstBook.childNodes.item(1)


//Finally access the node of the text of the author of the first
book
var TextAuthorFirstBook = AuthorFirstBook.childNodes.item(0)


//In other words, the node in the XML database tree
```

```
//that actually stores the author name value is located at:
//       document.wlXmls[0].XMLDocument.
//       documentElement.childNodes.item(0).childNodes.
//       item(1).childNodes.item(0)
//Now assign a new value
TextAuthorFirstBook.nodeValue = "S. B. David Lee Eddings"

InfoMessage("The changed author value, from J.R.R. Tolkein, to:
"
+ TextAuthorFirstBook.nodeValue)
InfoMessage("Print xml under the root: " +
xmlBookListElement.xml)
```

The changed bookstore database will look as follows:

```
<bookstore name="NODE1NAME">
  <book>
        <title lang="en">Everyday Italian</title>
        <author>David Eddings</author>
        <year>2005</year>
        <price>30.00</price>
  </book>
  <book category="CHILDREN">
        <title lang="en">Harry Potter</title>
        <author>J K. Rowling</author>
        <year>2005</year>
        <price>29.99</price>
  </book>
  <book category="WEB">
        <title lang="en">XQuery Kick Start</title>
        <author>James McGovern</author>
        <author>Per Bothner</author>
        <author>Kurt Cagle</author>
        <author>James Linn</author>
        <author>Vaidyanathan Nagarajan</author>
        <year>2003</year>
        <price>49.99</price>
  </book>
  <book category="WEB">
        <title lang="en">Learning XML</title>
        <author>Erik T. Ray</author>
        <year>2003</year>
```

```
        <price>39.95</price>
    </book>
</bookstore>
```

You can change the bookstore data in the same way by using the XMLParserObject. The Agenda code is even shorter:

```
xmlObject = new XMLParserObject();
doc = xmlObject.
parseURI("http://www.webloadmpstore.com/xmlsamples/sample1.xml")
;
InfoMessage("The xml under the root : " + doc.xml)
doc1 = doc.getDocumentElement()

FirstBook =  doc1.getElementsByTagName("book").item(0);
AuthorFirstBook  = FirstBook.childNodes.item(1) ;
TextAuthorFirstBook  = AuthorFirstBook.childNodes.item(0)
TextAuthorFirstBook.nodeValue = "S. B. David Lee Eddings"
InfoMessage("The changed author value, from J.R.R. Tolkein,
to: " + TextAuthorFirstBook.nodeValue)
InfoMessage("The xml under the root : " + doc.xml)
```

Adding New Bookstore Data

The following Agenda fragment illustrates adding a new 'book element' to the `xmlBooksite` database. See Appendix B in the *WebLOAD JavaScript Reference Guide,* for a description of the XML DOM properties used here.

```
var xmlBookListDoc = document.wlXmls[0]
var xmlBookListIsland = xmlBookListDoc.XMLDocument

//STEP 1, create the new book element
//       and append it to the tree
newBook = xmlBookListIsland.createElement("book")
xmlBookListIsland.documentElement.appendChild(newBook)

//STEP 2, create and name the new book's title element
title1 = xmlBookListIsland.createElement("title")
name = xmlBookListIsland.createTextNode("River God")
//append the new name node to the title node
title1.appendChild(name)
//append the new title node to the book node
newBook.appendChild(title1)
```

```
//STEP 3, create and name the new book's author element
author = xmlBookListIsland.createElement("author")
name = xmlBookListIsland.createTextNode("Wilbur Smith")
author.appendChild(name)
newBook.appendChild(author)

//STEP 4, create and name the new book's year element
year = xmlBookListIsland.createElement("year")
name = xmlBookListIsland.createTextNode("1975")
year.appendChild(name)
newBook.appendChild(year)

//STEP 4, create and name the new book's price element
price = xmlBookListIsland.createElement("price")
amount = xmlBookListIsland.createTextNode("40.55")
price.appendChild(amount)
newBook.appendChild(price)
```

The newly expanded bookstore database will look as follows:

```
<bookstore name="NODE1NAME">
  <book>
        <title lang="en">Everyday Italian</title>
        <author>S. B. David Lee Eddings</author>
        <year>2005</year>
        <price>30.00</price>
  </book>
  <book category="CHILDREN">
        <title lang="en">Harry Potter</title>
        <author>J K. Rowling</author>
        <year>2005</year>
        <price>29.99</price>
  </book>
  <book category="WEB">
        <title lang="en">XQuery Kick Start</title>
        <author>James McGovern</author>
        <author>Per Bothner</author>
        <author>Kurt Cagle</author>
        <author>James Linn</author>
        <author>Vaidyanathan Nagarajan</author>
```

```
        <year>2003</year>
        <price>49.99</price>
    </book>
    <book category="WEB">
        <title lang="en">Learning XML</title>
        <author>Erik T. Ray</author>
        <year>2003</year>
        <price>39.95</price>
    </book>
    <book>
        <title>River God</title>
        <author>Wilbur Smith</author>
        <year>1975</year>
        <price>40.55</price>
    </book></
    </bookstore>
```

You can achieve the same result using XMLParserObject. The Agenda Code is as follows:

```
xmlObject = new XMLParserObject();
doc = xmlObject.
parseURI("http://www.webloadmpstore.com/xmlsamples/sample1.xml")
;
doc1 = doc.getDocumentElement()

newNode1 = doc.createElement("book");
doc1.appendChild(newNode1);

title1 = doc.createElement("title") ;
titleText = doc.createTextNode("River God");
newNode1.appendChild(title1)
title1.appendChild(titleText )

author = doc.createElement("author") ;
authorName =  doc.createTextNode("Wilbur Smith");
newNode1.appendChild(author)
author.appendChild(authorName )

year = doc.createElement("year") ;
yearInd =  doc.createTextNode("1975");
newNode1.appendChild(year )
```

```
year.appendChild(yearInd )


price = doc.createElement("price") ;
priceAmount =  doc.createTextNode("40.55");
newNode1.appendChild(price)
price.appendChild(priceAmount )
```

Creating and Filling New XML DOM Objects

WebLOAD supports the creation of new XML DOM objects through the
`WLXmlDocument()` constructor. New XML DOM objects may either be created:

- Empty, with the user adding data as needed.

- Already loaded with data found in existing XML files or text strings.

New XML DOM objects created by the `WLXmlDocument()` constructor are not
produced from HTML documents, so they do not include the HTML property set (`id`,
`innerHTML`, and `src`). HTML properties have no meaning for XML DOM objects
created without any connection to an HTML document.

XML DOM objects created with the `WLXmlDocument()` constructor include the
complete XML DOM Document Interface described in *Appendix B* in the *WebLOAD
JavaScript Reference Guide*. Users may also load and reload XML data into XML DOM
objects through the `load()` and `loadXML()` methods. This section describes the
different methods of creating and filling data into XML DOM objects.

WLXmlDocument(xmlStr)—Creating XML DOM Object from XML String

The `WLXmlDocument(xmlStr)` constructor accepts an optional parameter, an XML
string that includes the entire XML document data. For example:

```
NewXMLObj = new WLXmlDocument(xmlStr)
```

Use this form of the `WLXmlDocument()` constructor to create new XML DOM objects
from an XML string parameter rather than relying on Data Islands from an HTML
page.

Working with XML in Native (Direct) Browsing Mode

WebLOAD automatically processes Data Islands found on HTML documents.
WebLOAD also provides access to pure XML documents. For example, assume a
website uses Native Browsing rather than having XML data explicitly embedded in
Data Islands. If you wish to work with this website you could issue a Get transaction to
access the XML page after enabling the `SaveSource` property to save the page source
to a file. Once the XML string has been saved in the data file produced by the Get

transaction, you may then use `WLXmlDocument()` to create a new XML DOM object from that XML data string. You now have a fully functional XML DOM object, with a Document Interface identical to that of XML DOM objects created from Data Islands. You may use this XML DOM object to manipulate the data, assign or change data values, and post the changed data back to the server, exactly as you would when working with a Data Island.

The following Agenda fragments illustrates this sequence:

1. Set `SaveSource` so that the downloaded XML data will be saved to a file that can later be passed to the `WLXmlDocument()` constructor:

   ```
   wlHttp.SaveSource=true
   ```

2. Get the XML page, either as a static page from the server:

   ```
   wlHttp.Get("http://demosite/bookstore.xml")
   ```
   Or through a database query to the bookstore:

   ```
   wlHttp.Get("http://demosite/bookstore.exe?
           author=Mark Twain&MaxPrice=$20.00")
   ```

3. Create a new XML DOM object using the saved page source, which happens to include the XML string:

   ```
   newXmlObj = new WLXmlDocument(document.wlSource)
   ```

4. At this point you have an XML DOM object which may be manipulated in any way just as you would manipulate XML DOM objects produced by Data Islands. You may make any changes or additions you wish to the XML DOM object data, as described in previous examples illustrating changing or adding bookstore data.

   For example, you may post the new XML DOM data back to the server:

   ```
   wlHttp.Data.Type="text/xml"
   wlHttp.Data.Value=xmlobj.xml
   wlHttp.Post("Http://demosite/bookstore.exe?operation=upd
   ate")
   ```

Note: The WLXmlDocument(xmlStr) constructor must be passed complete, self-contained XML strings *only*. The DTD section must not contain any external references when using this form of the constructor. See *Document Type Definition (DTD)* (on page 92), for more information.

WLXmlDocument()—Creating a New, Blank XML DOM Object

The `WLXmlDocument()` constructor may be used without any parameters. In this case, a new, blank XML DOM object will be created. For example:

```
NewBlankXMLObj = new WLXmlDocument()
```

You may now use the `loadXML()` method to add XML data to your new blank XML DOM object.

```
NewBlankXMLObj = loadXML("<?xml version='1.0'?>
    <bookstore></bookstore>")
```

To add more content to the document, create elements and add them as child nodes as described in the example in the previous section.

Loading XML Files into XML Objects

The MSXML Document Interface provides two methods for loading XML documents into XML DOM objects:

* `loadXML("XMLdocumentstring")`
* `load("URL")`

This section describes the advantages and limitations of the `loadXML()` and `load()` methods when used in WebLOAD IDE Agendas, and discusses how to select the method that will be most effective in your Agendas.

Note: You may use `loadXML()` and `load()` repeatedly to load and reload XML data into XML DOM objects. Remember that each new 'load' into an XML DOM object will overwrite any earlier data stored in that object.

Using loadXML(XMLDocString)

The `loadXML(XMLDocString)` method accepts a literal XML document in string format as its only parameter. This allows users to work with XML documents and data that did not originate in HTML Data Islands, such as with Native Browsing. In a typical scenario, a user downloads an XML document. WebLOAD saves the document contents in string form. The string is then used as the parameter for `loadXML()`. The information is loaded automatically into an XML object.

For example:

```
// Create a new XML document object
NewXMLObj = new WLXmlDocument()
wlHttp.SaveSource = true
wlHttp.Get(http://www.server.com/xmls/doc.xml)
XMLDocStr = document.wlSource
// Load the new object with XML data from the saved source.
// We are assuming no external references, as explained below
NewXMLObj.loadXML(XMLDocStr)
```

Note: Creating a new, blank XML DOM object with `WLXmlDocument()` and then loading it with an XML string using `loadXML()` is essentially equivalent to creating a new XML DOM object and loading it immediately using `WLXmlDocument(xmlStr)`. As with the `WLXmlDocument(xmlStr)` constructor, only standalone, self-contained DTD strings may be used for the `loadXML()` parameter. External references in the DTD section are not allowed.

Using load("URL")

The `load("URL")` method accepts a URL or filename where the XML document may be found as its only parameter. `load()` relies on the MSXML parser to handle any Get transactions needed to download the XML document. The XML data is then loaded automatically into the XML object.

For example:

```
myXMLDoc = document.wlXmls[0]
myXMLdoc.load(http://server/xmls/file.xml")
```

When you use the `load()` method in your Agenda, the MSXML module performs all the underlying HTTP transactions. External references in the DTD section are not allowed when using `load()`. However, the MSXML module accesses external servers and completes all necessary transactions without any control or even knowledge on the part of the WebLOAD system tester. From WebLOAD's perspective, these transactions are never performed in the context of the test session. For this reason, any settings that the user enters through the WebLOAD IDE or Console will not be relayed to the MSXML module and will have no effect on the document 'load'. For the same reason, the results of any transactions completed this way will not be included in the WebLOAD statistics reports.

Comparing loadXML() and load()

WebLOAD supports both the `load()` and the `loadXML()` methods to provide the user with maximum flexibility. The following table summarizes the advantages and disadvantages of each method:

Table 5: load() and loadXML() comparison

	Advantages	**Disadvantages**
loadXML()	Parameters that the user has defined through WebLOAD for the testing session will be applied to this transaction.	The method fails if the DTD section of the XML document string includes any external references.

	Advantages	Disadvantages
load()	The user may load XML files that include external references in the DTD section.	Parameters that the user has defined through WebLOAD for the testing session will not be applied to this transaction. WebLOAD does not record the HTTP Get operation. (See note below.) The transaction results are not included in the session statistics report. Using this method may adversely affect the test session results.

Note: If you wish to measure the time it took to load the XML document using the `load()` method, create a timer whose results will appear in the WebLOAD IDE statistics. For example:

```
myXMLDoc = document.wlXmls[0]
SetTimer("GetXML")
myXMLdoc.load("http://server/xmls/file.xml")
SendTimer("GetXML")
```

Example: Building an XML Database from Scratch

The next example will put together some of the pieces from the previous sections. Here we will create a new, skeletal XML DOM object for a bookstore database, build new book elements, and add each new book to our bookstore database.

```
// 1. CREATE AN EMPTY XML OBJECT with a skeleton:
xmlBookstoreDoc = new WLXmlDocument
   ("<?xml version='1.0'?><bookstore></bookstore>")
// 2. ADD CONTENT TO THE DOCUMENT:
//create the first book element
newBook = xmlBookstoreDoc.createElement("book")
//append the new book to the bookstore tree
xmlBookstoreDoc.documentElement.appendChild(newBook)
//create, name, and append the new book's author element
author = xmlBookstoreDoc.createElement("author")
name = xmlBookstoreDoc.createTextNode("Mark Twain")
author.appendChild(name)
newBook.appendChild(author)
//create, name, and append the new book's title element
title = xmlBookstoreDoc.createElement("title")
name = xmlBookstoreDoc.createTextNode("Tom Sawyer")
```

```
title.appendChild(name)
newBook.appendChild(title)
//create, name, and append the new book's price element
price = xmlBookstoreDoc.createElement("price")
amount = xmlBookstoreDoc.createTextNode("$12.00")
price.appendChild(amount)
newBook.appendChild(price)
// 3. CONTINUE TO ADD CONTENT TO THE DOCUMENT:
//create the second element and append it to the tree
newBook = xmlBookstoreDoc.createElement("book")
xmlBookstoreDoc.documentElement.appendChild(newBook)
//create, name, and append author, title, and price elements
author = xmlBookstoreDoc.createElement("author")
name = xmlBookstoreDoc.createTextNode("Leo Tolstoy")
author.appendChild(name)
newBook.appendChild(author)
title = xmlBookstoreDoc.createElement("title")
name = xmlBookstoreDoc.createTextNode("War and Peace")
title.appendChild(name)
newBook.appendChild(title)
price = xmlBookstoreDoc.createElement("price"
amount = xmlBookstoreDoc.createTextNode("$20.00")
price.appendChild(amount)
newBook.appendChild(price)
```

A new XML DOM bookstore database object has now been created and filled with information about two books. The product of the `xmlBookstoreDoc.documentElement.xml` property would be:

```
<?xml version="1.0"?>
<bookstore>
    <book>
        <author>Mark Twain</author>
        <title>Tom Sawyer</title>
        <price>$12.00</price>
    </book>
    <book>
        <author>Leo Tolstoy</author>
        <title>War and Peace</title>
        <price>$20.00</price>
    </book>
```

```
</bookstore>
```

Handling Web Service Transactions

WebLOAD supports working with Web services, including manipulating the Web service's responses.

A Web service response is a soap-formatted XML message. Accessing this XML message becomes possible, after parsing the response using WebLOAD's XMLParserObject. This enables you to locate the node containing the desired part of the web service response.

The following sample Agenda demonstrates how WebLOAD handles web service transactions by extracting the dynamic XML from the web service response and parsing it using the XMLParserObject. The desired return value is then located by searching through the parsed XML.

```
xmlObject = new XMLParserObject()

wlGlobals.GetFrames = false
wlHttp.Get("http://www.webloadmpstore.com/ajaxsample")

wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/ajaxsample/"
wlHttp.FormData["wsdl"] = "$WL$VOID$STRING$"
wlHttp.Get("http://www.webloadmpstore.com/ajaxsample/serveradd1.
php")

wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/ajaxsample/"
wlHttp.Data["Type"] = "text/xml; charset=utf-8"
wlHttp.Data["Value"] = "<?xml version=\"1.0\" encoding=\"utf-
8\"?><soap:Envelope
xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"
xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\"><soap:B
ody><add
xmlns=\"urn:addwsdl\"><a>20</a><b>30</b></add></soap:Body></soap
:Envelope>"

wlHttp.Post("http://www.webloadmpstore.com/ajaxsample/serveradd1
.php")

//Extract the XML part from the response
doc = document.wlXmls[0]
InfoMessage (doc.xml)
```

```
//Load the extracted XML into the XMLParserObject
xo = xmlObject.loadXML(doc.xml)


//Locate the node containing the Web Service result according to
the name of the node: "return"
node = xo.getElementsByTagName("return").item(0)
InfoMessage(node.xml)
```

Document Type Definition (DTD)

The term DTD is an acronym for Document Type Definition. The Document Type Definition is a set of rules that describes the grammar of a particular XML document. The DTD is used to validate an XML document.

For example, the DTD for a bookstore database would do the following:

1. *Define the structure* and attributes of that database.

    Based on the rules found in the DTD, you might have a database that requires every book entry to include information on the book's title, author, and price. Information on publishers and reprint requests may be declared optional.

2. After the DTD defines the structure and attributes of the database, the XML Data Island *assigns values* to the attributes listed in the DTD.

3. Once an XML document has been downloaded, the Browser *validates the document* by checking that the XML data follows the rules set by the DTD.

    If the price is missing from a certain book item, for example, and a price is required by the DTD, then the Browser will not be able to validate that XML document.

Note: For the Browser to parse and validate an XML document according to a particular DTD, the author must include a `<DOCTYPE>` section in the XML document. WebLOAD IDE supports DTD verification only if the DTD is completely included in the Data Island and there are *no external references* in the DTD.

The following code illustrates a self contained DTD for the bookstore example used throughout this section of the manual:

```
<!-DOCUMENT STARTS HERE->
<?xml version="1.0"?>
<!-THE DTD STARTS HERE -->
<!DOCTYPE
[
   <!ELEMENT books-table (book)+>
```

```
<!ELEMENT book (title,author+,isbn,publication-date,book-
type,book-family+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT isbn (#PCDATA)>
<!ELEMENT publication-date (#PCDATA)>
<!ELEMENT book-type (#PCDATA)>
<!ELEMENT book-family (#PCDATA)>
<!--attribute list-->
      <!ATTLIST book id ID #REQUIRED>
      <!ATTLIST author rank CDATA #IMPLIED>
]>
<!--THE DTD ENDS HERE-->
<!--DATA PART OF THE DOCUMENT->
  <books-table>
      <book id="thisbook">
          <title></title>
          <author rank=""></author>
          <isbn></isbn>
          <publication-date></publication-date>
          <book-type></book-type>
          <book-family></book-family>
      </book>
  </books-table>
<!-DOCUMENT ENDS HERE->
```

Working with Java

Java is an object-oriented, dynamic, platform-independent programming language developed by Sun Microsystems in the mid-1990s. Java provides its users with a complete run-time environment, a Virtual Machine (VM). With the Java VM sitting in your Web browser, any HTML document can include embedded Java, adding active content to your Web page. Java includes a variety of useful features, such as Remote Method Invocation (RMI), which allows local developers to seamlessly access distributed application functions and procedures, and Java Database Connectivity (JDBC), which allows Java clients to interact with any ODBC- or JDBC-compliant database, whether it resides locally or on a remote server. By utilizing Java tools and classes, user applications are able to offer far greater power and capabilities than when working with simple HTML code alone. For this reason, over the past few years, Java has become one of the most popular development mediums for large-scale Internet applications.

WebLOAD supports full Java access from your JavaScript Agendas. Full Java support means that your WebLOAD Agendas will not only test access time to an HTML page. WebLOAD Agendas can also invoke and run local Java classes used by the Java applications embedded within an HTML page. The Java components that stand behind that Web page need no longer be considered 'black boxes'. Your JavaScript Agenda can directly access and test the classes of each locally residing Java component, allowing you to test each aspect of your application.

WebLOAD uses LiveConnect3, developed by Netscape Communications Corporation and maintained and supported by the Mozilla Foundation, to allow your JavaScript Agendas to communicate with Java classes. LiveConnect is an invisible architecture that runs in the background. Your JavaScript code may work directly with Java. Your WebLOAD Agenda can even include both JavaScript code that calls Java code and Java code that calls JavaScript code. The interface between Java and JavaScript is managed for you, allowing you to create the Java/JavaScript testing scenario that best meets your needs.

For a detailed specification of the syntax and classes used by LiveConnect, see *LiveConnect Overview* (on page 189). For more information about the interaction between Java and JavaScript using LiveConnect, as well as complete documentation for Java, JavaScript, and LiveConnect, including the latest features and enhancements, go to the following websites:

- https://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Guide:LiveConnect_Overview

  The basic manual for JavaScript, containing an excellent LiveConnect overview and syntax summary.

- https://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference:LiveConnect

  Core JavaScript reference manual index. LiveConnect Class Reference, for a comprehensive explanation of the `JSException` and `JSObject` classes.

- https://www.mozilla.org/js/liveconnect/lc3_method_overloading.html

  Provides a clear, complete explanation of type conversions and method overloading.

- https://developer.mozilla.org/en/docs/LiveConnect

  The main LiveConnect page on the Mozilla Developer Center.

Requirements

This section provides a brief explanation of how to work with Java components directly from your JavaScript Agenda. Sample Agendas illustrate typical usage.

JDK/JRE 1.5 (or Higher)

Install the standard Sun Microsystems Java Virtual Machine (JVM), version 1.5 or higher, on your system.

To verify that you are working with version 1.5 or higher:

1. Select **Control Panel** from the Windows **Start ➤ Settings** menu and double click the **Add/Remove Programs** icon.

2. Check the items listed in the dialog box. Entries that reference Java should be labeled version 1.5 or higher.

3. If you wish to update your version of Java, download the latest version from http://java.sun.com (http://java.sun.com).

   Follow the standard Sun installation instructions. It makes no difference if you install the JVM before or after installing WebLOAD.

If you are doing any development work that may require application compilation, you must install the complete Java Developer's Kit (JDK). If your application is complete and you will only be running it, without making any changes or corrections that may require compilation, then the Java Runtime Environment (JRE) will be enough to simply run a test session. The decision to install only the JRE, or the complete JDK, depends on your own application's requirements. WebLOAD IDE provides full Java support in both cases.

PATH Environment Variable

Your command path must include both the Java bin and Java classic directories. The Java bin directory is where the Java programs and compiler are located. The Java classic directory is where the `jvm.dll` file is located.

For example, after a typical installation of JDK version 1.5, your PATH should include something similar to this:

```
PATH=%PATH%;C:\Program Files\jdk1.5\bin;
  C:\Program Files\jdk1.5\jre\bin\classic
```

CLASSPATH Environment Variable

Your CLASSPATH environment variable must point to the following:

* JVM class libraries

* LiveConnect class libraries

* Your application's class libraries (a series of `*.class` and `*.jar` files)

For example, assuming your Java class libraries are located in the `Java\lib` directory, the LiveConnect class libraries are located in `LiveConnect\classes`, and your application class libraries are located in the `AppDev\lib` directory, your CLASSPATH should include the following:

```
CLASSPATH=%CLASSPATH%;C:\Java\lib\rt.jar;
   C:\LiveConnect\classes;C:\AppDev\lib
```

In general, following a WebLOAD IDE installation, the LiveConnect class libraries are located in:

```
C:\Program Files\RadView\WebLOAD\LiveConnect\classes
```

Setting PATH and CLASSPATH Environment Variables

Set the PATH and CLASSPATH environment variables to include the necessary directories. The settings can be modified through Windows or a DOS command window.

To verify that both your PATH and CLASSPATH environment variables include the necessary directories:

1. From the Windows taskbar, select **Start ➤ Settings ➤ Control Panel**.

2. Double click the **System** icon and select the **Environment** tab.

3. Check the Environment Variables to confirm that your environment variables are set correctly.

Alternatively, you could open a DOS command window:

- Type `path`, and press <Enter>.

   Your current PATH setting will appear in the window.

   If you type `set classpath` and press <Enter>, your current CLASSPATH setting will appear in the window.

Remember that the exact full directory path name will differ from system to system. Check with your system administrator to verify the correct Java directory path names for your system.

Alternatively, you could do the following:

1. Right-click the **My Computer** icon on your desktop.

2. Select **find…**, and search for `jvm.dll`.

This file is usually found in a directory path that ends with `…\bin\classic`. This directory, and the same directory without the `\classic` ending, should both appear in the PATH environment variable.

If your PATH or CLASSPATH environment variables do not include the necessary directories do the following:

1. Add the correct directory names for your system to the PATH or CLASSPATH environment variable definitions in the Environment tab of the System dialog box.

2. Click **Set**.

3. Click **Apply**, to reset the variables to the new settings.

   You do not have to reboot your system. But you do have to restart TestTalk for WebLOAD to recognize the new PATH environment variable definition.

Public Methods

JavaScript Agendas adhere to the same rules of object-oriented programming (OOP) as any other program. All programs must respect the OOP concepts of public and private. As in any OOP language, Java class methods may be either public or private. Public methods are functions that are externally visible and can be called by other Java classes and by JavaScript. For JavaScript to be able to control a Java class, the Java class must provide public methods.

Identifying Java Objects in JavaScript Agendas

Java uses its own specific terminology. Java objects are referred to as *classes*. These classes are organized hierarchically within *packages*. To access Java objects, specify the full path to that class, using the format:

```
<java package full path>.<java class>
```

For example, to reference the Java class `Lassie` contained in the package `dogs`, which is contained within the package `animals`, write `animals.dogs.Lassie`.

Accessing Java Objects from JavaScript Agendas

Java objects found in JavaScript Agendas are accessed according to the rules listed here. Detailed explanations and examples illustrating these rules are provided in the Netscape websites listed in *Working with Java* (on page 93).

• To access a built-in Java object, use the package `java`. This package acts as a flag, informing the compiler that it is about to start work with a built-in Java object. You may then continue to work directly with that object, its properties, and methods.

   For example:

   ```
   var myJavaString = new java.lang.String("Hello world")
   ...
   stringlen = myJavaString.length()
   ```

…

-Or-

```
s = new java.net.Socket ("12.3.5.56",23)

…

s.getInputStream

…
```

In these examples, `java.lang` and `java.net` are the full package names and `String` and `Socket` are the class names.

- To access a *locally defined* or third party Java object that is not part of the Java, Sun, or Netscape packages, use the reserved Java keyword `Packages`. This keyword acts as a flag, informing the compiler that it is about to start work with a locally-defined Java object. You may then continue to work directly with that object, its properties, and methods.

  For example:

  ```
  var myJavaDog = new Packages.JavaDog(Lassie)

  …

  dogbreed = myJavaDog.dogBreed

  …
  ```

Use the `Packages` keyword to access any user-defined class whose definition is included in the CLASSPATH library list, described in *Requirements* (on page 94).

Forestalling Errors

Most common compilation and runtime errors can be prevented if you follow the instructions in this guide. Usually, you won't have any problems as long as you verify that Java was installed correctly and specify Java objects with the correct full path and argument list. This section lists the most common potential pitfalls, in an effort to prevent problems before they occur.

- You must have Java version 1.2 installed on your system. Working with earlier versions of Java will cause your Agendas to fail with the following WebLOAD IDE error message:

  ```
  WebLOAD can't find jvm.dll
  ```

  If you receive this error message, you may have an older version of the JVM, or you may not have Java installed at all on your system. If this is the problem, download and install the newer version of Java from http://java.sun.com.

- Java version 1.2 or higher must be installed and environment variables defined on each Load Generator included in your test session. Test sessions that involve multiple Load Generators require a resident JVM with the correct environment variable settings on each Load Generator.

- Whether to install only the JRE or the complete JDK depends on your application's requirements. If you will be doing any development or compilation, install the complete JDK. Trying to compile Java class files (such as `JavaClass.java`) on a computer that does not have the JDK installed will trigger an error. The JDK must be installed on *each* computer on which Java files will be compiled. The JRE can not handle any compilation requests.

- Each system environment is slightly different. When setting the PATH or CLASSPATH environment variables for your system, substitute the actual path to the required Java directories and libraries on your system. Use the Environment tab in the System dialog box from the Windows Control Panel to set the environment variables.

- Environment variables must be set with the correct values before starting TestTalk or running any Load Generators. Make sure you first verify and set the environment variables through the Windows Control Panel and only then start TestTalk.

- If you do not have the correct Java class directories included in your PATH, your Agendas may fail with the following WebLOAD error message:

    ```
    WebLOAD can't find jvm.dll
    ```
  If you receive this error message and your version of the JVM is 1.2 or higher, check your system PATH (through the Windows Control Panel) to be sure it points to the `jvm.dll` library. Make sure the directories listed are in fact the correct directories, and verify the correct path name in your Agenda code.

- The class path and name in your Agenda files must be specified absolutely correctly, with all elements spelled correctly and with the correct case. For example, the class name `MyJavaClass` is not the same as **my**`JavaClass`.

- The directories included in the CLASSPATH environment variable must also be specified absolutely correctly, with all elements spelled correctly, to exactly match the class path and names that appear in your JavaScript Agendas.

  For example, assume you are working with an application that includes the Java class `MyJavaClass`, whose constructor expects a single string argument, located in the directory `I:\AppFiles\libs`.

  Your CLASSPATH environment variable should include:

    ```
    CLASSPATH=%CLASSPATH%;I:\AppFiles\libs
    ```
  Your Agenda should reference this class as follows:

    ```
    var myJavaObj = new Packages.MyJavaClass("stringArg")
    ```
  If you were not careful, you could accidentally define CLASSPATH to include the wrong directory, such as:

    ```
    CLASSPATH=%CLASSPATH%;I:\AppFiles\orig-libs
    ```

Or you could accidentally include an incorrect package path in the Agenda file, such as:

```
var myJavaObj = new
Packages.Jlib.MyJavaClass("stringArg")
```

In either of these cases the constructor call would fail and WebLOAD would return the following error message:

```
TypeError: <classname> is not a constructor
```

- Make sure to pass the correct number and types of arguments to Java class constructors or methods.

Realize that working with Java from your JavaScript Agenda is really very simple and intuitive, as illustrated in the simple examples found in the remainder of this section. You simply must take care to specify objects correctly. Careless spelling mistakes will trigger errors, as they would in most programming languages.

Example: passing simple variables between Java and JavaScript

You may pass any legal JavaScript variable or object as a parameter to a Java object or method. Conversions between basic variable types, such as integer, string, or Boolean, will be completed automatically. Return values will also convert correctly. Conversions between Java and JavaScript types are based on the conversion tables listed in *Data Type Conversions* (on page 197).

The following example illustrates passing basic values between Java and JavaScript objects. The Java class method expects two parameters, a string and an integer, and returns the concatenation of both parameters into a single string. Conversions between Java and JavaScript strings and integers are completed automatically.

Java side:

```
public class SimpleExample
{
   public String Concat(String a, int b)
   {
       return a + " : " + b;
   }
}
```

JavaScript side:

```
a = new Packages.SimpleExample()
InfoMessage(a.Concat("RoundNum",RoundNum))
```

Passing Objects Between Java and JavaScript

LiveConnect uses the reserved Java class `JSObject` to pass a JavaScript object to Java objects. `JSObject` tells the Java side that it is being sent a JavaScript object. To pass a JavaScript object as a parameter to a Java method, your Java file must include the following:

• Import the `netscape.javascript` package.

• Define the corresponding formal parameter of the method to be of type `JSObject`.

You may set and change a JavaScript object property value, and even add a new member to a JavaScript object, while working from the Java side.

For example, the following code illustrates passing values between parallel Java and JavaScript objects.

Java Side:

```
import netscape.javascript.JSObject;
public class JavaDog
{
  public String dogBreed;
  public String dogColor;
  public String dogSex;
  public JavaDog(JSObject jsDog)
  {
        this.dogBreed = (String)jsdog.getMember("breed");
        this.dogColor = (String)jsdog.getMember("color");
        this.dogSex = (String)jsdog.getMember("sex");
  }
}
```

Note: The `getMember` method of `JSObject` is used to access the properties of the JavaScript object. This example uses `getMember` to assign the value of the JavaScript property `jsDog.breed` to the Java data member `JavaDog.dogBreed`.

JavaScript Side:

To continue with this example, look at the following definition of the JavaScript object `Dog`:

```
function Dog(breed, color, sex)
{
  this.breed = breed
  this.color = color
  this.sex = sex
```

```
}
```
Create an instance of the **JavaScript object Dog** as follows:

```
Lassie = new Dog("collie", "chocolate", "female")
```
The JavaScript property `Lassie.color` here has been assigned the value `chocolate`.

Now create an instance of the **Java object JavaDog** in the JavaScript code by passing the JavaScript `Lassie` object to the `JavaDog` constructor as follows:

```
JavaDog = new Packages.JavaDog(Lassie)
```

Note: The JavaScript code uses the `Packages` class to identify `JavaDog` as a locally defined Java object, as described in *Identifying Java Objects in JavaScript Agendas* (on page 97).

The Java property `javaDog.dogColor` has the value `chocolate` because the `getMember` method in the Java constructor assigns the value of `Lassie.color` to `dogColor`.

Detailed explanations of the `JSObject` class are provided in http://developer.netscape.com/docs/manuals/js/core/jsref/lcjsobj.htm.

A sample Agenda illustrating joint Java and JavaScript activity is found in *Calling a WebLOAD API from a Java Application* (on page 104).

Automatic Timers And Counters For Java Objects

WebLOAD provides automatic timers and counters that wrap around every Java class method invocation found in your Agenda. Automatic timers and counters allow you to effortlessly measure and quantify the behavior and response time of each specific Java component individually, enabling the most comprehensive testing of your Java-based application.

Every time your JavaScript Agenda calls a Java class method, a timer and a counter are automatically created for that method. The results are included in the Statistics Report for that test session. These automatic Java timers are in addition to the standard WebLOAD Timer functions, which may be called from the Java side of your Agenda as well as the standard JavaScript side. An example illustrating JavaScript Timer calls from Java code appears in *Calling a WebLOAD API from a Java Application* (on page 104).

For example, assume you are communicating with a server using the Java socket class:

```
try
{
  // Connect to a server through Java socket
  s = new java.net.Socket ("12.3.5.56",23)
```

```
    // Create Java I/O streams
    from_server = new java.io.DataInputStream(s.getInputStream())

    to_server = new java.io.PrintStream(s.getOutputStream())

    //Read line from Input stream to JavaScript string value
    line = from_server.readLine()
    InfoMessage("" + line)

    // Write JavaScript string value to the Output stream
    to_server.println ("Client " + ClientNum + "-" + RoundNum )

    // Get answer from the server via the Input stream
    line = from_server.readLine()
    InfoMessage("" + line)

}

catch (Exception) {
    ErrorMessage ("Server is not connected")
}

finally
{
    try {
        if (s != null)
        s.close();
    }
    catch(Exception) {}
}
```

Each time you call any of the Java class methods, a timer and counter for that method is automatically incremented. The results appear in the Statistics Report. Each timer and counter is uniquely identified with the Java class, object, and method name, as illustrated in the following figure:

Measurements	Total	socket.Generator1@dudu
Round Time	0.402	0.402
Rounds	25.000	25.000
Successful Rounds	25.000	25.000
java.io.DataInputStream::readLine (success)	50.000	50.000
java.io.DataInputStream::readLine (Timer)	0.179	0.179
java.io.PrintStream::println (success)	25.000	25.000
java.io.PrintStream::println (Timer)	0.000	0.000
java.net.Socket::close (success)	25.000	25.000
java.net.Socket::close (Timer)	0.001	0.001
java.net.Socket::getInputStream (success)	25.000	25.000
java.net.Socket::getInputStream (Timer)	0.000	0.000
java.net.Socket::getOutputStream (success)	25.000	25.000
java.net.Socket::getOutputStream (Timer)	0.001	0.001

Double click on a column or row header or any table cell to view statistical details.

WebLoad D... 　 Statistics

Figure 22: Timer and Counter Identification with Java Class, Object, and Method Name

The counter and timer information is collected and displayed automatically, every time you call a Java class method from your Agenda. You do not have to add anything to your Agenda code to take advantage of this feature.

Calling a WebLOAD API from a Java Application

WebLOAD utilizes the locally understood, generic 'this' object, passing the current 'this' object as a parameter to a Java method. The JavaScript concept of 'this' object as your current working object is preserved even while working with that object's properties and methods from the Java side.

Using 'this' object, WebLOAD is able to call WebLOAD functions directly from within Java method code, passing to the Java functions the default 'this' object that is expected by the function.

For example:

Use the WebLOAD SetTimer and SendTimer functions to time Java activities.

Use InfoMessage to print messages to the log window.

The following Agenda illustrates calling WebLOAD functions from Java code.

Java side:

```
import netscape.javascript.JSObject;
// The JSObject class must be imported for the
// code to successfully compile
public class MyJClass
{
   public void TimersFromJava(JSObject myJavaScriptObj)
   {
        String args[] = {"Timer1"};
        myJavaScriptObj.call("SetTimer",args);
        …<do any other work> …;
        myJavaScriptObj.call("SendTimer",args);
   }
}
```

Note: You can call the JavaScript `SetTimer` and `SendTimer` functions directly from the Java class using the `JSObject.call()` method. Pass parameters to the functions being called (in this case the timer name expected by the timer functions), using the `args` array.

JavaScript side:

```
// The following JavaScript code uses the Java class
// defined in the preceding Java section.
myJavaObject = new Packages.MyJClass()
// The JavaScript code uses the Packages class to
// identify MyJClass as a locally defined Java object
try
{
   myJavaObject.TimersFromJava(this)
}
catch (e)
{
   SevereErrorMessage("Error : " + e);
}
```

Note: You work with `myJavaObject` as you would with any JavaScript object. `TimersFromJava` is called as a simple object method, passing the locally understood 'this' object as the parameter to the Java method.

The `try...catch` statement in the JavaScript code marks a block of statements to try. The `catch` block specifies the response the program should supply should an exception be thrown. If an exception is thrown, the `try...catch` statement catches it. WebLOAD recommends wrapping the JavaScript calls to Java functions within `try...catch` statements, to add robustness and error recovery to your code.

Example: reading data from a JDBC database

The following example illustrates a very common website activity—accessing a database. The Java side of the example is a straightforward Java program to access a JDBC, including methods to:

- Load a driver
- Open a connection
- Create and execute SQL queries

The JavaScript side of the example illustrates exactly how simple it is to work with the Java program from a JavaScript Agenda.

Java side:

```
import java.sql.*;
public class jdbcExample
{
   Connection conn;
   Statement st1;
   ResultSet rs;
   public static void loadDriver(String js_driver)
            throws ClassNotFoundException
   {
       Class.forName(js_driver);
   }


   public void getConnection(String js_url,
       String js_login, String js_pswd)
       throws SQLException
   {
       //get the connection & also create a statement:
       conn = DriverManager.getConnection
           (js_url, js_login, js_pswd);
       st1 = conn.createStatement();
   }

   public String executeQuery(String js_query)
       throws SQLException
   {
       //execute sql statement:
       rs = st1.executeQuery(js_query);
       // loop over the result set.
```

```
        String strOnSuccess = "3. Result Set is:";
        while (rs.next())
        {
        strOnSuccess = strOnSuccess + "\n" +
              rs.getString("ReportName");
        }
        // return the data
        return strOnSuccess;
    }
}
```

JavaScript side:

The JavaScript side of this test session Agenda runs a JDBC load test which checks the ability of the database to handle a large number of SQL queries.

Note: The following is the sequence of JDBC activity in this Agenda:
1. The JDBC driver is loaded only once for the test session, in the `InitAgenda()` function.
2. A JDBC connection is created once for each thread, in the `InitClient()` function.
3. Queries are created in the main body of the Agenda. Answers are sent to the Console.

```
function InitAgenda()
{
   try
   {
        // load the JDBC driver once per session
        jdbcDriver = "<driver class name>"
        Packages.jdbcExample.loadDriver(jdbcDriver);
   }
   catch (e)
   {
        SevereErrorMessage("Error : " + e);
   }
}
function InitClient()
{
   // create a separate jdbcObj object and connection
   // for each thread
   jdbcObj = new Packages.jdbcExample()
   // connect to the database
   url = "<jdbc:driver-name:host>"
   login = "<username>"
```

```
        password = "<password>"
        try
        {
                jdbcObj.getConnection(url, login, password)
        }
        catch (e)
        {
                SevereErrorMessage("Error : " + e);
        }
}
//Main body of Agenda
try
{
    // create query and send the results to the Console
    query = "SELECT * FROM Reports WHERE ReportId < 10 "
    query_result = jdbcObj.executeQuery(query)
    InfoMessage(""+ query_result)
}
catch (e)
{
    WarningMessage("Error : " + e);
}
```

Working with the Component Object Model (COM)

What is COM

Microsoft's Component Object Model (COM) provides a way for distinct software components to communicate with each other. COM technology provides universally reusable binary components. These components can be mixed and matched over versions and years, accessed by applications written in any of a variety of languages, running on any of a variety of platforms, located either locally or over a network. COM automation simplifies application development by maximizing component reusability while increasing the application's universality. Today, most Windows users rely on COM technology, often without even realizing it. For example, suppose you wish to work with a restricted website. Access to the application behind that website is controlled by a security database. To work with that application, you must first enter your name and password. A COM component will automatically access the database and verify your permission status.

COM components are objects that consist of a combination of properties, methods, and interfaces. An object's interface is simply a set of methods together with a defined set

of standards for what those methods do and how they are accessed, their parameters and return values. WebLOAD JavaScript provides direct object access to any component that has a COM wrapping and an `IDispatch` interface. These are known as ActiveX objects. See *ActiveX Object Interfaces* (on page 109), for more information.

An ActiveX object is viewed and manipulated exactly as any other JavaScript object within the Agenda. WebLOAD encapsulates COM automation functionality, providing an interface between JavaScript Agendas and ActiveX objects. For example, a WebLOAD JavaScript Agenda is able to fully test an ASP Web page or a Web page that manipulates an ADO database. In both of these technologies, ActiveX objects are widely used.

This section of the guide explains how to work with ActiveX objects within your JavaScript Agenda. For a detailed explanation of COM programming, see the Microsoft MSDN Online Library at http://msdn.microsoft.com (http://msdn.microsoft.com).

Note: If you are working with COM objects that are not thread-safe and can not handle multi-threading, you must set the multithreading to one thread per process (this is WebLOAD's default setting).

To set the multithreading:

1. In the WebLOAD Console, open the **Tools ➤ Default/Current Session/Agenda Options** dialog box.

2. Select the **Browser Parameters** tab.

3. In the Multi-thread Virtual Clients->Load Generator field, enter or scroll to 1, which sets one thread per process.

ActiveX Object Interfaces

ActiveX objects usually include the following interfaces:

- `IUnknown`—the core interface. Defines the ActiveX object. Includes the definitive ActiveX object methods QueryInterface, AddRef, and Release.

- `IDispatch`—the access interface. Enables ActiveX object automation, allowing the WebLOAD IDE JavaScript Agenda to manipulate the ActiveX object's properties and methods.

- Custom Interfaces—additional, specialized interfaces. Created by the user, specific to each ActiveX object.

  JavaScript Agendas access the methods and properties of the object interface exposed by the IDispatch interface.

- `ITypeInfo`—the index interface. Includes a list of all objects, properties, and methods for each interface included in the ActiveX object (optional).

The `ITypeInfo` interface of an ActiveX object provides access to the `TypeInfo` library for that object.

Note: The `ITypeInfo` interface is optional. WebLOAD is able to work with ActiveX objects whether or not they supply a `TypeInfo` library file. However, access to the `TypeInfo` library for an object saves overhead. Programs that can take information from the `TypeInfo` library do not have to spend time and energy analyzing an object to identify a specific item's characteristics. For example, access to the `TypeInfo` library eliminates the need to figure out a variable's data type or insert casting functions into your Agendas for greater security.

Activating ActiveX Objects from a JavaScript Agenda

A typical website is often the front end of an application that includes many ActiveX components. WebLOAD Agendas enable thorough testing of both the website and the applications that are accessed via that website by allowing you to activate an application's ActiveX object from your Agenda. For example, to use your WebLOAD Agenda to directly test access time to an ADO database, you would activate the ActiveX objects for that database.

To use ActiveX objects in your Agenda do the following(as you would with any JavaScript object):

1. Create a reference to a new object instance.

2. Assign and get values for the object's properties.

3. Execute the object's methods.

This section describes how to activate ActiveX objects from your WebLOAD JavaScript Agenda.

Note: The object you are accessing must already exist and be registered. When working with a remote server, through DCOM over HTTP, the object you are accessing should reside on the remote server. (This is described in *DCOM over HTTP* (on page 117)). When working locally, the COM object you are accessing should reside on the Load Generator.

The following section describes the `ActiveXObject` function syntax.

ActiveXObject() (constructor)

Method of Object

- ActiveX

Description

Creates a new ActiveX object. The new object is simply a local JavaScript object, and may be handled and manipulated like any other JavaScript object.

Syntax

```
my_ActiveXobject = new
    ActiveXObject("ApplicationName.ObjectName" [, "rServer"])
```
In VBScript terminology, the syntax appears as follows:

```
my_ActiveXobject = new ActiveXObject("ServerName.TypeName")
```

Parameters

`ApplicationName`—The name of the application providing the object.

`ObjectName`—The type or class of the object being created. When working with local COM objects, the object resides on the local Load Generator.

`rServer`—The name of the remote server being accessed. Optional, used when working through DCOM, to access objects residing on the remote server.

Return Value

A pointer to the new ActiveX object. The new object is simply a local JavaScript object, and may be handled and manipulated like any other JavaScript object.

Example

To create a new Excel spreadsheet:

```
ExcelSheet = new ActiveXObject ("Excel.Sheet")
```

Assigning Values to ActiveX Objects

At this point, now that you have created a reference to an ActiveX object, the fact that it refers to an ActiveX object does not affect usage and syntax within your JavaScript Agenda. Once the new object has been activated, `my_ActiveXobject` is simply a local JavaScript object that is used to communicate with a COM object. Your Agenda works with this object exactly as it would work with any other JavaScript object. You do not have to deal with any COM overhead or syntax issues. You access the new object's properties and methods as you would access any other JavaScript object.

This section describes how to work with local ActiveX objects in your WebLOAD JavaScript Agenda. For an example illustrating working with ActiveX objects on a remote server, see *DCOM over HTTP* (on page 117).

Work with your ActiveX objects properties and methods as you would with any JavaScript object's properties and methods, using the following syntax:

```
my_ActiveXobject.method(method-parameters)
orig_property_value = my_ActiveXobject.property
my_ActiveXobject.property = new_property_value
index_property_value =
    my_ActiveXobject.indexproperty(indexvalue)
```

The following JavaScript examples illustrate this use:

```
// To explicitly create two new object instances
mother = new ActiveXObject ("Family.MotherObject")
father = new ActiveXObject ("Family.FatherObject")
//mother and father are local JavaScript objects
// To assign values directly to my object's 4 properties
mother.Name = "Jane"
mother.Age = 21
mother.BirthDate = "Jan 20, 1978"
mother.Smoke = false
// To assign values to my object's 4 properties through an
// object passed as a parameter to my object's SetInfo method
father_birthday = new Date("Mar 20,1975")
father.SetInfo("John",25,father_birthday,true)
// To implicitly create a new child object
child = mother.MakeChild(father)
// To assign values to my child object's 4 properties through
// a combination of direct property assignment and
// method execution
child.Name = "Patrick"
color = child.GetEyesColor()
child.DemandCare(mother,father)
```

Timers and Counters for ActiveX Objects

WebLOAD provides automatic timers and counters for ActiveX objects. Automatic timers and counters allow you to effortlessly measure and quantify the behavior and response time of each specific ActiveX component individually, enabling the most comprehensive testing of your COM-based application.

Every time your Agenda calls an ActiveX method, a timer and a counter are automatically created for that method. The results are included in the Statistics Report for that test session.

For example, assume you are working on a project about animal behavior, using the following ActiveX object:

```
animal = new ActiveXObject ("Animals.Dog")
animal.Name = "Bingo"
animal.Drink("Water")
animal.Eat("Meat","Fish")
```

Each time you call any of the animal object's methods, a timer and counter for that method is automatically incremented. The results appear in the Statistics Report. Each timer and counter is uniquely identified with the ActiveX application, object, and method name, as illustrated in the following figure:

Measurements	Total	animal_agenda.Generator1@vadimx
Load Size	12.111	12.111
Rounds Per Second	1.700	1.700
Successful Rounds Per Second	1.700	1.700
Round Time	11.171	11.171
Rounds	34.000	34.000
Successful Rounds	34.000	34.000
Animals.Dog::Drink (Counter)	33.000	33.000
Animals.Dog::Drink (Timer)	2.393	2.393
Animals.Dog::Eat (Counter)	34.000	34.000
Animals.Dog::Eat (Timer)	3.538	3.538
Load Size Delta	-	-

Figure 23: Timer and Counter Identification with ActiveX Application, Object and Method Name

The counter and timer information is collected and displayed automatically, every time you call an ActiveX method from your Agenda. You do not have to add anything to your Agenda code to take advantage of this feature.

Note: There are no counters for calls to ActiveX object constructors.

Automatic Conversion between JavaScript and COM Data Types

The examples found in the preceding section illustrate how simple it is to refer to ActiveX objects within your JavaScript Agenda. You do not need to know or declare anything special about an object's definition or typing when working within your

Agenda. The fact that JavaScript and COM use slightly different data types is not an issue. WebLOAD simply supports all standard JavaScript and COM data types.

WebLOAD JavaScript Agendas automatically convert between JavaScript and the corresponding COM data types. The table below illustrates the data type conversions that are supported. The default choices used if no data type information is available are marked in **bold** in the table.

Working with JavaScript and COM data types within a JavaScript Agenda is simple. Passing values from JavaScript Agendas as parameters to COM objects can be more complicated. COM does not expect, nor does COM know how to convert from, a JavaScript data type. WebLOAD is responsible for smoothing the interface between JavaScript and COM, converting between JavaScript and the corresponding COM data types when necessary.

When the data type is known, WebLOAD automatically converts between JavaScript data types and the corresponding COM data type. If the data type is unknown, WebLOAD checks to see if a `TypeInfo` library is available for the object. (`TypeInfo` is usually available.) When available, WebLOAD takes the data type information from the `TypeInfo` library and completes the conversion as usual.

If `TypeInfo` information is not accessible, WebLOAD uses a basic common-sense heuristic to determine the data type and select the appropriate conversion, based on the preceding default conversion table. For example, if a variable A has been assigned a value of 5, WebLOAD assumes that the variable should be of type `Integer`.

However, relying on sensible assumptions may inadvertently lead to complications. For example, the user may actually intend to pass that variable A as a parameter to a method that expects a `String` "5". Or you may be working with an array containing a whole set of variables of unknown data types, a more complicated situation. For these reasons, WebLOAD recommends using casting functions when the data type is unknown, to ensure that the variables are converted to the correct data type before being passed as parameters to an ActiveX object method. The WebLOAD casting functions are described in the following section, *Using Casting Functions for JavaScript and COM Data Types* (on page 115).

Table 6: JavaScript-COM supported data type conversions

JavaScript Data Type ®	Integer	Double	Boolean	String	Date	ActiveX Object	Array
COM Data Type							
Byte	Yes	Yes	Yes	-	-	-	-
Short	Yes	Yes	Yes	-	-	-	-
Long	Yes	Yes	Yes	-	-	-	-

JavaScript Data Type ®	Integer	Double	Boolean	String	Date	ActiveX Object	Array
Float	Yes	Yes	Yes	-	-	-	-
Double	Yes	Yes	Yes	-	-	-	-
VARIANT_ BOOL	Yes	Yes	Yes	-	-	-	-
Date	-	-	-	-	Yes	-	-
BSTR	Yes	Yes	Yes	Yes	-	-	-
IUnknown	-	-	-	-	-	Yes	-
IDispatch	-	-	-	-	-	Yes	-
SAFEARRAY	-	-	-	-	-	-	Yes

(Arrays of a specific type. See discussion of array data types in *Using Casting Functions for JavaScript and COM Data Types* on page 115.)

| VARIANT | Yes | Yes | Yes | Yes | Yes | Yes | - |

Using Casting Functions for JavaScript and COM Data Types

WebLOAD recommends using casting functions under the following circumstances:

- When no `TypeInfo` library is available.

- When you know you are passing mismatched data types.

- If you receive a `TypeError` message from COM.

Casting functions ensure that JavaScript variables are converted to the correct data type before being passed as parameters to an ActiveX object method. To explain why casting functions are recommended, this section focuses on how JavaScript and COM work with arrays.

Arrays are collections or sets of variables. The values stored within a JavaScript array may either all be of the same data type, (i.e., all integers or all strings), or they may consist of a variety of different data types. COM includes a `SafeArray` option, asserting that all the items within the array are of the same type. For maximum flexibility, `SafeArrays` also provide the option of all items being of type `Variant`. While each item is officially of the 'same type', this actually means that each item may be of *any* `Variant`– compatible data type.

While there are very few restrictions on the kinds of data types acceptable by ActiveX objects for array parameters, the one requirement is that whatever is passed must be of the correct (expected) data type. If an ActiveX object method expects to receive an array of type `Integer`, it must be passed an array of type `Integer` and not an array of type `Variant`, even if all the array elements do happen to be integers. When

passing a JavaScript array object as a parameter to an ActiveX object method, the items of that array must be converted correctly to the corresponding ActiveX object data types or the method will fail.

If nothing is known about the data type of an array's elements, WebLOAD tries to choose the most logical data type for that array. For example, if all the values in the array appear to be of the same data type, WebLOAD will pass an array of that data type. If the values in the array appear to be of different data types, WebLOAD will pass an array of type `Variant`. While this approach will almost always work, it may occasionally fail. For example, in the rare event that an array of type `Variant` coincidentally contains only items that happen to all be of the same `Integer` data type, WebLOAD will analyze the elements of that array, conclude that it must be of type `Integer`, and convert it accordingly to an ActiveX array object of type `Integer`. This may be a logical decision, but the method will fail.

To avoid this small possibility of failure during your testing session, WebLOAD provides a complete set of casting functions. While casting functions may be used at any time, explicitly setting a data type is usually recommended only when no `TypeInfo` library is available, when you know you are passing mismatched data types, or if you receive a `TypeError` message from COM. (COM errors, which appear as standard error messages on the WebLOAD Console, are usually triggered by an error in the application being tested, for example, by an error in the ActiveX object's `TypeInfo` library.)

WebLOAD provides the following casting functions:

- `CByte()`
- `CInt()`
- `CLng()`
- `CDbl()`
- `CFlt()`
- `CBool()`
- `CVARIANT()`

These functions take a variable as a parameter and return that value cast to the specified data type. Only legal data type conversions, as listed the table in *Automatic Conversion between JavaScript and COM Data Types* (on page 113), are allowed.

For example, assume you had an array that should be passed as data type Variant, but it only holds items with integer values. To prevent problems, use the `CVARIANT()` function to explicitly cast the array elements to the `Variant` data type, as follows:

```
My_variant_arr = new Array()
My_variant_arr[0] = CVARIANT(1)
My_variant_arr[1] = CVARIANT(21)
```

```
My_variant_arr[2] = CVARIANT(32)
My_variant_arr[4] = CVARIANT(44)
Result = RemoteCOMobject.Add(My_variant_arr)
```

Remember, if your array contains elements with a variety of data types, you don't have to explicitly cast the array elements to force the array to be converted to data type `Variant`. WebLOAD will understand that this is a `Variant` array based on the variety of data types found. However, you may always use a casting function if you wish:

```
My_variant_arr = new Array()
My_variant_arr[0] = CVARIANT(1)
My_variant_arr[1] = CVARIANT(21.1)
My_variant_arr[2] = CVARIANT("John")
My_variant_arr[4] = CVARIANT(new Date(1999,1,1))
Result = RemoteCOMobject.Concat(My_variant_arr)
```

In another example, assume you had an array that should be passed as data type `Double`. The elements were assigned a variety of `Integer` and `Double` data type values. There is no `TypeInfo` library available. WebLOAD, seeing a combination of integer and double values, will assume that this is an array of data type `Variant`. However, the method is expecting an array of data type `Double`. To prevent problems, use the `CDbl()` function to explicitly cast the integer array elements to the `Double` data type, as follows:

```
My_double_arr = new Array()
My_double_arr[0] = 100.1
My_double_arr[1] = CDbl(21)
My_double_arr[2] = 22.2
Result = RemoteCOMobject.AddDoubles(My_double_arr)
```

DCOM over HTTP

In today's work environment, applications are often distributed over a network. In the course of your testing session, you may need to activate an ActiveX object that is located on a remote server and accessed through a network. For this reason, WebLOAD supports ActiveX object access through Remote Data Service (RDS).

Remote object access is accomplished in two steps: First, declare a new **RDS object**. That RDS object is used as a bridge, pointing to the ActiveX object residing on the remote server. Then, activate an **ActiveX object** located on a remote server, use the **initial instantiation syntax** described in *Remote ActiveXObject() Constructor* (on page 118).

Note: WebLOAD assumes a basic familiarity with RDS use, including object declaration and initialization, and system configuration. For more information about Microsoft's Remote Data Services of ADO (RDS), go to the following websites:

For a general overview, see the Microsoft MSDN Online Library at http://msdn.microsoft.com.

For a more complete example that illustrates RDS use, see http://support.microsoft.com/kb/q184630/.

To learn how to configure `RDS.DataSpace` to create a Custom Business Object (ActiveX DLL) on either NT or Win2000, see http://support.microsoft.com/kb/q185169/.

To learn how to configure RDS for Windows 2003, see http://support.microsoft.com/kb/837981/en-us.

Remote ActiveXObject() Constructor

Method of Object

- ActiveX

Description

Creates a new remote ActiveX object. The new object is simply a local JavaScript object, and may be handled and manipulated like any other JavaScript object.

Syntax

```
RDS_object = new ActiveXObject ("RDS.DataSpace")
my_RemoteActiveXobject =
  RDS_object.CreateObject("AppName.ObjName", "http://rServer")
```

Parameters

`RDS.DataSpace`—Creates an RDS object to act as a 'bridge' to an ActiveX object residing on the remote server.

`ApplicationName`—The name of the application providing the object.

`ObjectName`—The type or class of the object being created.

`rServer`—The name of the remote server being accessed.

Return Value

A pointer to the new ActiveX object. The new object is simply a local JavaScript object, and may be handled and manipulated like any other JavaScript object.

Comments

When working with a remote server, the object you are accessing must reside on the Web server computer (in this example: http://rserver).

Once your new object is instantiated, it is simply a local JavaScript object, just like any other JavaScript object. You may manipulate the object as you would any other JavaScript object. An example of remote ActiveX object instantiation and use is illustrated in the next section.

Example: Remote ActiveX Object Access

The following Agenda fragment illustrates ActiveX object access from a WebLOAD JavaScript Agenda. This example retrieves a recordset from a database, updates the recordset, and then makes the necessary changes to the database.

```
// Instantiate a new ActiveX object
DataSpace = new ActiveXObject("RDS.Dataspace")
// Invoke server object. localhost is the server here
svrObject = DataSpace.CreateObject
   ("RDSServer.DataFactory", "http://localhost")
// svrObject is now a local JavaScript object used to
// access the remote ActiveX object. svrObject is
// manipulated exactly as any other JavaScript object.
// Create output file with the unique ClientNum included
// as part of the name for identification purposes
wlLocals.MyFileObj = new wlOutputFile
        ("C:\\OutputFile" + ClientNum + ".txt")
// svrObject.Query returns a recordset
strRecord = svrObject.Query
        ("DSN=AdvWorks2", "Select* from Customers")
// Initialize database record counter
n_recordCounter = 0
// The objField2 variable is used for debugging,
// to verify the correct field and check its value.
objField2 = strRecord.Fields("Threads")
// Notice that it is not necessary to explicitly state:
// objField2 = strRecord.Fields.item("Threads")
// because WebLOAD supports default fields
// The Agenda works in a While-loop that
// verifies that I am not at the last record.
// If I am, then the update code does not run
while(strRecord.EOF == false)
```

```
{
    // Access the current field value
    strCurrentFieldValue = strRecord.Fields(0).Value
    // Add five to current record value and
    // verify that I am talking to the dataSource
    // so I can keep track of where I am in the field
    if (strCurrentFieldValue == 5)
        strCurrentFieldValue += 5
    // Update just saves the field.dot value
    objField2.Value = strCurrentFieldValue
    objField2.Value.Update
    // More debug coded when necessary.
    txtCurrent = strRecord.Fields(0).Value
    // Submit changes to the record source and
    // move to the next record
    svrObject.SubmitChanges("DSN=AdvWorks2", strRecord)
    strRecord.MoveNext()
}
// End of While-loop
```

COM Error Management

Failure of a COM function call always triggers a WebLOAD error message, whether or not the COM call was enclosed in a `try...catch` block. This gives the Agenda programmer an opportunity to handle application errors within the context of the WebLOAD test session Agenda.

ActiveX Object Limitations

Due to the nature of ActiveX object implementation, WebLOAD ActiveX support is subject to certain limitations. The following items are not supported:

* Events. The ActiveX object is not able to make a call to the client.

  For example, work with the `IConnectionPoint` interface is not supported.

* Setting a property value within a function parameter.

  For example, you cannot write:

  ```
  ReturnValue = FunctionCall(My_Object.Property=5))
  ```

* Passing method calls that return values as parameters to another method call of the same object.

  For example, the following method call will fail:

```
AU = new ActiveXObject ("Persits.AspUser")
CurrentUser = AU.GetUser(AU.GetUserName())
```

Instead, either call the method using a second, temporary object, such as this:

```
AU = new ActiveXObject ("Persits.AspUser")
AUTemp = new ActiveXObject ("Persits.AspUser")
CurrentUser = AU.GetUser(AUTemp.GetUserName())
```

Or use a method and a property rather than two methods, such as this:

```
AU = new ActiveXObject ("Persits.AspUser")
CurrentUser = AU.GetUser(AU.UserNameProp)
```

- Assigning values to indexed properties.

   For example, you cannot assign a value as follows:

```
My_object.property(1) = 5
```

- Calling an indexed property without parentheses.

   For example, the following call to the indexed property `Environment` will fail:

```
WshShell = new ActiveXObject("WScript.Shell")
env = WshShell.Environment
```

Instead, use the following syntax:

```
WshShell = new ActiveXObject("WScript.Shell")
env = WshShell.Environment("System")
```

Chapter 4

Working with HTTP Protocol

Website load testing usually means testing how typical user activities are handled by the system under test not in a single-user scenario but rather under heavy usage load. Using load testing, we can test the system's performance, scalability and reliability, all in real-life simulated usage scenarios and patterns. A system's performance is all about how fast it functions; its reliability is about how often it's available (where a system might be unavailable due to a lack of a certain resource or because of a bug); and a system's scalability is about how these two factors (performance and reliability) change as the usage of the system increases.

WebLOAD was designed as a protocol-level load testing tool, enabling the QA professional to "bombard" the system under test with protocol-level commands and transactions, simulating real usage of a large amount of users. The main and most important protocol for load testing Internet applications is the HTTP protocol. WebLOAD supports the common HTTP methods and headers, as defined in the HTTP RFCs, most notably RFC 2616 (1999), that was defined by the W3C and the IETF.The WebLOAD DOM extension set includes objects, methods, properties, and functions that support designing tests at the HTTP protocol level. Using the DOM, functional and verification tests can be done, to check the system's reliability under load.

This chapter documents the features that apply to QA professionals creating Agendas based on the HTTP Protocol, where HTTP transactions and the responses they trigger are the focus of the test session. This chapter includes a few sample Agendas that you can study to help learn the HTTP Protocol testing technique. The examples are fictitious, but you can copy the Agendas and edit them for use in real WebLOAD tests. More sample Agendas may be found on the WebLOAD Script Libraries, at http://www.webload.org/index.php?option=com_wrapper&Itemid=160.

Understanding the WebLOAD DOM Structure

For optimum website testing, WebLOAD extended the standard browser DOM with many features and functions that are critical to site testing and evaluation. The following figure, an extended version of the basic DOM hierarchy tree, highlights many of the added elements of the WebLOAD DOM hierarchy. The remainder of this chapter provides more detailed information about some of the special elements and features that WebLOAD added to the basic DOM model that are used when working in HTTP Protocol mode.
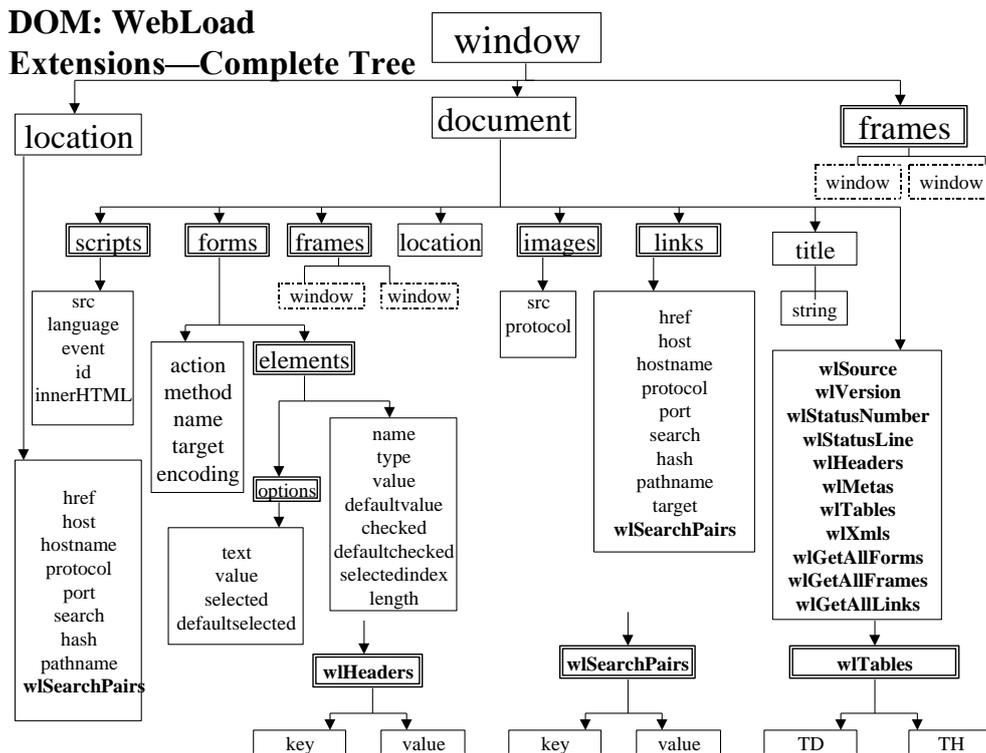
Figure 24: WebLOAD DOM Hierarchy

The preceding illustration highlights in bold the WebLOAD extensions used by WebLOAD when working with the Document Object Model.

Note: WebLOAD-specific objects are identified by the `wl` prefix.

The following table lists some of the extensions that WebLOAD has added to the standard DOM objects, properties, and methods.

Table 7: WebLOAD DOM Extension Set Highlights

WebLOAD extensions	Description
Objects	
wlCookie	Sets and deletes cookies.
wlException	WebLOAD error management object.
wlGeneratorGlobal and wlSystemGlobal objects	Handles global values shared between Agenda threads or Load Generators.
wlGlobals	Manages global system and configuration values.
wlHeader	Contains the key/value pairs in the HTTP command headers that brought the document. (Get, Post, etc.)
wlHtml	Retrieves parsed elements of HTTP header fields.
wlHttp	Performs HTTP transactions and stores configuration property values for individual transactions.
wlLocals	Stores local configuration property values.
wlMeta	Stores the parsed data for an HTML meta object.
wlOutputFile	Writes Agenda output messages to a global output file.
wlRand	Generates random numbers.
wlSearchPair	Contains the key/value pairs in a document's URL search strings.
wlTable, row, and cell objects	Contains the parsed data from an HTML table.
XML DOM objects	XML DOM object set that both accesses XML site information *and* generates new XML data to send back to the server for processing.
Properties	
wlSource	Contains the complete HTML source code of the frame, in a read-only string.
wlStatusLine	Contains the status line of the HTTP header, in a read-only string.
wlStatusNumber	Contains the HTTP status value, which WebLOAD retrieves from the HTTP header, in a read-only integer.
wlVersion	Contains the HTTP protocol version, which WebLOAD retrieves from the HTTP header, in a read-only string.
Methods	
wlGetAllForms	Retrieves a collection of all forms (<FORM> elements) in an HTML page and its nested frames.
wlGetAllFrames	Retrieves a collection of all frames in an HTML page, at any level of nesting.
wlGetAllLinks	Retrieves a collection of all links (<A> elements) in an HTML page and its nested frames.

Using Multiple IP Addresses

WebLOAD enables the creation of a single test script that includes multiple IP addresses, simulating the behavior of actual users.

Note: Before you enable support of multiple IP addresses, you must first generate additional IP addresses on your machine to use when testing. For more information, see *Generating IP Addresses in the Agenda* on page 126.

Enable the use of all available IP addresses through the property `wlGlobals.MultiIPSupport`. The values of `MultiIPSupport` are:

- false – Use only one IP address. (default)
- true – Use all available IP addresses.

Indicate whether WebLOAD should use the same IP or a different IP for every round, by setting the value of the `wlGlobals.MultiIPSupportType` property which supports the following values:

- PerClient – WebLOAD preserves the current behavior, meaning that there are different IPs per client but the same IP is used for all rounds. This is the default setting.
- PerRound – WebLOAD supports the use of a different IP from the pool per client, per round, until the pool is exhausted, after which it returns to the beginning.

The exact number of IPs that WebLOAD supports depends on the operating system being used:

- With Unix, WebLOAD uses the `gethostbyname()` function to access the IP address. There is no limit to the number of IP addresses that WebLOAD supports.
- With Windows, WebLOAD loads an SNMP agent dll file, through which WebLOAD accesses the IP information. The SNMP agent exists on all Windows versions since Win95 and NT, even if an SNMP agent is not installed. This method queries all of the IPs, one at a time, and places the result in a list maintained in the WebLOAD's engine code. WebLOAD continues to go over the entire IP list, until it reaches the engine code's limit of 100,000 IPs.

  If for some reason the SNMP does not work, WebLOAD uses another method whose limitation is 35 IPs.

Generating IP Addresses in the Agenda

After enabling the use of all available IP addresses, you can generate additional IP addresses on your machine by setting the TCP/IP properties through the Windows UI. For further instructions consult your system administrator. Alternatively, you can use the `FillIP.bat` file provided with WebLOAD, as described below.

To generate IP addresses in the Agenda:

1. Locate the `FillIP.bat` file in `<WebLOAD directory>\bin`.

2. Run the batch file by entering the file name in the command line or double-clicking the file.

   The command line window appears.
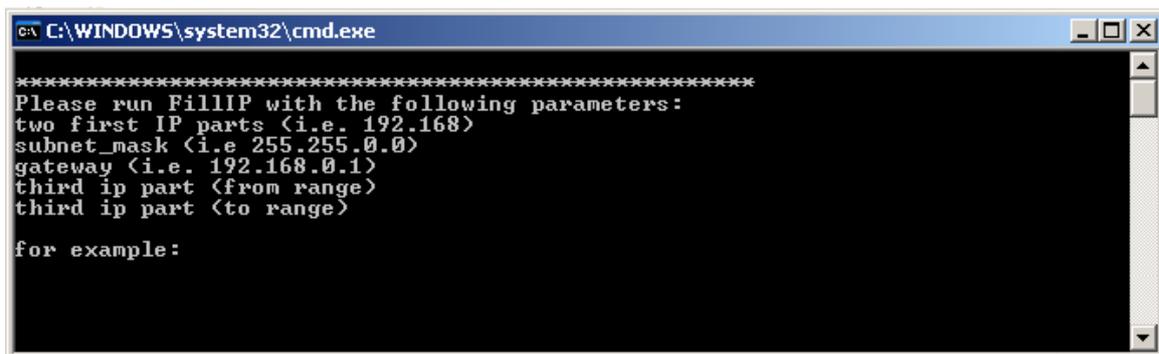
Figure 25: Command Line Window

3. Enter the `netsh` command to generate the list of IP addresses.

   `FillIP IP_prefix subnet_mask gateway from to`

   The `netsh` command parameters are described in the following table:

Table 8: FillIP Command Parameters

Parameter	Description
IP_prefix	The prefix used for all of the generated IP addresses. For example, if `IP_prefix` is `192.168` then the generated IP address structure is `192.168.xxx.xxx`.
subnet_mask	The `subnet_mask` is matched with the IP address to determine the fourth section of the generated IP address, along with the `gateway` value. For example `255.255.0.0`.
gateway	The `gateway` used for all of the generated IP addresses to determine the fourth section of the generated IP address, along with the `subnet_mask` value. For example, `192.168.0.1`.
from to	Used to generate the third section in the IP address. For example, if `from to` is `1 24`, then the generated IP addresses are all from `xxx.xxx.1.xxx` to `xxx.xxx.24.xxx`.

Upon successful completion, the Command Line window returns `OK`. If an error is encountered (for example, an illegal IP address), the Command Line window returns details about the error.

For example, to generate approximately 6000 IP addresses, use the following command in the Agenda:

```
FillIP 192.168 255.255.0.0 192.168.0.1 1 24
```

The following IP addresses are generated:

192.168.1.1	192.168.2.1	…	192.168.24.1
192.168.1.2	192.168.2.2	…	192.168.24.2
…			
192.168.1.255	192.168.2.255	…	192.168.24.255

Parsing Web Pages

Web testing sessions work with Web pages, by accessing a specified page and verifying that all operations are completed correctly. As part of the Web page access, downloaded Web pages are parsed and the data from each page stored in a logical structure. This section gives a brief overview of the parsing approach to a Web page.

A Typical Web Page and the Corresponding Parse Tree

The following is an example of a typical Web page, including three child windows with their own nested frames, forms, and links:
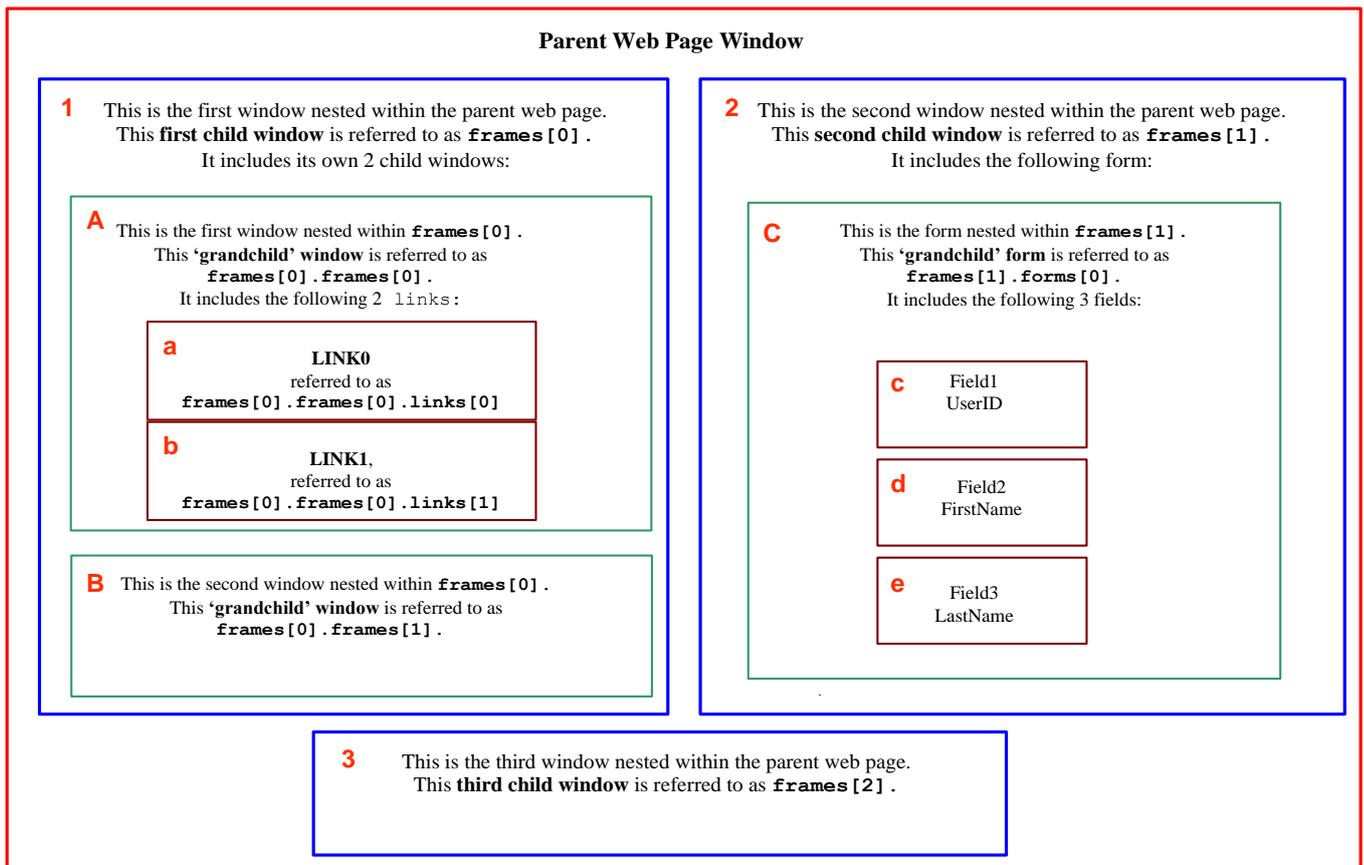
Figure 26: Typical Web Page

The figure above illustrates a URL that includes a single document with three child windows (**1**, **2**, and **3**).

The *first child window* (**1**) includes two children of its own, here referred to as 'grandchildren' (**A** and **B**).

The *first 'grandchild window'* (**A**) includes two links (**a** and **b**).

The *second child window* (**2**) contains a single form, here referred to as a 'grandchild form' (**C**). This *grandchild form* includes three element fields (**c**, **d**, and **e**).

The frames and links are identified by index numbers (0, 1, 2, 3,...). The form fields are identified by name ("UserId", "FirstName", etc.).

The preceding Web page relies on a hierarchical window structure that corresponds to the following virtual tree:
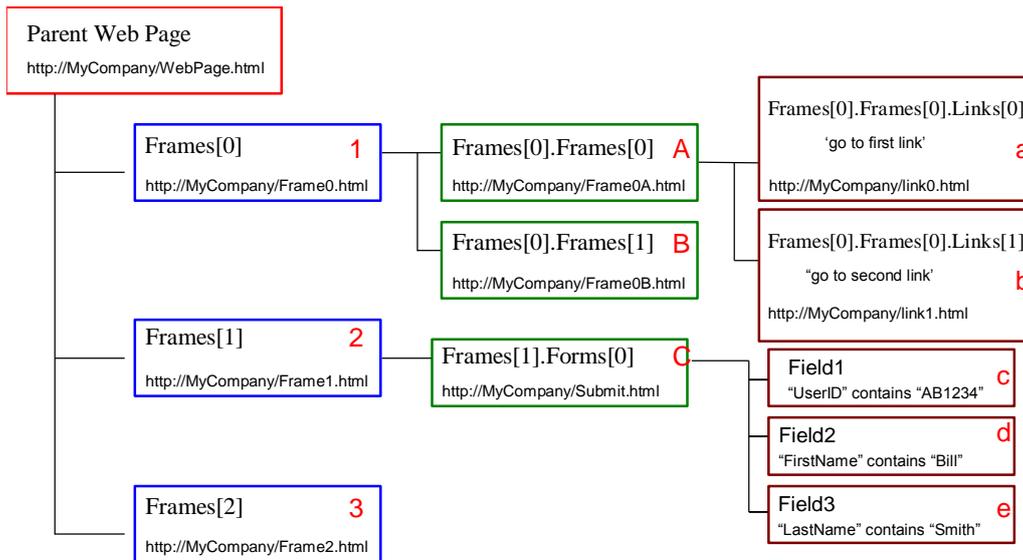
Figure 27: Web Page Virtual Hierarchy Tree

In the `document` object property description, the examples given will refer to the Web page illustrated here.

Parsing and Navigating Nested Frames on a Dynamic HTML Page

The following Agenda downloads a dynamic HTML page belonging to a retailer that advertises on the Internet. The second frame on the page offers bargain prices on three different items each hour. The Agenda performs the following operations, some of which are marked in bold in the Agenda code:

- Downloads the page including the nested frames.

- Retrieves the second frame.

- Searches the HTML source code of the second frame to find the <TITLE> element.

- Confirms that the title is "Offer of the Hour". (If it is not, the Agenda displays an error message and stops the current round of the thread where the error occurred. The thread continues with the next round.)

- Randomly selects one of the first three links in the second frame.

- Follows the link.

Note: This Agenda includes DOM objects that are not explained in this section. For more information about a specific object or usage, (such as `wlGlobals`, `wlRand`, or `wlHttp`), see that object's section in the *WebLOAD JavaScript Reference Guide*.

```
function InitAgenda()
{
//Enable parsing of all links
   wlGlobals.ParseLinks = true
   //Enable downloading nested frames
   wlGlobals.GetFrames =  true
}
function InitClient()
{
//Initialize the random number generator with a
   //different seed for each thread (ensures that each
   //thread follows a different sequence of links)
   wlRand.Seed(ClientNum)
}
//Main Agenda Body
//Store the most recent download in document.wlSource
wlHttp.SaveSource = true
//Download the dynamic HTML page
wlHttp.Get("http://www.webloadmpstore.com/general_sample/frames/
frames.htm")
//Retrieve the second frame
MyFrame = document.frames[1]
InfoMessage(MyFrame)
InfoMessage(MyFrame.document.wlSource)


//Confirm that the <TITLE> element of the frame is
//"Offer of the Hour". If it is not, display an error
//message and abort the current round.
//Remember to escape the slash in </TITLE>


// Using extractValue to catch the title
myTitle = extractValue( "<title>", "<\/title>",
MyFrame.document.wlSource)


// will print the content of the title
InfoMessage (myTitle )


if (myTitle  != "offer of the hour")
{
   ErrorMessage("Thread " + ClientNum.toString() + ", Round " +
   RoundNum.toString() + ": Title is not 'Offer of the Hour' but
   is " + myTitle )
```

```
}
//Retrieve the URL of a randomly selected link
SelectedItem = wlRand.Select(0, 1, 2)
LinkUrl = MyFrame.document.links[SelectedItem].href
InfoMessage(LinkUrl )
//Follow the link
wlHttp.Get(LinkUrl)
```

Using wlHtml to Follow a Dynamic Link

The third link on a company's home page is an advertisement that links to one of several products. On successive accesses, the advertisement switches dynamically among the products.

The following WebLOAD Agenda tests the time that a user would need to follow the link.

The Agenda:

- Links to the home page and parses the HTML code.

- Retrieves the URL of the third link.

- Measures the time it takes WebLOAD to:

  - Download the main page and, depending on the parameters, do any of the following:

    - Download frames, images, tables, or whatever else has been specified.

    - Download nested subframes, images, etc.

    - Parse the document.

In this Agenda, WebLOAD displays two time statistics:

- The Round Time.

- The Link Time.

```
function InitAgenda()
  //Set the default URL to the dynamic HTML home page
  wlGlobals.Url = "http://www.ABCDEF.com"
  //Parse links. Forms are not needed.
  wlGlobals.ParseLinks = true
  wlGlobals.ParseForms = false
}
//Main script
//Connect to the home page
wlHttp.Get()
```

```
// Retrieve the third link on the home page using the manual
// GetLinkByIndex() method.
// See IdentifyObject() for the dynamic ASM alternative
Link3 = wlHtml.GetLinkByIndex(3)
//Measure the time to connect to the link
SetTimer("Link Time")
wlHttp.Get(Link3.href)
SendTimer("Link Time")
```

Data Submission Properties

You may submit many types of data to an HTTP server in a Get, Post, or Head command. For example, you can submit a search string, the results of form fields, or a file. You may assign values to variable data stored in these collections using the `wlSet` method.

The `wlHttp` object provides the following collections to store the data that you want to submit:

- `FormData`
- `Data`
- `DataFile`
- `Header`

At the time you first create your test Agenda, you should decide which collection is most appropriate for your test session activities.

- `FormData` is the standard collection of field values, common to all HTML forms.
- `Header` is the collection of header field values only.
- `Data` and `DataFile` are both collections that hold sets of data.
  - `Data` collections are stored within the Agenda itself, and are useful when you prefer to see the data directly.
  - `DataFile` collections store the data in local text files, writing only the name of the data file within the Agenda itself, and are useful when you are working with large amounts of data, which would be too cumbersome to store within the Agenda code itself.

Your Agenda can work with both `Data` and `DataFile` collections. Do not use both properties on the same object (a single HTTP request can use either Data or DataFile collections, but not both).

FormData

`FormData` is a collection containing form field values. WebLOAD submits the field values to the HTTP server when you call one of the following methods of the `wlHttp` object:

- `Get()`
- `Post()`
- `Head()`

The collection indices are the field names (HTML name attributes). Before you call `wlHttp.Post()`, set the value of each element to the data that you want to submit in the HTML field. The fields can be any HTML controls, such as:

- buttons
- text areas
- hidden controls

Generally, when an HTTP client (Microsoft Internet Explorer/Firefox or WebLOAD) sends a URL-encoded request to the server, the data is HTTP encoded. Different clients – browsers or JavaScript functions such as Ajax – perform encoding with slight differences. Encoding form data replaces special characters such as blanks, ">" signs, and so on, with "%xx" (an ASCII hexa number). For example, a space is encoded as "%20" or as a "+".

The `Get()` method converts a URL address to contain only allowed URL characters. The `Post()` method converts the content of forms to contain only allowed URL characters. Forms contain one of the following content types:

- `FormData ["key"]` – a value
- `Data []` – raw information
- `DataFile []` – both a value and raw information

Turn off the encoding when the Agenda sends large requests that have no data that needs to be encoded. This improves performance as it bypasses the scanning and reformatting of the request buffer. You can enable or disable encoding, by accessing the HTTP Parameters tab in the Current or Default Project Options dialog box, in WebLOAD IDE. Check or uncheck Encode Form Data to enable or disable encoding.

Getting FormData Using Get()

You can get form data using a `Get()` call.

For example:

```
wlHttp.FormData["FirstName"] = "Bill"
```

```
wlHttp.FormData["LastName"] = "Smith"
wlHttp.FormData["EmailAddress"] = "bsmith@ABCDEF.com"
wlHttp.Get("http://www.ABCDEF.com/submit.cgi")
```
WebLOAD appends the form data to the URL as a query statement, using the following syntax:

```
http://www.ABCDEF.com/submit.cgi
   ?FirstName=Bill&LastName=Smith
   &EmailAddress=bsmith@ABCDEF.com
```

Submitting FormData Using Post()

Suppose you are testing an HTML form that requires name and email address data. You need to submit the form to the `submit.cgi` program, which processes the data.

You can code this in the following way:

```
wlHttp.FormData["FirstName"] = "Bill"
wlHttp.FormData["LastName"] = "Smith"
wlHttp.FormData["EmailAddress"] = "bsmith@ABCDEF.com"
wlHttp.Post("http://www.ABCDEF.com/submit.cgi")
```
The `Post()` call connects to `submit.cgi` and sends the `FormData` fields. In the above example, WebLOAD would post the following fields:

```
FirstName=Bill
LastName=Smith
EmailAddress=bsmith@ABCDEF.com
```

Submitting FormData with Missing Fields

You can use the string "`$WL$EMPTY$STRING$`" to represent a missing name or value (an empty string) in a `FormData` field. You can use the string "`$WL$VOID$STRING$`" to represent a token that is posted alone, without even an equal sign.

The following lines illustrate the syntax:

```
wlHttp.FormData["FirstName"] = "$WL$EMPTY$STRING$"
wlHttp.FormData["$WL$EMPTY$STRING$"] = "Smith"
wlHttp.FormData["EmailAddress"] = "$WL$VOID$STRING$"
wlHttp.Post("http://www.ABCDEF.com/submit.cgi")
```
In this example, the `Post()` call sends the following data:

```
FirstName=
=Smith
EmailAddress
```

A `Get()` call using the same form data would send:

```
http://www.ABCDEF.com/submit.cgi?FirstName=&=Smith&EmailAddress
```

Note: JavaScript supports two equivalent notations for named collection elements:
`FormData.FirstName`
`FormData["FirstName"]`
The latter notation also supports spaces in the name, for example, `FormData["First Name"]`.

Using FormData with Data Files

You can coordinate the `FormData` property with input data files. In this case, one of the `FormData` fields stores identifying information about the data file, including the name of the file and the type of data in the file (text, binary, bmp, etc.).

The following lines illustrate the syntax:

```
wlHttp.FormData["InputFile.wlFile-Name"] = "myFavoritePicture"
wlHttp.FormData["InputFile.wlContent-Type] = "bmp"
```

The following Agenda fragment illustrates the `FormData` properties documented in this section. The input file lines are marked in bold.

```
...
/***** WLIDE - URL : http://ws3/webft/fullform.asp - ID:4 *****/
wlGlobals.UserAgent = "Mozilla/4.0 (compatible; MSIE 7.0;
Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727; .NET CLR
3.0.4506.2152; .NET CLR 3.5.30729; .NET CLR 1.1.4322)"
wlHttp.Header["Referer"] = "http://ws3/webft/index.asp"
wlHttp.Get("http://ws3/webft/fullform.asp")
 // END WLIDE


/***** WLIDE - Sleep - ID:5 *****/
Sleep(18890)
 // END WLIDE


/***** WLIDE - URL : http://ws3/webft/fullform.asp - ID:6 *****/
wlGlobals.UserAgent = "Mozilla/4.0 (compatible; MSIE 7.0;
Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727; .NET CLR
3.0.4506.2152; .NET CLR 3.5.30729; .NET CLR 1.1.4322)"
wlHttp.Header["Referer"] = "http://ws3/webft/fullform.asp"
wlHttp.FormdataEncodingType = 1
wlHttp.ContentType = "application/x-www-form-urlencoded"
wlHttp.FormData["text1"] = "user1"
wlHttp.FormData["password1"] = "password1"
```

```
wlHttp.FormData["textarea1"] = "some text"

wlHttp.FormData["checkbox1"] = "second value"

wlHttp.FormData["select1"] = "first select value"

wlHttp.FormData["submit2"] = "Submit2"

wlHttp.Post("http://ws3/webft/fullform.asp")
 // END WLIDE

...
/***** WLIDE - URL : http://ws3/upload/ssl-upload.html - ID:16
*****/
wlGlobals.UserAgent = "Mozilla/4.0 (compatible; MSIE 7.0;
Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727; .NET CLR
3.0.4506.2152; .NET CLR 3.5.30729; .NET CLR 1.1.4322)"
wlHttp.Get("http://ws3/upload/ssl-upload.html")
 // END WLIDE


/***** WLIDE - Sleep - ID:17 *****/
Sleep(22999)
 // END WLIDE


/***** WLIDE - URL : http://ws3/upload/ssl-upload-handler.php -
ID:18 *****/
wlGlobals.UserAgent = "Mozilla/4.0 (compatible; MSIE 7.0;
Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727; .NET CLR
3.0.4506.2152; .NET CLR 3.5.30729; .NET CLR 1.1.4322)"
wlHttp.Header["Referer"] = "http://ws3/upload/ssl-upload.html"
wlHttp.EncodeFormdata = false;
wlHttp.ContentType = "multipart/form-data"

wlHttp.FormData["text"] = "abcd"

wlHttp.FormData["uploadFile[].wlFile-Name"] = "a.txt"

wlHttp.FormData["uploadFile[].wlContent-Type"] = "text/plain"

wlHttp.FormData["uploadFile[].wlFile-Name"] =
"$WL$EMPTY$STRING$"

wlHttp.FormData["uploadFile[].wlContent-Type"] =
"application/octet-stream"

wlHttp.FormData["uploadFile[].wlFile-Name"] =
"$WL$EMPTY$STRING$"

wlHttp.FormData["uploadFile[].wlContent-Type"] =
"application/octet-stream"

wlHttp.FormData["submit"] = "Submit Query"

wlHttp.Post("http://ws3/upload/ssl-upload-handler.php")


 // END WLIDE
```

Data

`Data` is a string to be submitted in an HTTP Post command.

The `Data` property has two subfields:

* `Data.Type`—the MIME type for the submission

* `Data.Value`—the string to submit

You can use `Data` in two ways:

* As an alternative to `FormData` if you know the syntax of the form submission.

* To submit a string that is not a standard HTML form and cannot be represented by `FormData`.

Thus the following three code samples are equivalent:

```
//Sample 1
wlHttp.Data.Type =
   "application/x-www-form-urlencoded"
wlHttp.Data.Value =
   "SearchFor=icebergs&SearchType=ExactTerm"
wlHttp.Post("http://www.ABCDEF.com/query.exe")
//Sample 2
wlHttp.FormData.SearchFor = "icebergs"
wlHttp.FormData.SearchType = "ExactTerm"
wlHttp.Post("http://www.ABCDEF.com/query.exe")
//Sample 3
wlHttp.Post
("http://www.ABCDEF.com/query.exe" +
   "?SearchFor=icebergs&SearchType=ExactTerm")
```

Note: `Data` and `DataFile` are both collections that hold sets of data.
`Data` collections are stored within the Agenda itself, and are useful when you prefer to see the data directly.
`DataFile` collections store the data in local text files, and are useful when you are working with large amounts of data, which would be too cumbersome to store within the Agenda code itself. When working with `DataFile` collections, only the name of the text file is stored in the Agenda itself.

Your Agenda can work with both Data and DataFile collections. Do not use both properties for the same lHTTP object (a single HTTP request can use either Data or DataFile properties, but not both). When you record your Agenda with the WebLOAD IDE, use the Post Data tab in the Recording and Script Generation Options dialog box to define the default behavior for submitting content types. The default

behaviors are Data and DataFile, although you can edit these definitions and add additional predefined behaviors for other content types as well.

DataFile

`DataFile` is a file to be submitted in an HTTP Post command. WebLOAD sends the file using a MIME protocol.

`DataFile` has two subfields:

- `DataFile.Type`—the MIME type

- `DataFile.Filename`—the name of the file, for example:

    `"c:\\MyWebloadData\\BigFile.doc"`

WebLOAD sends the contents of the file stored in `<filename>` in the Post command.

Header

A collection of HTTP header fields that you want to send in a `Get()`, `Post()`, or `Head()` call.

By default, WebLOAD sends the following header in any HTTP command:

```
Accept: */*
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
Host: <host>
Connection:Keep-Alive
```

Here, `<host>` is the host name to which you are connecting, for example:

```
www.ABCDEF.com:81.
```

By default, the Referer header is also recorded and sent, where it is being used by the Web server.

You may reset these properties, for example, as follows:

```
wlHttp.UserAgent = "Mozilla/4.03 [en] (WinNT; I)"
```

Note: The user-agent header is defined in WebLOAD IDE. Access **Tools ➢ Default or Current Project Options**, select the Browser Parameters tab, and edit the user-agent field.

Alternatively, you can use the `Header` property to override one of the default header fields. For example, you can redefine the following header field:

```
wlHttp.Header["user-agent"] =
  "Mozilla/4.03 [en] (WinNT; I)"
```

Additional header fields are recorded and sent according to the checked Record Headers options in the Script Content tab of the Recording and Script Generation Options dialog box (in WebLOAD IDE):

```
Accept-Language: en-us
UA-CPU: x86
Pragma: no-cache
```

Note: The headers `If-Modified-Since` and `If-None-Matched` will be commented out to overcome the situation where recorded links were fetched from the browser's cache during the recording. The request header `Accept-Encode: gzip` will also be commented out, to ensure correct behavior.

When the Record Custom Headers option is enabled, WebLOAD records any headers that are not explicitly defined in the RFC, such as the SOAP Action header. This option is not selected by default.

Note: Any information set using the `wlHttp.Header` property *takes priority* over any defaults set using the global, local, or other `wlHttp` properties. If there is any discrepancy between the document header information and the HTTP values, WebLOAD will work with the information found in the `wlHttp.Header` property while also issuing a warning to the user.

WebLOAD offers a simple way to reset configuration properties using the Options tab of the Session Control menu. Resetting configuration properties as you run and rerun various testing scenarios allows you to fine tune your tests to match your exact needs at that moment. However, that this real-time configuration setting will be overruled by any configuration properties that are explicitly set by `wlHttp.Header` within your test Agendas. For greatest reliability and flexibility, WebLOAD recommends that you set header properties using the more general `wlGlobals, wlLocals,` and `wlHttp` object properties, fine-tuning your test sessions using the WebLOAD Session menu. See *Rules of Scope for Local and Global Variables* (on page 49), for more information on precedence and priority in Agenda variables. Remember that you cannot override the host header or set a cookie header using the `Header` property. To set a cookie, see *How WebLOAD Works with Cookies* (on page 144).

Erase

Clear the WebLOAD properties of a `wlHttp` object after each `Get()`, `Post()`, or `Head()` call.

`WlHttp.Erase` is a read/write property. The default value is `true`. This section will describe the implications of each setting.

Erase=true (default)

When `Erase` is set to `true`, WebLOAD automatically erases all `wlHttp` property values after each HTTP access. You must reassign any properties you need before the next HTTP access. For this reason, assign the properties of `wlHttp` only in the *main script*, not in `InitClient()`, so they will be reassigned in every round.

Thus if `Erase` is set to `true` the following Agenda is incorrect. In this Agenda, the `wlHttp` properties are assigned values in `InitClient()`. The Agenda would connect to the `Url` and submit the `FormData` only in the first round. After the first `Post()` call, the `Url` and `FormData` property values are erased, so WebLOAD cannot use them in subsequent rounds.

The following script displays the incorrect method of assigning the `wlHttp` properties:

```
function InitClient() {    //Wrong!
wlHttp.Url =
   "http://www.ABCDEF.com/products.exe"
wlHttp.FormData["Name"] = "John Smith"
wlHttp.FormData["Product Interest"] = "Modems"
}
//Main script
wlHttp.Post()
```

To solve the problem, assign the `wlHttp` property values in the main script, so that the assignments are executed before each `Get()`, `Post()`, or `Head()` call:

```
//Main script            //OK
wlHttp.Url =
   "http://www.ABCDEF.com/products.exe"
wlHttp.FormData["Name"] = "John Smith"
wlHttp.FormData["Product Interest"] = "Modems"
wlHttp.Post()
```

Alternatively, you could assign values to `wlLocals` properties, which are not erased:

```
function InitClient() {    //OK
wlLocals.Url =
   "http://www.ABCDEF.com/products.exe"
wlLocals.FormData["Name"] = "John Smith"
wlLocals.FormData["Product Interest"] =
   "Modems"
}
//Main script
wlHttp.Post()
```

Erase=false

You may set `Erase` to `false` to prevent erasure. For example, if for some reason you absolutely had to assign values to the `wlHttp` properties in the `InitClient()` function of the Agenda, change the value of the `Erase` property to `false`. If `Erase` is `false`, the properties retain their values through subsequent rounds.

Thus another way to correct the preceding example is to write:

```
function InitClient() {    //OK
wlHttp.Erase = false
wlHttp.Url = "http://www.ABCDEF.com/products.exe"
wlHttp.FormData["Name"] = "John Smith"
wlHttp.FormData["Product Interest"] = "Modems"
}
//Main script
wlHttp.Post()
```

User-defined properties are not linked to the `wlHttp.Erase` property and will not be erased automatically by WebLOAD. The only way to reset or erase user-defined properties is for the user to set the new values explicitly.

Posting form Data Using Elements

Note: This Agenda includes DOM objects that are not explained in this section. For more information about a specific object or usage, (such as `wlGlobals` or `wlHttp`), see that object's section in the *WebLOAD JavaScript Reference Guide*.

The home page of a company displays the following form, where a user can specify interest in different products.
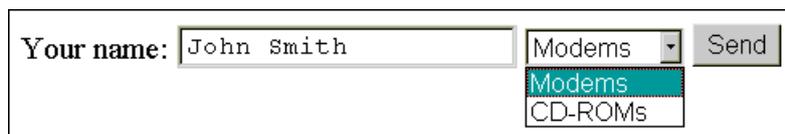
Figure 28: Home Page Form Example

The HTML code for the form is:

```
<FORM
   action="http://www.ABCDEF.com/FormProcessor.exe"
   method="post">
   <P>Your name: <INPUT type="text" name="yourname">
   <SELECT name="interest">
        <OPTION selected>Modems</OPTION>
```

```
      <OPTION>CD-ROMs</OPTION>
  </SELECT>
  <INPUT type="submit" value="Send">
</FORM>
```

An Agenda can download the form as follows:

```
function InitAgenda() {
  wlGlobals.Url = "http://www.ABCDEF.com"
  wlGlobals.ParseForms = true
}
wlHttp.Get()
```

WebLOAD parses the HTML code and creates an `elements` collection containing the first two form elements (the text box and the drop-down list). WebLOAD does not include the third element (the Send button) in the collection because it does not have a name attribute.

This is a listing of the `elements` collection:

```
document.forms[0].elements[0].name = "yourname"
document.forms[0].elements[0].type = "text"
document.forms[0].elements[0].value = ""
document.forms[0].elements[1].name = "interest"
document.forms[0].elements[1].type = "SELECT"
document.forms[0].elements[1].selectedindex = 0
document.forms[0].elements[1].options[0].text = "Modems"
document.forms[0].elements[1].options[0].value = "Modems"
document.forms[0].elements[1].options[0].selected = true
document.forms[0].elements[1].options[1].text = "CD-ROMs"
document.forms[0].elements[1].options[1].value = "CD-ROMs"
document.forms[0].elements[1].options[1].selected = false
```

The Agenda can use the above data to post the form back to the server. The following code illustrates how the Agenda might do this.

```
MyArray = document.forms[0].elements
i = 0
while (i < MyArray.length) {
  switch MyArray[i].type {
      case "text" :
      //Retrieve the default value of a text box
      wlHttp.FormData[MyArray[i].name] = MyArray[i].value]
      break
      case "SELECT"
      //Retrieve the first option in a drop-down list
```

```
        wlHttp.FormData[MyArray[i].name]=MyArray[i].options[0].v
        alue
        break
    }
i++
}
//Post the data to the form server
wlHttp.Post(document.forms[0].action)
```

Managing Cookies through the wlCookie Object

The wlCookie Object

The `wlCookie` object sets and deletes cookies. These activities may be required by an HTTP server.

Note: You may use the methods of `wlCookie` to create as many cookies as needed. For example, each WebLOAD client running an Agenda can set its own cookie identified by a unique name.

`wlCookie` is a local object. WebLOAD automatically creates an independent `wlCookie` object for each thread of an Agenda. You cannot manually declare `wlCookie` objects yourself. See the *WebLOAD JavaScript Reference Guide* for a complete syntax specification for the `wlCookie` object and its methods.

How WebLOAD Works with Cookies

WebLOAD always accepts cookies that are sent from a server. When WebLOAD connects to a server, it automatically submits any cookies in the server's domain that it has stored. By default, WebLOAD clears the cookie cache after every round.

The `wlCookie` object lets you supplement or override this behavior in the following ways:

• A thread can create its own cookies.

• A thread can delete cookies that it created.

Except for the above, WebLOAD does not distinguish in any way between cookies that it receives from a server and those that you create yourself. For example, if a thread creates a cookie in a particular domain, it automatically submits the cookie when it connects to any server in the domain.

wlCookie Methods

The `wlCookie` object works with the following methods:

- `ClearAll()`—Delete all cookies set by `wlCookie` in the current thread.

- `Delete()`—Delete the cookie identified by the method parameters.

   The cookie must have been previously created by `wlCookie`.

- `Get()`—Returns the cookie value. If there is more than one cookie with the same parameters it returns the first cookie.

- `Set()`—Creates a cookie. You can set an arbitrary number of cookies in any thread. If you set more than one cookie applying to a particular domain, WebLOAD submits them all when it connects to the domain.

Note: Set cookies within the main script of the Agenda. WebLOAD deletes all the cookies at the end of each round. If you wish to delete cookies in the middle of a round, use the `Delete()` or `ClearAll()` method.

Example: using a cookie

Cookies are used by applications to store the application requests on the client side. This enables the client to perform a one-time application request and from then on the request information is retrieved from the client's cookies.

```
//Request information entered into the application
wlHttp.ContentType = "application/x-www-form-urlencoded"
wlHttp.FormData["username"] = "zadp10"
wlHttp.FormData["password"] = "Start100"
wlHttp.FormData["submit1"] = "Login"

//Setting the cookie
wlCookie.Set("UserID", "zapd10", "www.abcdef.net,", "/", "Sun, 19-
Jun-2011 17:29:00 GMT")

//WebLOAD submits the cookie
wlHttp.Post("https://www.abcdef.net/default.aspx")

//Getting the cookie
CookieValue = wlCookie.Get("UserID")
//The value returned in CookieValue is "zapd10"
```

In some cases it is also necessary to delete the cookie settings. You can do this by entering:

```
//Delete the cookie
wlCookie.ClearAll()
```

Handling Binary Data

WebLOAD supports the simulation of binary data sent from the browser to the server. Binary data is handled specially, since it cannot be displayed as JavaScript literals in the Agenda, unlike other types of data that can be displayed, such as numbers or text.

Recording Binary Data

WebLOAD supports recording binary data in an Agenda in one of the following ways:

* Recording binary data as a data file, which is external to the Agenda.
* Encoding binary data in the Agenda, so that you can view and edit the data.

Recording Binary Data as a Data File

Modifying the Agenda Options settings in the WebLOAD IDE enables writing binary data into a data file, which is stored locally. When working with data files, only the name of the file is stored in the Agenda itself.

Recording binary data as a data file, enables WebLOAD to simulate sending binary data to the application, although the data itself cannot be edited in the Agenda.

To record binary data as a data file:

1. In WebLOAD IDE, select **Tools Recording and Script Generation Options**.

   The Recording and Script Generation Options dialog box appears.

2. In the Recording and Script Generation Options dialog box select the **Post Data** tab.

   The Post Data tab appears.
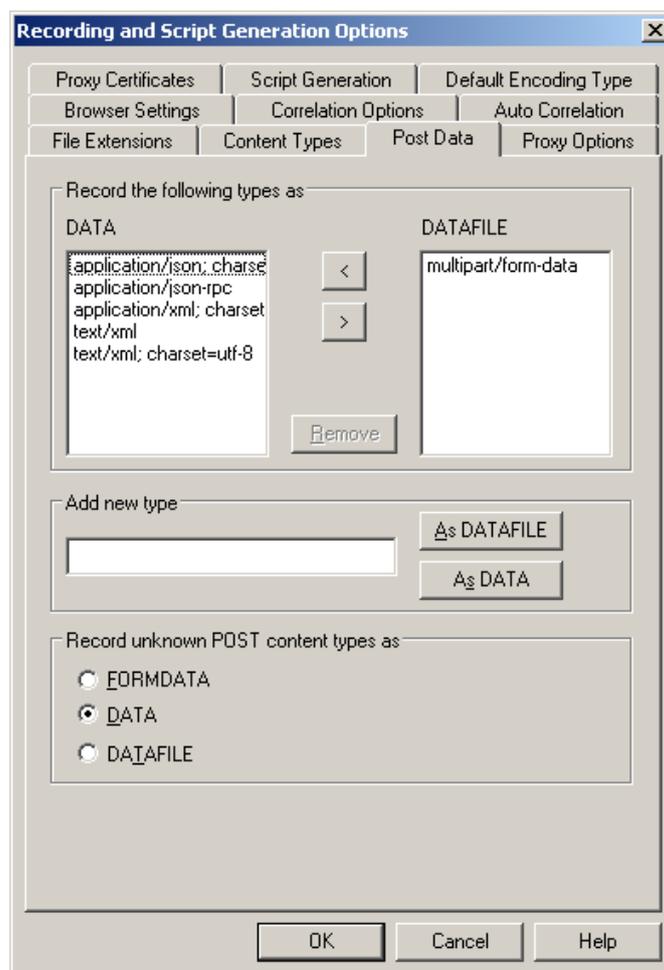
Figure 29: Recording and Script Generation Options – Post Data Tab

3. In the Add New Type field, enter the binary content-type, such as `application/x-amf`, which is used for binary content generated by a Flash application.

4. Click **As DATAFILE**.

   The binary content-type is added to the DATAFILE block list.

5. In the Recording and Script Generation Options dialog box, select the **Content Types** tab.

   The Content Types tab appears.
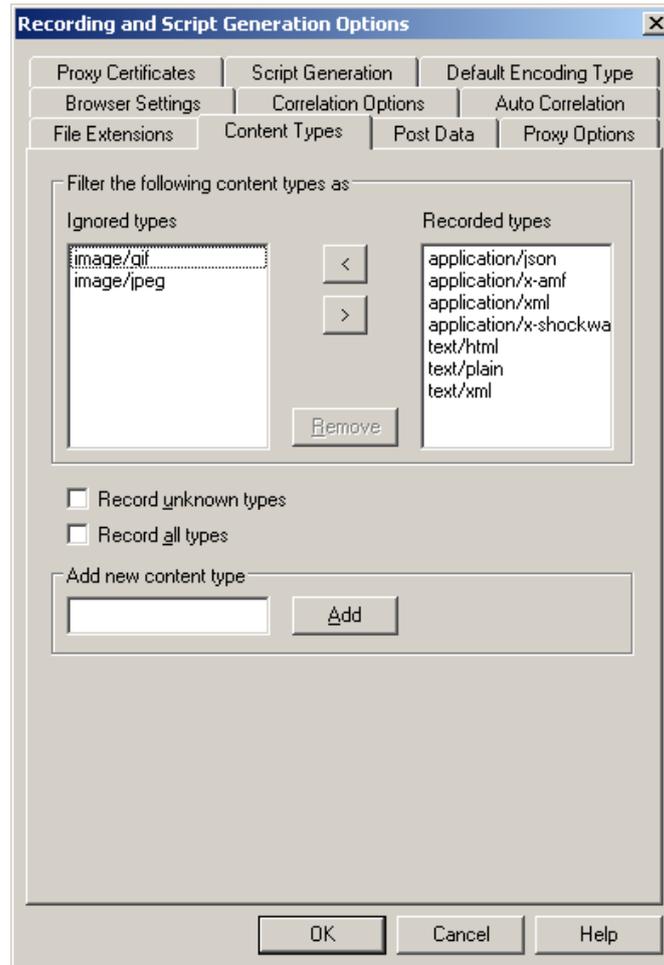
Figure 30: Recording and Script Generation Options – Content Types Tab

6. Ensure that the appropriate binary content-type appears in the Recorded Types area. If the content does not appear, add the binary content type in the Add new content type field and click **Add**.

   After setting the Record Option settings, binary data is stored in a local data file and the name of the file appears in the Agenda.

The following sample Agenda demonstrates how WebLOAD records binary data as a data file:

```
function InitAgenda()
{
CopyFile("wl2288601006.dat","wl2288601006.dat")
}
wlGlobals.GetFrames = false
wlHttp.Get("http://www.webloadmpstore.com/flex/bin/Sample.html")

Sleep(9056)
```

```
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/flex/bin/Shop.swf"
wlHttp.Header["Content-type"] = "application/x-amf"
wlHttp.Header["Content-length"] = "58"
wlHttp.DataFile["Type"] = "application/x-amf"
wlHttp.DataFile["Filename"] = "wl2288601006.dat"
wlHttp.Post("http://www.webloadmpstore.com/flashservices/gateway
.php")
```

Encoding Binary Data in the Agenda

Modifying the Agenda Options settings in the IDE enables the encoding of binary data in the Agenda during the recording. Once recorded, the Agenda can be edited, parameterized, and load tested as a regular Agenda.

To enable encoding binary data in the Agenda:

1. In WebLOAD IDE, select the **Tools ➤ Recording and Script Generation Options** dialog box.

   The Recording and Script Generation Options dialog box appears.

2. In the Recording and Script Generation Options dialog box select the **Script Generation** tab.

   The Script Generation tab appears.
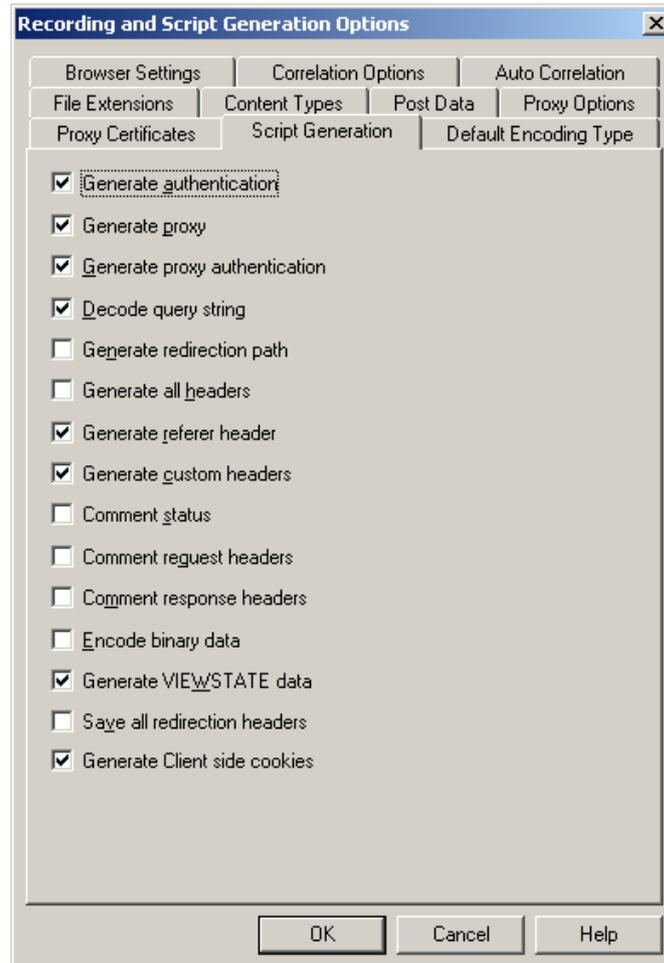
Figure 31: Recording and Script Generation Options - Script Generation Tab

3. Select the **Encode binary data** checkbox.

   This enables the user to view and parameterize the binary content.

4. In the Recording and Script Generation Options dialog box, select the **Post Data** tab.

   The Post Data tab appears.
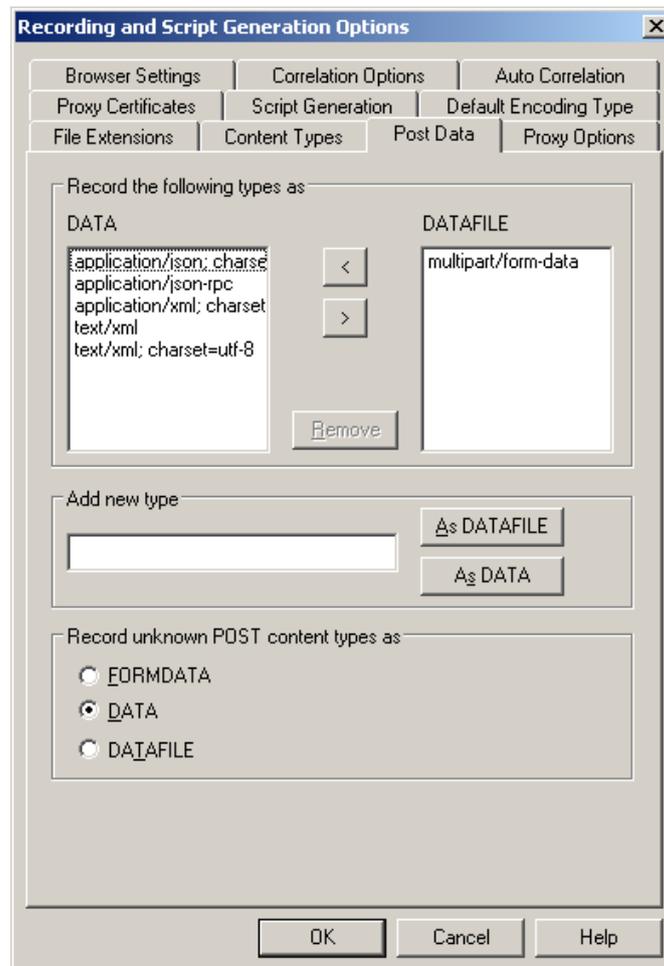
Figure 32: Recording and Script Generation Options – Post Data Tab

5. In the Add New Type field, enter the binary content-type, such as `application/x-amf`, which is used for binary text generated by a Flash application.

6. Click **As DATA**.

    The binary content-type is added to the DATA block list.

7. In the Recording and Script Generation Options dialog box, select the **Content Types** tab.

    The Content Types tab appears (see Figure 30).

8. Ensure that the appropriate binary content-type appears in the Recorded Types area. If the content does not appear, add the binary content type in the Add new content type field and click **Add**.

After setting the Record Option settings, when binary data is recorded and encoded in an Agenda, `wlHttp.EncodeRequestBinaryData = true` is automatically added to the script. This encodes the binary data into the following usable format:

```
wlHttp.Data["Value"] =
"%00%00%00%00%00%01%00/com.oreilly.frdg.SearchProducts.getSearch
Result%00%02/1%00%00%00%18%0a%00%00%00%01%03%00%06search%02%00%0
4wood%00%00%09"
```

The following sample Agenda demonstrates how WebLOAD handles binary data
encoded in the Agenda:

```
function InitAgenda()
{

IncludeFile("wlamf.js",WLExecuteScript);
}
wlGlobals.GetFrames = false
wlHttp.EncodeRequestBinaryData = true
wlGlobals.UserAgent = "Mozilla/4.0 (compatible; MSIE 8.0;
Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727; .NET CLR
3.0.4506.2152; .NET CLR 3.5.30729; .NET CLR 1.1.4322)"
wlHttp.Get("http://www.webloadmpstore.com/flex/bin/Sample.html")

wlHttp.EncodeRequestBinaryData = true
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/flex/bin/Sample.html"
wlHttp.Get("http://www.webloadmpstore.com/flex/bin/history.htm")

wlHttp.EncodeRequestBinaryData = true
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/flex/bin/history.htm"
wlHttp.Get("http://www.webloadmpstore.com/flex/bin/history.swf")

wlHttp.EncodeRequestBinaryData = true
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/flex/bin/Sample.html"
wlHttp.Get("http://www.webloadmpstore.com/flex/bin/Shop.swf")

wlHttp.EncodeRequestBinaryData = true
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/flex/bin/Shop.swf"
wlHttp.Data["Type"] = "application/x-amf"
wlHttp.Data["Value"] =
"%00%03%00%00%00%01%00%15talkback.returnString%00%02/1%00%00%00%
15%0a%00%00%00%02%02%00%06George%02%00%04Bush"
```

```
wlHttp.Post("http://www.webloadmpstore.com/flashservices/gateway
.php")
```

Handling Authentication in the Agenda

WebLOAD supports working with Agendas containing Basic, NTLM, and Kerberos user authentication methods.

WebLOAD handles authentication in playback in the following way:

1. WebLOAD sends a regular request with no request for authentication, to which the server responds with an "unauthorized" error (401), and a prompt to enter a username and password.

2. If WebLOAD is using:

   - **Basic authentication**, WebLOAD encrypts the username and password in `wlHttp.Username` and `wlHttp.Password`, and makes a request.

   - **NTLM authentication**, WebLOAD sets the following:

     `wlGlobals.AuthType = "NTLM"`, where `NTLM` is the default value if no authentication type is specified.

   - **Kerberos authentication**, WebLOAD sets the following:

     `wlGlobals.AuthType = "Kerberos"`.

     `wlGlobals.KDCServer = <server name>`. If no value is specified for `KDCServer`, the authentication type is automatically assumed to be `NTLM`, even if the `AuthType` is `Kerberos`.

The following examples demonstrate the authentication processes in WebLOAD.

Kerberos authentication:

```
wlGlobals.AuthType = "Kerberos"
wlGlobals.KDCServer = "mulier.qalab.internal"
wlHttp.NTPassWord = "Buga859"
wlHttp.NTUserName = "qalab.internal\\test"
wlHttp.Get("http://mulier.qalab.internal/")
```

NTLM authentication:

```
wlGlobals.AuthType = "NTLM"
wlHttp.NTPassWord = "Buga859"
wlHttp.NTUserName = "qalab.internal\\test"
wlHttp.Get("http://mulier.qalab.internal/")
OR
```

```
wlHttp.NTPassWord = "Buga859"
wlHttp.NTUserName = "qalab.internal\\test"
wlHttp.Get("http://mulier.qalab.internal/")
```

Basic authentication:

```
wlHttp.PassWord = "Buga859"
wlHttp.UserName = "test"
wlHttp.Get("http://mulier.qalab.internal/")
```

Scripting Samples

This chapter provides sample scripts which you can adapt to your own Agendas. Each scripting sample demonstrates different features that can be performed by editing your Agenda's script.

The following scripting samples are provided:

- Basic Recording
- Correlation
- Parameterizing an Agenda
- Using AJAX and Web services
- Using AJAX and JSON

Scripting Sample of a Basic Recording

The sample Agenda in this section demonstrates how a basic Agenda is recorded. This Agenda is used as a basis for the scripts that appear in the following sections.

What the Script Does

- Records user actions in the website. This is done by recording the HTTP traffic between the browser and the Web server.
- Records Html form data sent from the browser.
- Records the `Get()` and `Post()` methods.

How to Create the Script

The basic Agenda is created by starting to record in the IDE, browsing through the www.webloadmpstore.com website, stopping the recording, and saving the Agenda.

Step 1 – Starting to Record the Agenda

1. In WebLOAD IDE open a new Agenda.

2. Click ⏺ or select **Record ➤ Start Record** to start recording.

3. Browse to www.webloadmpstore.com in the browser that opens.

    The following script describing your action appears in the Agenda.

```
wlGlobals.GetFrames = false
wlHttp.Get("http://www.webloadmpstore.com/")


Sleep(9094)
```

Step 2 – Logging in to the Site

1. Click **Login** in the webloadmpstore website and the Login page appears. The script for this action is as follows:

```
wlHttp.Header["Referer"] = "http://www.webloadmpstore.com/"
wlHttp.Get("http://www.webloadmpstore.com/login.php")


Sleep(7704)
```

2. In the username and password fields, enter `demo-username` and `demo-password` to log in to the website.

    WebLOAD IDE records the login action and simulates a script containing the username and password that you entered in the login page.

```
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/login.php"
wlHttp.ContentType = "application/x-www-form-urlencoded"
wlHttp.FormData["login"] = "demo"
wlHttp.FormData["password"] = "demo"
wlHttp.FormData["sessionID"] =
"webloadmpstore.62.90.23.122.ae97bc7877d7bd4ebcb64c4c0e21ba1c"
wlHttp.FormData["event"] = "login"
wlHttp.FormData["Submit"] = "Login"
wlHttp.Post("http://www.webloadmpstore.com/login.php")


Sleep(3015)
```

    The site approves your login information and the WebLOAD MP Store home page appears.

```
wlHttp.Get("http://www.webloadmpstore.com/index.php")
Sleep(5282)
```

Step 3 – Purchasing a Product

1. From the product page, click **Debugging and Handling Dynamic Data** to view the product's additional details.

   A page with the product description for Debugging and Handling Dynamic Data appears.

```
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/index.php"
wlHttp.FormData["id"] = "1"
wlHttp.Get("http://www.webloadmpstore.com/product.php")


Sleep(4984)
```

2. Click **Add to Cart** to purchase the product.

```
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/product.php?id=1"
wlHttp.FormData["event"] = "addproduct"
wlHttp.FormData["id"] = "1"
wlHttp.Get("http://www.webloadmpstore.com/cart.php")
```

Step 4 – Saving the Agenda

1. In WebLOAD IDE, click ▇ to stop recording the Agenda.

2. Select **File ➤ Save As** to save the Agenda.

   The Save As dialog box appears.

3. In the Save As dialog box browse to the following location:
   `D:\\Radview\\<Sample scripts folder>\\ Agenda 1-Basic Recording Agenda.wlp`

4. Click **Save**.

The Full Agenda: Agenda 1-Basic Recording Agenda

Note: The WLIDE – URL comments throughout the Agenda are modified to give the Agenda nodes meaningful names.

```
/***** WLIDE - URL : Open webloadmpstore home page - ID:2 *****/
wlGlobals.GetFrames = false
wlHttp.Get("http://www.webloadmpstore.com/")
// END WLIDE


/***** WLIDE - Sleep - ID:3 *****/
Sleep(9094)
```

```
// END WLIDE


/***** WLIDE - URL : Open login page - ID:4 *****/
wlHttp.Header["Referer"] = "http://www.webloadmpstore.com/"
wlHttp.Get("http://www.webloadmpstore.com/login.php")
// END WLIDE


/***** WLIDE - Sleep - ID:5 *****/
Sleep(7704)
// END WLIDE


/***** WLIDE - URL :Insert login and password - ID:6 *****/
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/login.php"
wlHttp.ContentType = "application/x-www-form-urlencoded"
wlHttp.FormData["login"] = "demo"
wlHttp.FormData["password"] = "demo"
wlHttp.FormData["sessionID"] =
"webloadmpstore.62.90.23.122.ae97bc7877d7bd4ebcb64c4c0e21ba1c"
wlHttp.FormData["event"] = "login"
wlHttp.FormData["Submit"] = "Login"
wlHttp.Post("http://www.webloadmpstore.com/login.php")
// END WLIDE


/***** WLIDE - Sleep - ID:7 *****/
Sleep(3015)
// END WLIDE


/***** WLIDE - URL : http://www.webloadmpstore.com/index.php -
ID:8 *****/
wlHttp.Get("http://www.webloadmpstore.com/index.php")
// END WLIDE


/***** WLIDE - Sleep - ID:9 *****/
Sleep(5282)
// END WLIDE


/***** WLIDE - URL : Select show product details - ID:10 *****/
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/index.php"
wlHttp.FormData["id"] = "1"
wlHttp.Get("http://www.webloadmpstore.com/product.php")
```

```
// END WLIDE


/***** WLIDE - Sleep - ID:11 *****/
Sleep(4984)
// END WLIDE


/***** WLIDE - URL : Add to cart - ID:12 *****/
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/product.php?id=1"
wlHttp.FormData["event"] = "addproduct"
wlHttp.FormData["id"] = "1"
wlHttp.Get("http://www.webloadmpstore.com/cart.php")
// END WLIDE
```

Scripting Sample of Correlation

When you record HTML pages using the IDE, there can be dynamic values that the application is using, which WebLOAD records automatically to the Agenda. Such dynamic values can contain state management information, such as the session-id, which is usually passed as URL encoded parameters or hidden form fields. The dynamic values that are recorded in the IDE are different during each run. Since the value that was recorded in the IDE is static, using it in further executions of the Agenda will result in an application error.

To overcome this situation, you need to manually edit the Agenda and perform correlation. Correlation enables you to store the dynamic value when it is received in a local parameter in the application and then use the dynamic value instead of the static value, in the rest of the Agenda.

What the Script Does

- Demonstrates how to extract the contents of the hidden field.
- Demonstrates how to perform correlation of the session ID.

How to Create the Script

The script is created by locating and extracting the contents of the hidden field. Add a variable to store the contents and then replace the static values with the variable.

Step 1 – Locating and Extracting the Hidden Field

1. Open the Agenda1-Basic Recording Agenda and run the Agenda.

2. Select **View ➤ Browser View** and then **View ➤ DOM View** to display the Browser and DOM views.

    The Browser view appears with the DOM view open.

3. In the Execution Tree, select the first node.

4. In the Browser View, search for an error message. If there is no error message, select the next node in the Execution Tree and search for an error message there.
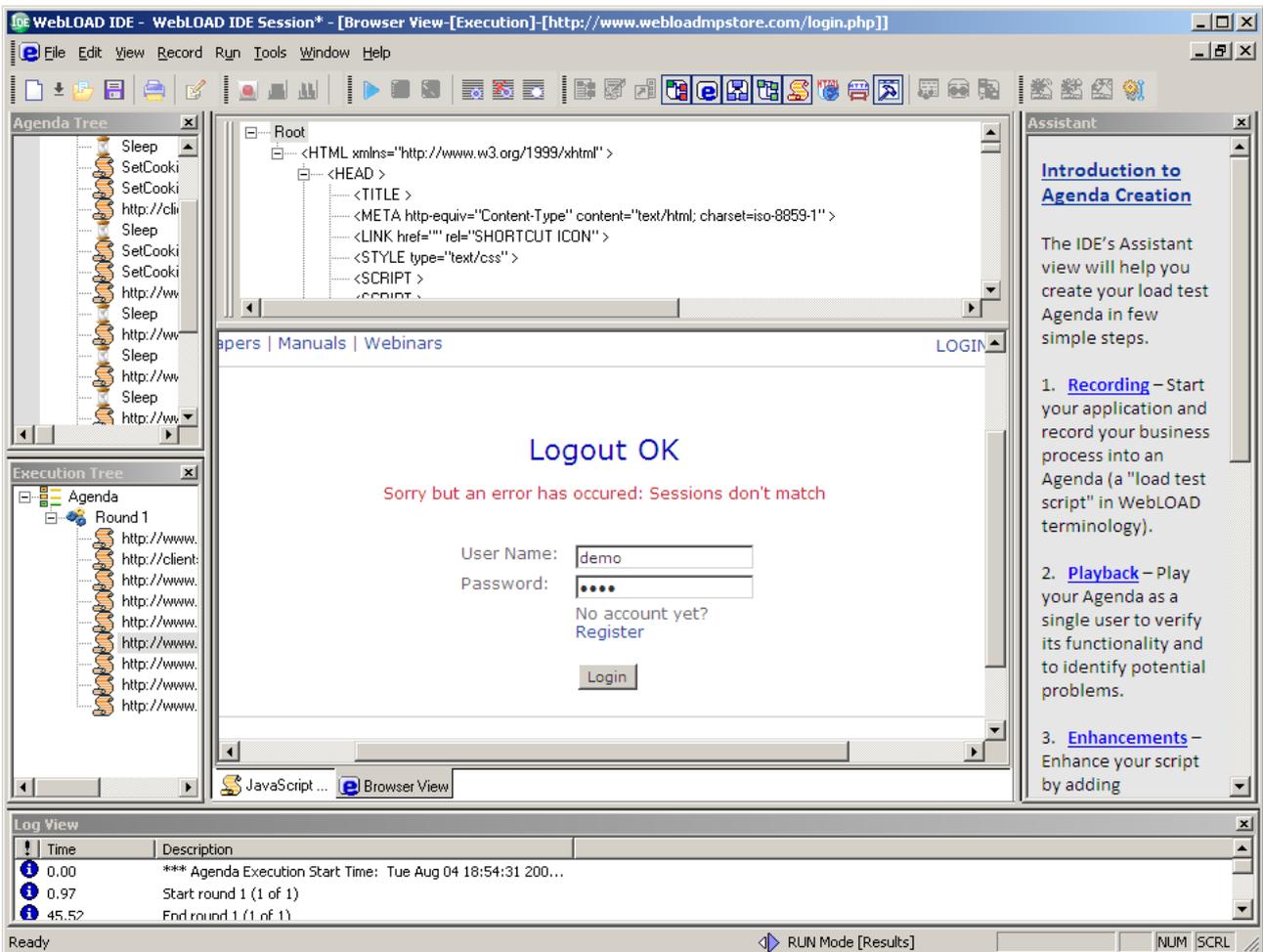
Figure 33: Browser View Displaying Error Message

5. Once you locate the error message, select **View ➤ Java Script View** to open the JavaScript View.

    The JavaScript View appears.

6. Reselect the node in the Execution Tree to display the requested block of code in the Agenda.

7. Within the selected block of code, locate the dynamic value (for example, the session-id field). This field must be retrieved from the previous block of code.

```
21  □ /***** WLIDE - URL :Insert login and password - ID:6 *****/
22    wlHttp.Header["Referer"] = "http://www.webloadmpstore.com/login.php"
23    wlHttp.ContentType = "application/x-www-form-urlencoded"
24    wlHttp.FormData["login"] = "demo"
25    wlHttp.FormData["password"] = "demo"
26    wlHttp.FormData["sessionID"] = "webloadmpstore.62.90.23.122.ae97bc7877d7bd4ebcb64c4c0e2Tb
27    wlHttp.FormData["event"] = "login"
28    wlHttp.FormData["Submit"] = "Login"
29    wlHttp.Post("http://www.webloadmpstore.com/login.php")
30
```

Figure 34: Browser View Displaying Error Message

8. Select the previous node in the Execution Tree to search for the element that contains the dynamic value. Make sure the Browser and DOM Views are open. (Select **View ▶ Browser View** and **View ▶ DOM View**.)

9. In the DOM View, locate the element that contains the dynamic value. This is usually a hidden input field.

   The following figure displays the hidden field in the DOM view.

Figure 35: Hidden Field in DOM View

10. Right-click the hidden field and select **Smart Copy** from the pop-up window to extract the dynamic value of the hidden field.

   The Smart Copy dialog box appears.
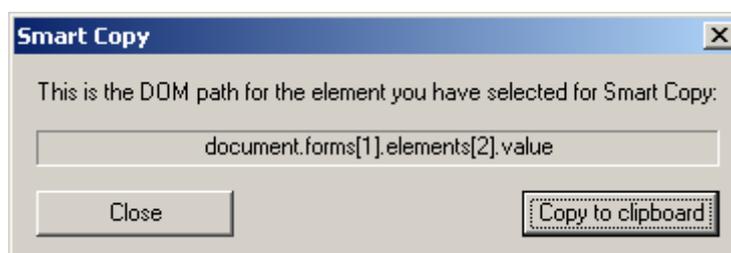
Figure 36: Smart Copy Dialog Box

11. Click **Copy to clipboard** and click **Close**.

Step 2 – Adding the Variable for the Dynamic Data

1. Select **Edit ➤ Start Visual Editing** to edit the JavaScript.

2. Drag the JavaScript Building Block from the Toolbox to the Agenda Tree to create a variable for the extracted dynamic field.

3. Type the following in the new JavaScript Building Block:

   ```
   session_id =
   ```

4. Paste the clipboard text by placing the cursor after the equal sign and selecting **Edit ➤ Paste**.

   ```
   session_id = document.forms[1].elements[2].value
   ```

Step 3 – Replacing the Static Values with the Variable

1. Locate the dynamic value (for example the session-id field). This is the same value that you located in step 7 of *Step 1 – Locating and Extracting the Hidden Field* (see Figure 34).

2. Remove the static value of the Session ID from the hidden field and replace it with the variable containing the dynamic field.

Before:

```
wlHttp.Header["Referer"]
="http://www.webloadmpstore.com/login.php"
wlHttp.ContentType ="application/x-www-form-urlencoded"
wlHttp.FormData["login"] = "demo"
wlHttp.FormData["password"] = "demo"
wlHttp.FormData["sessionID"] =
"webloadmpstore.62.90.23.122.ae97bc7877d7bd4ebcb64c4c0e21ba1c"
wlHttp.FormData["event"] = "login"
wlHttp.FormData["Submit"] = "Login"
wlHttp.Post("http://www.webloadmpstore.com/login.php")
```

After:

```
wlHttp.Header["Referer"]
="http://www.webloadmpstore.com/login.php"
wlHttp.ContentType ="application/x-www-form-urlencoded"
wlHttp.FormData["login"] = "demo"
wlHttp.FormData["password"] = "demo"
wlHttp.FormData["sessionID"] = session_id
wlHttp.FormData["event"] = "login"
wlHttp.FormData["Submit"] = "Login"
wlHttp.Post("http://www.webloadmpstore.com/login.php")
```

3. Save the Agenda as Agenda2-Correlation Agenda.

The Agenda is now corrected. You can now run the Agenda successfully without receiving error messages.

The Full Agenda: Agenda 2-Correlation Agenda

```
/***** WLIDE - URL : Open webloadmpstore home page - ID:2 *****/
wlGlobals.GetFrames = false
wlHttp.Get("http://www.webloadmpstore.com/")
// END WLIDE


/***** WLIDE - Sleep - ID:3 *****/
Sleep(9094)
// END WLIDE


/***** WLIDE - URL : Open login page - ID:4 *****/
wlHttp.Header["Referer"] = "http://www.webloadmpstore.com/"
wlHttp.Get("http://www.webloadmpstore.com/login.php")
// END WLIDE


/***** WLIDE - Sleep - ID:5 *****/
Sleep(7704)
// END WLIDE


/***** WLIDE - Session ID correlation - ID:13 *****/
session_id = document.forms[1].elements[2].value


/***** WLIDE - URL :Insert login and password - ID:6 *****/
wlHttp.Header["Referer"]
="http://www.webloadmpstore.com/login.php"
wlHttp.ContentType ="application/x-www-form-urlencoded"
wlHttp.FormData["login"] = "demo"
wlHttp.FormData["password"] = "demo"
wlHttp.FormData["sessionID"] = session_id
wlHttp.FormData["event"] = "login"
wlHttp.FormData["Submit"] = "Login"
wlHttp.Post("http://www.webloadmpstore.com/login.php")
 // END WLIDE


/***** WLIDE - Sleep - ID:7 *****/
```

```
Sleep(3015)
 // END WLIDE


/***** WLIDE - URL : http://www.webloadmpstore.com/index.php -
ID:8 *****/
wlHttp.Get("http://www.webloadmpstore.com/index.php")
// END WLIDE


/***** WLIDE - Sleep - ID:9 *****/
Sleep(5282)
// END WLIDE


/***** WLIDE - URL : Select show product details - ID:10 *****/
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/index.php"
wlHttp.FormData["id"] = "1"
wlHttp.Get("http://www.webloadmpstore.com/product.php")
// END WLIDE


/***** WLIDE - Sleep - ID:11 *****/
Sleep(4984)
// END WLIDE


/***** WLIDE - URL : Add to cart - ID:12 *****/
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/product.php?id=1"
wlHttp.FormData["event"] = "addproduct"
wlHttp.FormData["id"] = "1"
wlHttp.Get("http://www.webloadmpstore.com/cart.php")
// END WLIDE
```

Scripting Sample of Parameterizing an Agenda

You can replace the static variables that were recorded in your WebLOAD Agenda
with parameters, making it easier to manage the script. When you want to update a
recorded value, you need to locate each instance where the value appears in the script
and update the value. After parameterizing your Agenda, the recorded values are
replaced with parameters, whose definitions can easily be modified, without having to
go over the whole script and modify each recorded value.

You can parameterize your script from within the Agenda using the JavaScript
Building Block. With this method the parameters are defined within the Agenda. To

update the parameter values, you must edit the definition of the parameter within the script.

You can also parameterize your script from an external file. This method enables you to update the parameter values by simply editing the information saved in the external file without having to edit the script. In addition, storing the parameter values in an external file ensures that the information is kept in a more secure location.

What the Script Does

- Demonstrates how to add additional code to an existing Agenda.

- Demonstrates how to work with the parameters within the script and parameters whose values are imported from an external file.

- Demonstrates how to work with Building Blocks.

How to Create the Script

In this Agenda you create parameters, both within the Agenda and from an external file. The parameter within the Agenda is created by adding a JavaScript Building Block to your Agenda. Within the building block's script you define the new parameters.

In this Agenda you also import additional parameter values from an external file. You create the external file by using the GlobalInputFile Building Block. The Building Block stores the location and contents of the external file. Once you add the Building Block to your Agenda, the `InitAgenda` function associates the location and contents of the external file with the script. The script added by the GlobalInputFile Building Block to the Agenda, contains a variable that stores the contents of the current line of the external file.

Both of these methods enable you to control the login information by simply updating the parameter definition in a single location, instead of having to edit each of the recorded values in the script. After creating the parameters, you must search through the Agenda for each instance where the username and password information was recorded, and replace the recorded value with the parameter.

Step 1 – Continuing the Recording

1. Open the Agenda2-Correlation Agenda.

2. Open a browser, navigate to the Cart page of the webloadmpstore site, and copy the URL of the page.

3. In the IDE, click or select **Record ➤ Start Record** to start recording.

4. Paste the URL of the cart page in the browser that opens.

   The new script is appended to the end of the existing Agenda.

5. In the cart page, in the quantity field, enter 3 and click ✅ to update the page.

   The appended script is as follows:

   ```
   wlGlobals.GetFrames = false
   wlHttp.FormData["event"] = "update"
   wlHttp.FormData["id"] = 1
   wlHttp.FormData["quantity"] = 3
   wlHttp.Get("http://www.webloadmpstore.com/cart.php")
   ```

6. In WebLOAD IDE, click 🟥 to stop recording the Agenda.

Step 2 – Creating Parameters

1. In the IDE, drag the JavaScript Building Block from the Toolbox to the top of the Agenda Tree.

2. In the JavaScript Building Block, create two internal parameters by defining a name and value for each of the parameters.

   For example:

   ```
   product_id = "1"
   quantity = "3"
   ```

3. Instead of giving the parameters a static value, you can define a range from which a random value is chosen when the Agenda is run.

   ```
   product_id = wlRand.Range(1,9)
   quantity = wlRand.Range(1,12)
   ```

Step 3 – Adding the GlobalInput File Building Block

In this step, an external file is created and associated with the Agenda. Two variables are created to store the information from the external file to use within the Agenda.

1. Drag the GlobalInputFile Building Block 📄▶ anywhere in the Agenda Tree.

   The Global Input File dialog box appears.
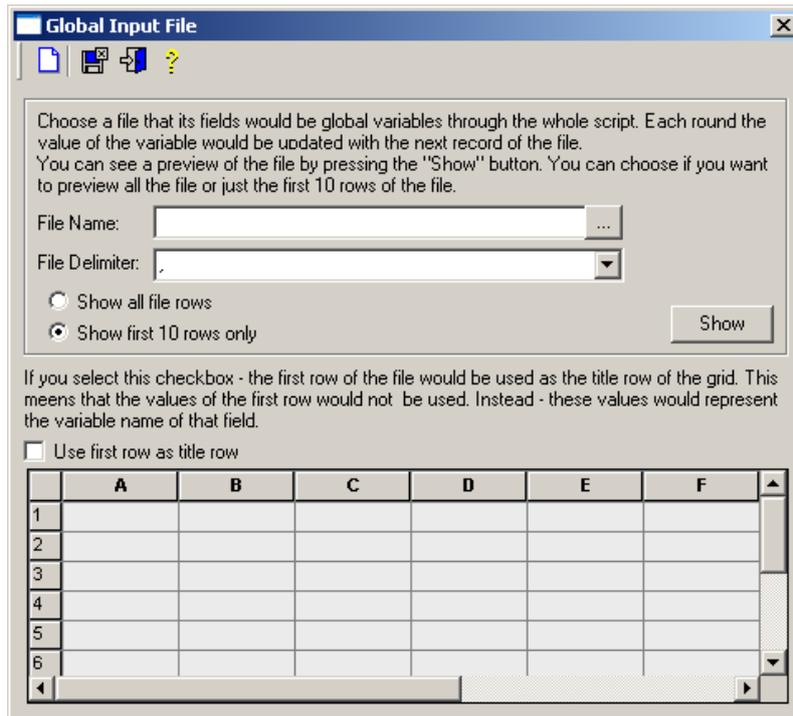
Figure 37: Global Input File Building Block

2. Check Use first row as title row.

3. Click [ ] to create a new external file.

   The Create Data File dialog box appears.
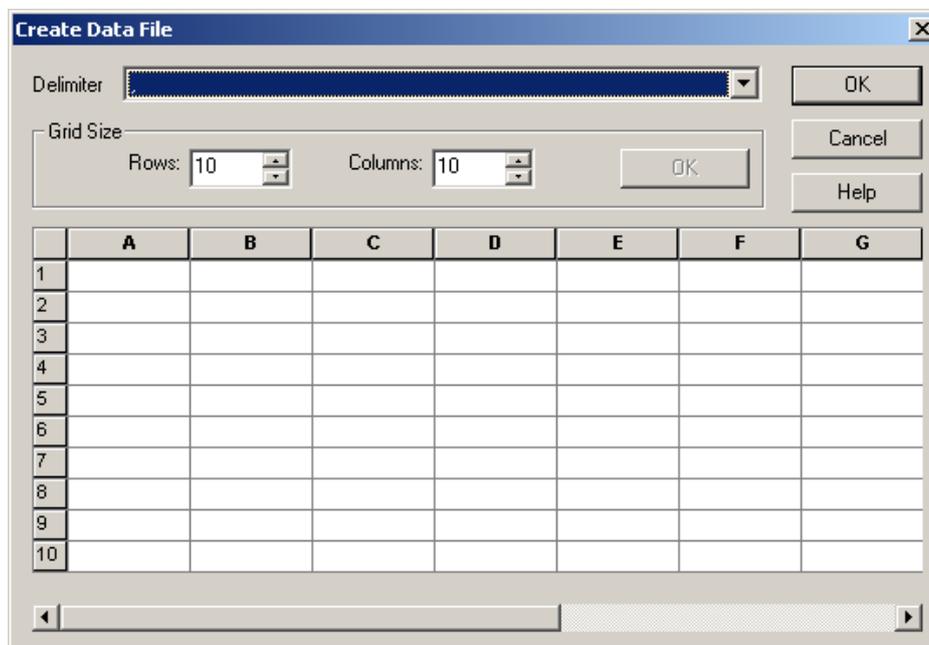
Figure 38: Create Data File

4. In the Grid Size area, in the Rows field enter `3` and in the Columns field enter `2`. Click **OK**.

The newly sized grid appears.

5. In the title row, enter `Username` and `Password`. Enter two new usernames (`demo1`,and `demo2`) and passwords (`password1`, and `password2`) in the grid fields and click **OK**. This creates two sets of parameters to be used during each round in the Agenda.

The Save As dialog box appears.

6. In the Save As dialog box, save the file in the folder where the sample scripts are located.

After saving the file, you are prompted with the Create message box asking you if you want to set this file to be imported by the GlobalInputFile Building Block.

7. In the Create message box, click **Yes**.

The Global Input File dialog box appears (Figure 37) with the path to the file location in the File Name field and the file information displayed in the grid.
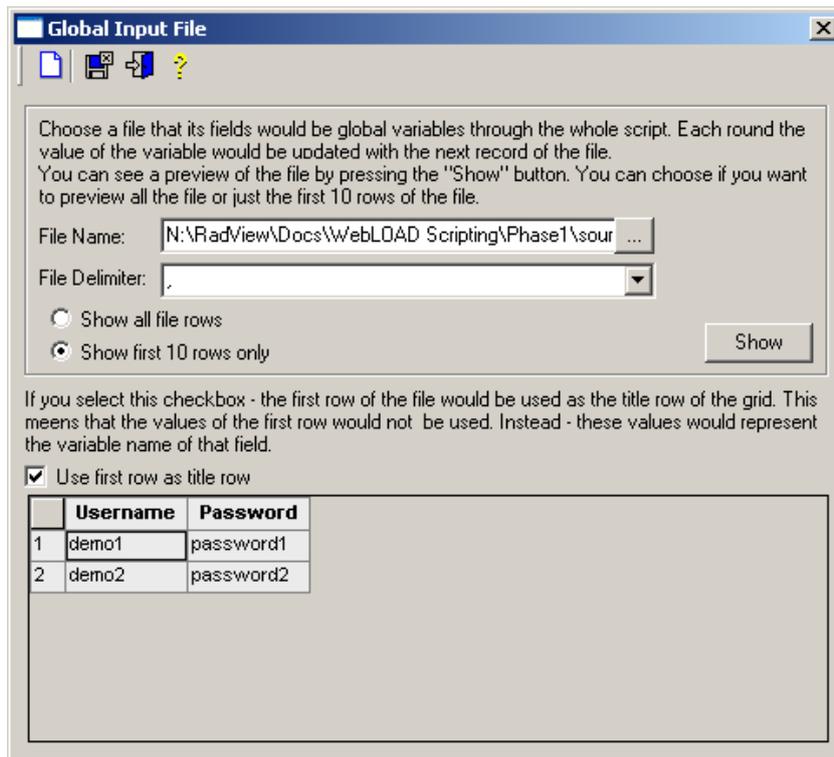
Figure 39: Global Input File Building Block Displaying Grid

8. Click ![save icon] to save the file and create the Building Block in the Agenda.

The external file is automatically associated with the script in the `initAgenda()` function. The external file is referred to within the Agenda as `Infile1`. When there is more than one external file, files are referred to as `Infile2`, `Infile3` etc.

Note: You can optionally provide a filename of your choice in the Global Input File dialog box, while saving the external file.

```
function InitAgenda()
{
InFile1 = CopyFile("P:\\Supporting Tools\\Sample
Scripts\\New\\InFile1.txt")
Open(InFile1)
strGlobalInputFileLine = GetLine(InFile1,",")
}
```

In the GlobalInputFile Building Block script, the `strGlobalInputFileLine` parameter stores the contents of the current line that was read from `Infile1`.

```
strGlobalInputFileLine = GetLine(InFile1,",")
inFile1_Col1 = strGlobalInputFileLine[1]
```

9. In the GlobalInputFile node, edit the existing variable name to be `user_name` and add a second variable to store the password information from the `strGlobalInputFileLine` parameter.

Note: This step is only necessary if you did not check Use first row as title row in the Global Input file dialog box in step 2 of *Step 3 – Adding the GlobalInput File Building Block*.

Before:

```
strGlobalInputFileLine = GetLine(InFile1,",")
inFile1_Col1 = strGlobalInputFileLine[1]
```

After:

```
strGlobalInputFileLine = GetLine(InFile1,",")
user_name = strGlobalInputFileLine[1]
password = strGlobalInputFileLine[2]
```

Step 4 – Replacing the Recorded Values with the New Parameters

1. Throughout the Agenda, replace the values for the product_id and quantity with the new parameters that you set up in *Step 2 – Creating Parameters* (`product_id` and `quantity`).

Before:

```
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/index.php"
wlHttp.FormData["id"] = 1
wlHttp.Get("http://www.webloadmpstore.com/product.php")


Sleep(4984)


wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/product.php?id=product_id
"
wlHttp.FormData["event"] = "addproduct"
wlHttp.FormData["id"] = 1
wlHttp.Get("http://www.webloadmpstore.com/cart.php")


wlGlobals.GetFrames = false
wlHttp.FormData["event"] = "update"
wlHttp.FormData["id"] = 1
wlHttp.FormData["quantity"] = 3
wlHttp.Get("http://www.webloadmpstore.com/cart.php")
```

After:

```
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/index.php"
wlHttp.FormData["id"] = product_id
wlHttp.Get("http://www.webloadmpstore.com/product.php")


Sleep(4984)


wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/product.php?id=product_id
"
wlHttp.FormData["event"] = "addproduct"
wlHttp.FormData["id"] = product_id
wlHttp.Get("http://www.webloadmpstore.com/cart.php")


wlGlobals.GetFrames = false
wlHttp.FormData["event"] = "update"
wlHttp.FormData["id"] = product_id
wlHttp.FormData["quantity"] = quantity
wlHttp.Get("http://www.webloadmpstore.com/cart.php")
```

2. Throughout the Agenda, replace the static login values with the new parameters that you set up in *Step 3 – Adding the GlobalInput File Building Block* (`user_name` and `password`).

Before:

```
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/login.php"
wlHttp.ContentType = "application/x-www-form-urlencoded"
wlHttp.FormData["login"] = "demo_user"
wlHttp.FormData["password"] = "demo_password"
wlHttp.FormData["sessionID"] = session_id
wlHttp.FormData["event"] = "login"
wlHttp.FormData["Submit"] = "Login"
wlHttp.Post("http://www.webloadmpstore.com/login.php")
```

After:

```
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/login.php"
wlHttp.ContentType = "application/x-www-form-urlencoded"
wlHttp.FormData["login"] = user_name
wlHttp.FormData["password"] = password
wlHttp.FormData["sessionID"] = session_id
wlHttp.FormData["event"] = "login"
wlHttp.FormData["Submit"] = "Login"
wlHttp.Post("http://www.webloadmpstore.com/login.php")
```

3. Save the Agenda as Agenda3-Parametrizing an Agenda.

The Agenda is now parameterized.

Note: Before running the Agenda set the number of iterations in Playback to two, by selecting **Tools ➤ Settings ➤ Playback**. For every round WebLOAD reads a different set of parameters from the file and uses the parameters during the login process. You can check the Browser View to see whether the login was successful for each round.

The Full Agenda: Agenda3-Parameterizing an Agenda

```
function InitAgenda()
{
InFile1 = CopyFile("P:\\Supporting Tools\\Sample
Scripts\\New\\InFile1.txt")
Open(InFile1)
strGlobalInputFileLine = GetLine(InFile1,",")
```

```
}
/***** WLIDE - GlobalInputFile: InFile1 (P:\\Supporting
Tools\\Sample Scripts\\New\\InFile1.txt) - ID:18 *****/
strGlobalInputFileLine = GetLine(InFile1,",")
if (strGlobalInputFileLine.LineNum == 1)
   strGlobalInputFileLine = GetLine(InFile1,",")

Username = strGlobalInputFileLine[1]
Password = strGlobalInputFileLine[2]
// END WLIDE


/***** WLIDE - URL : Open webloadmpstore home page - ID:2 *****/
wlGlobals.GetFrames = false
wlHttp.Get("http://www.webloadmpstore.com/")
// END WLIDE


/***** WLIDE - Sleep - ID:3 *****/
Sleep(9094)
// END WLIDE


/***** WLIDE - URL : Open login page - ID:4 *****/
wlHttp.Header[ "Referer"] = "http://www.webloadmpstore.com/"
wlHttp.Get( "http://www.webloadmpstore.com/login.php")
// END WLIDE


/***** WLIDE - Sleep - ID:5 *****/
Sleep(7704)
// END WLIDE


/***** WLIDE - Session ID correlation - ID:13 *****/
session_id = document.forms[1].elements[2].value
// END WLIDE


/***** WLIDE - URL : Insert login and password - ID:6 *****/
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/login.php"
wlHttp.ContentType = "application/x-www-form-urlencoded"
wlHttp.FormData["login"] = user_name
wlHttp.FormData["password"] = password
wlHttp.FormData["sessionID"] = session_id
wlHttp.FormData["event"] = "login"
```

```
wlHttp.FormData["Submit"] = "Login"
wlHttp.Post("http://www.webloadmpstore.com/login.php")
// END WLIDE


/***** WLIDE - Sleep - ID:7 *****/
Sleep(3015)
// END WLIDE


/***** WLIDE - URL : http://www.webloadmpstore.com/index.php -
ID:8 *****/
wlHttp.Get("http://www.webloadmpstore.com/index.php")
// END WLIDE


/***** WLIDE - Sleep - ID:9 *****/
Sleep(5282)
// END WLIDE


/***** WLIDE - Parameterizing product ID and category number -
ID:16 *****/
product_id = wlRand.Range(1,9)
quantity = wlRand.Range(1,12)
// END WLIDE


/***** WLIDE - URL : Select show product details - ID:10 *****/
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/index.php"
wlHttp.FormData["id"] = product_id
wlHttp.Get("http://www.webloadmpstore.com/product.php")
// END WLIDE


/***** WLIDE - Sleep - ID:11 *****/
Sleep(4984)
// END WLIDE


/***** WLIDE - URL : Add to cart - ID:12 *****/
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/product.php?id=product_id"
wlHttp.FormData["event"] = "addproduct"
wlHttp.FormData["id"] = product_id
wlHttp.Get("http://www.webloadmpstore.com/cart.php")
// END WLIDE
```

```
/***** WLIDE - URL : Product quantity - ID:17 *****/
wlGlobals.GetFrames = false
wlHttp.FormData[ "event"] = "update"
wlHttp.FormData[ "id"] = product_id
wlHttp.FormData[ "quantity"] = quantity
wlHttp.Get( "http://www.webloadmpstore.com/cart.php")
// END WLIDE
```

Scripting Sample Using AJAX and Web Services

WebLOAD supports automatic recording of AJAX calls into the test Agenda, enabling debugging and full access to all request data (headers and body), both in the script and during runtime. WebLOAD supports various formats for the AJAX payload: XML, JSON, other text-based formats, and binary data.

Besides demonstrating the use of AJAX calls, this Agenda demonstrates the use of functions and external files in the script.

Storing sections of the logic in a function enables you to reuse the same lines of code without duplications, making the script more modular. A function can be part of your main script file or can be stored in a separate, external JavaScript file. When storing the function in a separate file, the main Agenda file includes the function so that it can be used within the Agenda.

One of the main benefits of including files in the Agenda, is to reduce the maintenance needed for the scripts. The same included file can be used in a number of Agendas simply by adding the include command and calling the function in the script.

Using included files is also more efficient. When the information in the included file needs to be updated, the included file is the only place that needs to be modified and the whole script will be affected. Without using an included file, you would need to search for every place that the information is used and update the information manually.

What the Script Does

- Demonstrates how to record user actions in a website that is accessed on a secure server.

- Demonstrates how WebLOAD supports AJAX and Web services.

- Demonstrate how to validate a Web Service reply by parsing the XML content of its SOAP message.

- Demonstrates how to modify a recorded Agenda where a specific option is selected, so that the Agenda can accept additional options during runtime.

- Demonstrates adding a function to the script.
- Demonstrates how to extract a login script to an external JavaScript file and then reuse the code in the Agenda.

How to Create the Script

In this Agenda, you will record a webpage that is accessed on a secure server. Once the Agenda is recorded, you will modify the script so that during runtime, an InfoMessage notifies you whether your credit card information has been validated. This is done by adding a function that uses a Web Service to check the validation of the credit card and a parameter to store the result of the function.

You will create a function to store the script of the login process. Then, you can extract the function to an external file, which enables you to create a generic scenario, instead of a specific case. The function checks the login information received from the site and determines whether the user's login information is accurate. Although the login information was correct when the Agenda was recorded, by modifying the script you can add additional scenarios to be accepted during runtime, such as, when the login information is incorrect.

This script also demonstrates WebLOAD's support of AJAX and Web services. During the recording, when the credit card information is being validated, the IDE records the Web service transactions that take place within the application.

Step 1 – Entering Credit Card Information and Checking Out

1. Open the Agenda3-Parametrizing an Agenda Agenda.

2. Open a browser, navigate to the Cart page, and copy the URL of the page.

3. In the IDE, click 🔴 or select **Record ➤ Start Record** to start recording.

4. Paste the URL of the cart page in the browser that opens.

   The new script is appended to the end of the existing Agenda.

5. In the cart page, click **Checkout**.

   The credit card information page appears. This page is an HTTPS page and is appended to the Agenda as follows:

   ```
   wlGlobals.GetFrames = false
   wlHttp.Header["Referer"] =
   "http://www.webloadmpstore.com/cart.php?event=update&id=
   2&quantity=3"
   wlHttp.Get("https://www.webloadmpstore.com/checkout.php"
   )
   ```

6. In the credit card information page, enter your credit card information and click **Done**. The Agenda is recorded as follows:

```
wlGlobals.GetFrames = false
wlHttp.Header["Referer"] =
"https://www.webloadmpstore.com/checkout.php"
wlHttp.FormData["wsdl"] = "$WL$VOID$STRING$"
wlHttp.Get("https://www.webloadmpstore.com/soap/server.p
hp")


wlHttp.Header["Referer"] =
"https://www.webloadmpstore.com/checkout.php"
wlHttp.Data["Type"] = "text/xml; charset=utf-8"
wlHttp.Data["Value"] = "<?xml version=\"1.0\"
encoding=\"utf-8\"?><soap:Envelope
xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"
xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\"
><soap:Body><checkValidity
xmlns=\"http://example.org/CreditCardProcess\"><strCardN
umber>ABCD</strCardNumber><strHolderID>AB1234</strHolder
ID></checkValidity></soap:Body></soap:Envelope>"


wlHttp.Post("https://www.webloadmpstore.com/soap/server.
php")


wlHttp.ContentType = "application/x-www-form-urlencoded"
wlHttp.FormData["name"] = "Radview"
wlHttp.FormData["address"] = "Hamelacha 14"
wlHttp.FormData["shippingAddress"] = "Park Afek"
wlHttp.FormData["cardNumber"] = "ABCD"
wlHttp.FormData["idNumber"] = "AB1234"
wlHttp.FormData["event"] = "process"
wlHttp.Post("https://www.webloadmpstore.com/checkout.php
")
```

7. In WebLOAD IDE, click to stop recording the Agenda.

Step 2 – Adding the Results Parameter and ResultParser Function

1. Add a `results` parameter to the Validate Cred – Pass Node of the Agenda.

    The `results` parameter stores a value received from the `resultParser` function, which checks whether your credit card information is valid. The HTTP response containing the result of the credit card validation Web Service

(`document.wlXmls[0]`) that is extracted from the script, is sent to the `resultParser` function.

The `results` parameter appears in the Validate Cred – Pass Node of the Agenda as follows:

```
wlHttp.Header["Referer"] =
"https://www.webloadmpstore.com/checkout.php"

wlHttp.Data["Type"] = "text/xml; charset=utf-8"

wlHttp.Data["Value"] = "<?xml version=\"1.0\"
encoding=\"utf-8\"?><soap:Envelope
xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"
xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\"
><soap:Body><checkValidity
xmlns=\"http://example.org/CreditCardProcess\"><strCardN
umber>ABCD</strCardNumber><strHolderID>AB1234</strHolder
ID></checkValidity></soap:Body></soap:Envelope>"


wlHttp.Post("https://www.webloadmpstore.com/soap/server.
php")


results = resultParser( document.wlXmls[0] )
```

2. Create an infoMessage to notify you whether your credit card has been validated or not according to the `result` parameter value.

```
wlHttp.Header["Referer"] =
"https://www.webloadmpstore.com/checkout.php"

wlHttp.Data["Type"] = "text/xml; charset=utf-8"

wlHttp.Data["Value"] = "<?xml version=\"1.0\"
encoding=\"utf-8\"?><soap:Envelope
xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"
xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\"
><soap:Body><checkValidity
xmlns=\"http://example.org/CreditCardProcess\"><strCardN
umber>ABCD</strCardNumber><strHolderID>AB1234</strHolder
ID></checkValidity></soap:Body></soap:Envelope>"


wlHttp.Post("https://www.webloadmpstore.com/soap/server.
php")


results = resultParser( document.wlXmls[0] )
if (results == "1")
InfoMessage("Validation check return 'Check OK' like it
should")
else
```

```
    InfoMessage("Validation check is not working correctly")
```

3. Add the `resultParser` function to the script, which checks whether your credit card information is valid. The function retrieves the information from `document.wlXmls[0]` in the `results` parameter and uses the built-in WebLOAD XML DOM to parse and return XML data to the parameter.

```
function resultParser (doc)
{
        //get 'Result' tags, we expect exactly one result:
        ResultsElements =
        doc.getElementsByTagName("Result")

        if ( ResultsElements.length != 1 )
        // Verify only one element with that name exists
        throw "Expecting a single Result tag. received " +
        ResultsElements.length + "elements"

        ResultElm = ResultsElements.item(0)

        if ( ResultElm.childNodes.length == 0 )
            return null
        else
        {
                Result = ResultElm.firstChild
                return Result.nodeValue
        }
}
```

Step 3 – Creating the External Login JavaScript File

1. Select the section of the script that is responsible for the login process.

2. Copy the script to a separate JavaScript file and save the file as `login_js.js`.

    The script in the `login_js.js` file is as follows:

```
function Login() {
strGlobalInputFileLine = GetLine(InFile1,",")
user_name = strGlobalInputFileLine[1]
password = strGlobalInputFileLine[2]

wlGlobals.GetFrames = false
wlHttp.Get("http://www.webloadmpstore.com/")
```

```
      Sleep(9094)

      wlHttp.Header["Referer"] =
      "http://www.webloadmpstore.com/"
      wlHttp.Get("http://www.webloadmpstore.com/login.php")

      Sleep(7704)

      session_id = document.forms[1].elements[2].value

      wlHttp.Header["Referer"] =
      "http://www.webloadmpstore.com/login.php"
      wlHttp.ContentType = "application/x-www-form-urlencoded"
      wlHttp.FormData["login"] = user_name
      wlHttp.FormData["password"] = password
      wlHttp.FormData["sessionID"] = session_id
      wlHttp.FormData["event"] = "login"
      wlHttp.FormData["Submit"] = "Login"
      wlHttp.Post("http://www.webloadmpstore.com/login.php")
      }
```

Step 4 – Including and Using the External File in the Agenda

1. Include the external JavaScript file in the Agenda by copying the following into the `InitAgenda()` function of the Agenda:

   ```
   IncludeFile("login_js.js")
   ```

2. Drag the JavaScript object Building Block into the Agenda.

3. Add the `rs` parameter that calls the login function from the included `login_js.js` file.

   ```
   rs = Login()
   ```

4. Save the Agenda as Agenda 4–Ajax and Web services.

The Full Agenda: Agenda 4-AJAX and Web Services

```
function InitAgenda()
{
InFile1 = CopyFile("P:\\Supporting Tools\\Sample
Scripts\\New\\InFile1.txt")
Open(InFile1)
IncludeFile("login_js.js")
}
```

```
/***** WLIDE -function resultParser - ID:33 *****/
function resultParser (doc)
{
   //get 'Result' tags, we expect exactly one result:
   ResultsElements = doc.getElementsByTagName("Result")

   if ( ResultsElements.length != 1 )
   // Verify only one element with that name exist
        throw "Expecting a single Result tag. received " +
        ResultsElements.length + "elements"

   ResultElm = ResultsElements.item(0)

   if ( ResultElm.childNodes.length == 0 )
        return null
   else
   {
        Result = ResultElm.firstChild
        return Result.nodeValue
   }
}
 // END WLIDE


/***** WLIDE - Login Function - ID:19 *****/
rs = Login()
 // END WLIDE


/***** WLIDE - Sleep - ID:7 *****/
Sleep(3015)
 // END WLIDE


/***** WLIDE - URL : http://www.webloadmpstore.com/index.php -
ID:8 *****/
wlHttp.Get("http://www.webloadmpstore.com/index.php")
 // END WLIDE


/***** WLIDE - Sleep - ID:9 *****/
Sleep(5282)
 // END WLIDE


/***** WLIDE - Parameterizing product ID and category number  -
ID:28 *****/
```

```
product_id = wlRand.Range(1,9)
quantity = wlRand.Range(1,12)
 // END WLIDE


/***** WLIDE - URL : Select show product details - ID:10 *****/
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/index.php"
wlHttp.FormData["id"] = product_id
wlHttp.Get("http://www.webloadmpstore.com/product.php")
 // END WLIDE


/***** WLIDE - Sleep - ID:11 *****/
Sleep(4984)
 // END WLIDE


/***** WLIDE - URL : Add to cart - ID:12 *****/
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/product.php?id=product_id"
wlHttp.FormData["event"] = "addproduct"
wlHttp.FormData["id"] = product_id
wlHttp.Get("http://www.webloadmpstore.com/cart.php")
 // END WLIDE


/***** WLIDE - URL : Product quantity - ID:17 *****/
wlGlobals.GetFrames = false
wlHttp.FormData["event"] = "update"
wlHttp.FormData["id"] = product_id
wlHttp.FormData["quantity"] = quantity
wlHttp.Get("http://www.webloadmpstore.com/cart.php")
 // END WLIDE


/***** WLIDE - URL : HTTPS - ID:22 *****/
wlGlobals.GetFrames = false
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/cart.php?event=update&id=2&quanti
ty=3"
wlHttp.Get("https://www.webloadmpstore.com/checkout.php")
 // END WLIDE


/***** WLIDE - URL :
https://www.webloadmpstore.com/soap/server.php?wsdl - ID:23
*****/
```

```
wlGlobals.GetFrames = false

wlHttp.Header["Referer"] =
"https://www.webloadmpstore.com/checkout.php"

wlHttp.FormData["wsdl"] = "$WL$VOID$STRING$"

wlHttp.Get("https://www.webloadmpstore.com/soap/server.php")
 // END WLIDE


/***** WLIDE - URL : Validate Cred - Pass - ID:26 *****/

wlHttp.Header["Referer"] =
"https://www.webloadmpstore.com/checkout.php"

wlHttp.Data["Type"] = "text/xml; charset=utf-8"

wlHttp.Data["Value"] = "<?xml version=\"1.0\" encoding=\"utf-
8\"?><soap:Envelope
xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"
xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\"><soap:B
ody><checkValidity
xmlns=\"http://example.org/CreditCardProcess\"><strCardNumber>AB
CD</strCardNumber><strHolderID>AB1234</strHolderID></checkValidi
ty></soap:Body></soap:Envelope>"


wlHttp.Post("https://www.webloadmpstore.com/soap/server.php")


results = resultParser( document.wlXmls[0] )
if (results == "1")
  InfoMessage("Validation check return 'Check OK' like it
  should")
else
  InfoMessage("Validation check is not working correctly")
 // END WLIDE


/***** WLIDE - URL : Confirm Order - ID:27 *****/

wlHttp.ContentType = "application/x-www-form-urlencoded"

wlHttp.FormData["name"] = "Radview"

wlHttp.FormData["address"] = "Hamelacha 14"

wlHttp.FormData["shippingAddress"] = "Park Afek"

wlHttp.FormData["cardNumber"] = "ABCD"

wlHttp.FormData["idNumber"] = "AB1234"

wlHttp.FormData["event"] = "process"

wlHttp.Post("https://www.webloadmpstore.com/checkout.php")
 // END WLIDE
```

Scripting Sample Using AJAX and JSON to Validate a Web Server Response

WebLOAD supports automatic recording of AJAX calls into the test Agenda using JSON, enabling debugging and full access to all request data (headers and body), both in the script and during runtime.

What the Script Does

- Demonstrates how WebLOAD supports AJAX and JSON.
- Demonstrates how to validate a Web service response.

How the Script Works

In this Agenda, you will search for a product in the webloadmpstore site and in the search results page, you will request to see the site's statistics. The statistics displayed include the number of users online and the date and time. These statistics are updated every ten seconds to ensure their accuracy by using an asynchronous AJAX call. Each time the information is updated, the AJAX script is recorded in the Agenda by WebLOAD IDE.

The following steps demonstrate how to modify the AJAX script so that the IDE displays the JSON information in the log view during runtime. This is done by adding an InfoMessage to the Agenda, which displays the information based on the JSON response.

Step 1 – Recording AJAX Calls in the Agenda

1. Start recording a new Agenda and browse to www.webloadmpstore.com.

   The Agenda is recorded as follows:

   ```
   wlGlobals.GetFrames = false
   wlHttp.Get("http://www.webloadmpstore.com/index.php")
   ```

2. In the webloadmpstore website, enter WebLOAD in the Search field and click **SEARCH**.

   The search results for WebLOAD appear. In the IDE the following script is recorded:

   ```
   wlHttp.Header["Referer"] =
   "http://www.webloadmpstore.com/index.php"
   wlHttp.ContentType = "application/x-www-form-urlencoded"
   wlHttp.FormData["event"] = "search"
   wlHttp.FormData["searchTerm"] = "webload"
   ```

```
wlHttp.Post("http://www.webloadmpstore.com/search.php")
```

3. Check the **Show statistics** checkbox. The script in the Agenda appears as follows:

```
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/search.php"
wlHttp.Get("http://www.webloadmpstore.com/usersonlinecou
nter.php")
```

New nodes are recorded repeatedly and added to the script as the site automatically refreshes itself every ten seconds.

4. After a few nodes have been added to the script, click ◼ in WebLOAD IDE to stop recording the Agenda.

Step 2 – Parsing the JSON Response

1. In the `InitAgenda()` function in the Agenda, define the global variable values of the `SaveSource` property as follows:

```
function InitAgenda()
{
wlGlobals.SaveSource = true
}
```

This instructs WebLOAD to store the complete HTML source code downloaded in the `document.wlSource` object.

2. Drag the JavaScript object Building Block from the Toolbox to the Agenda Tree. In the Building Block, add a function that receives the `document.wlSource` object and manipulates it to retrieve the statistics. The script is as follows:

```
function evalResponse (source) {
json_response = eval("(" + source + ")")
}
```

3. Create an `InfoMessage` to notify you of the statistic values during runtime. The edited script is as follows:

```
function evalResponse (source) {
        json_response = eval("(" + source + ")")
InfoMessage ( "The number of online users is: " +
json response.usersOnline + " and The current time is: "
+ json_response.time)
}
```

4. In each of the AJAX calls recorded in the Agenda, add a call to the newly created function:

```
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/search.php"
```

```
          wlHttp.Get("http://www.webloadmpstore.com/usersonlinecou
          nter.php")
          evalResponse(document.wlSource)
```

5.   Save the Agenda as Agenda 5-AJAX and JSON.

Step 3 – Displaying JSON Information During Runtime

- Run the script. Each time the browser refreshes itself, a message in the log view is
  displayed. The message contains the statistics recorded in the browser, including
  the time and number of users online:

  ```
  The number of online users is: 1 and The current time
  is: 11:23:11am
  ```

The Full Agenda: Agenda 5-AJAX and JSON

```
function InitAgenda()
{
wlGlobals.SaveSource = true
 // END WLIDE
}
/***** WLIDE – URL : Home Page  - ID:20 *****/
wlGlobals.GetFrames = false
wlHttp.Get("http://www.webloadmpstore.com/index.php")
 // END WLIDE


/***** WLIDE - Sleep - ID:22 *****/
Sleep(8371)
 // END WLIDE


/***** WLIDE -Search page - ID:23 *****/
wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/index.php"
wlHttp.ContentType = "application/x-www-form-urlencoded"
wlHttp.FormData["event"] = "search"
wlHttp.FormData["searchTerm"] = "webload"
wlHttp.Post("http://www.webloadmpstore.com/search.php")
 // END WLIDE


/***** WLIDE - Sleep - ID:24 *****/
Sleep(8542)
 // END WLIDE


/***** WLIDE – JavaScriptObject - ID:27 *****/
```

```
function evalResponse (source) {
  json_response = eval("(" + source + ")")
  InfoMessage ( "The number of online users is: " +
  json_response.usersOnline + " and The current time is: " +
  json_response.time)
}
// END WLIDE


/***** WLIDE - URL : Ajax - ID:7 *****/

wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/search.php"

wlHttp.Get("http://www.webloadmpstore.com/usersonlinecounter.php
")


evalResponse(document.wlSource)
// END WLIDE


/***** WLIDE - Sleep - ID:8 *****/

Sleep(10811)
// END WLIDE


/***** WLIDE - URL : Ajax - ID:9 *****/

wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/search.php"

wlHttp.Get("http://www.webloadmpstore.com/usersonlinecounter.php
")


evalResponse(document.wlSource)
// END WLIDE


/***** WLIDE - Sleep - ID:10 *****/

Sleep(10640)
// END WLIDE


/***** WLIDE - URL : Ajax - ID:11 *****/

wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/search.php"

wlHttp.Get("http://www.webloadmpstore.com/usersonlinecounter.php
")


evalResponse(document.wlSource)
// END WLIDE
```

```
/***** WLIDE - Sleep - ID:12 *****/

Sleep(10655)
 // END WLIDE


/***** WLIDE - URL : Ajax - ID:13 *****/

wlHttp.Header["Referer"] =
"http://www.webloadmpstore.com/search.php"
wlHttp.Get("http://www.webloadmpstore.com/usersonlinecounter.php
")


evalResponse(document.wlSource)
 // END WLIDE
```

Appendix B

LiveConnect Overview

This appendix describes using LiveConnect technology to let Java and JavaScript code communicate with each other. LiveConnect is a registered trademark of Netscape Communications Corporation. This information is provided by Netscape, and can be found in the JavaScript Reference site at
http://www.js-examples.com/page/reference__partjava.html).

This appendix assumes you are familiar with Java programming. For additional information on LiveConnect, see the LiveConnect information on the Mozilla Developer Center (http://developer.mozilla.org/en/docs/LiveConnect).

Working with Wrappers

In JavaScript, a *wrapper* is an object of the target language data type that encloses an object of the source language. On the JavaScript side, you can use a wrapper object to access methods and fields of the Java object; calling a method or accessing a property on the wrapper results in a call on the Java object. On the Java side, JavaScript objects are wrapped in an instance of the class netscape.javascript.JSObject and passed to Java.

When a JavaScript object is sent to Java, the runtime engine creates a Java wrapper of type JSObject. When a JSObject is sent from Java to JavaScript, the runtime engine unwraps the JSObject revealing the original JavaScript object type. The JSObject class provides an interface for invoking JavaScript methods and examining JavaScript properties.

JavaScript to Java Communication

When you refer to a Java package or class, or work with a Java object or array, you use one of the special LiveConnect objects. All JavaScript access to Java takes place with these objects, which are summarized in the following table.

Table 9: The LiveConnect object set

Object	Description
JavaArray	A wrapped Java array, accessed from within JavaScript code.
JavaClass	A JavaScript reference to a Java class.
JavaObject	A wrapped Java object, accessed from within JavaScript code.
JavaPackage	A JavaScript reference to a Java package.

Note: Because Java is a strongly typed language and JavaScript is weakly typed, the JavaScript runtime engine converts argument values into the appropriate data types for the other language when you use LiveConnect. See *Data Type Conversions* (on page 197), for more information.

In some ways, the existence of the LiveConnect objects is transparent, because you interact with Java in a fairly intuitive way. For example, you can create a Java String object and assign it to the JavaScript variable myString by using the new operator with the Java constructor, as follows:

```
var myString = new java.lang.String("Hello world")
```
In the previous example, the variable myString is a JavaObject because it holds an instance of the Java object String. As a JavaObject, myString has access to the public instance methods of java.lang.String and its superclass, java.lang.Object. These Java methods are available in JavaScript as methods of the JavaObject, and you can call them as follows:

```
myString.length() // returns 11
```

The Packages Object

If a Java class is not part of the java, sun, or netscape packages, you access it with the Packages object. For example, suppose the Redwood Corporation uses a Java package called redwood to contain various Java classes that it implements.

To create an instance of the HelloWorld class in redwood, you access the constructor of the class as follows:

```
var red = new Packages.redwood.HelloWorld()
```
You can also access classes in the default package (that is, classes that don't explicitly name a package). For example, if the HelloWorld class is directly in the CLASSPATH and not in a package, you can access it as follows:

```
var red = new Packages.HelloWorld()
```

The LiveConnect java, sun, and netscape objects provide shortcuts for commonly used Java packages. For example, you can use the following:

```
var myString = new java.lang.String("Hello world")
```
instead of the longer version:

```
var myString = new Packages.java.lang.String("Hello world")
```

Working with Java Arrays

When any Java method creates an array and you reference that array in JavaScript, you are working with a JavaArray. For example, the following code creates the JavaArray x with ten elements of type int:

```
x = java.lang.reflect.Array.newInstance(java.lang.Integer, 10)
```
Like the JavaScript Array object, JavaArray has a length property that returns the number of elements in the array. Unlike Array.length, JavaArray.length is a read-only property, because the number of elements in a Java array are fixed at the time of creation.

Package and Class References

Simple references to Java packages and classes from JavaScript create the JavaPackage and JavaClass objects. In the earlier example about the Redwood corporation, for example, the reference Packages.redwood is a JavaPackage object. Similarly, a reference such as java.lang.String is a JavaClass object.

Most of the time, you don't have to worry about the JavaPackage and JavaClass objects—you just work with Java packages and classes, and LiveConnect creates these objects transparently.

In JavaScript 1.3 and earlier, JavaClass objects are not automatically converted to instances of java.lang.Class when you pass them as parameters to Java methods--you must create a wrapper around an instance of java.lang.Class. In the following example, the forName method creates a wrapper object theClass, which is then passed to the newInstance method to create an array.

```
// JavaScript 1.3
theClass = java.lang.Class.forName("java.lang.String")
theArray = java.lang.reflect.Array.newInstance(theClass, 5)
```
In JavaScript 1.4 and later, you can pass a JavaClass object directly to a method, as shown in the following example:

```
// JavaScript 1.4
theArray =
```

```
java.lang.reflect.Array.newInstance(java.lang.String, 5)
```

Arguments of Type Char

In JavaScript 1.4 and later, you can pass a one-character string to a Java method that requires an argument of type char. For example, you can pass the string "H" to the Character constructor as follows:

```
c = new java.lang.Character("H")
```

In JavaScript 1.3 and earlier, you must pass such methods an integer that corresponds to the Unicode value of the character. For example, the following code also assigns the value "H" to the variable c:

```
c = new java.lang.Character(72)
```

Handling Java Exceptions in JavaScript

When Java code fails at run time, it throws an exception. If your JavaScript code accesses a Java data member or method and fails, the Java exception is passed on to JavaScript for you to handle. Beginning with JavaScript 1.4, you can catch this exception in a try...catch statement.

For example, suppose you are using the Java forName method to assign the name of a Java class to a variable called theClass. The forName method throws an exception if the value you pass it does not evaluate to the name of a Java class. Place the forName assignment statement in a try block to handle the exception, as follows:

```
function getClass(javaClassName) {
  try {
      var theClass = java.lang.Class.forName(javaClassName);
  } catch (e) {
      return ("The Java exception is " + e);
  }
  return theClass
}
```

In this example, if javaClassName evaluates to a legal class name, such as "java.lang.String", the assignment succeeds. If javaClassName evaluates to an invalid class name, such as "String", the getClass function catches the exception and returns something similar to the following:

```
The Java exception is java.lang.ClassNotFoundException: String
```

See *Exception Handling Statements* (on page 205), for more information about JavaScript exceptions.

Java to JavaScript Communication

If you want to use JavaScript objects in Java, you must import the netscape.javascript package into your Java file. This package defines the following classes:

- netscape.javascript.JSObject allows Java code to access JavaScript methods and properties.

- netscape.javascript.JSException allows Java code to handle JavaScript errors.

Starting with JavaScript 1.2, these classes are delivered in a `.jar` file; in previous versions of JavaScript, these classes are delivered in a `.zip` file. See *JSException and JSObject Classes* (on page 208), for more information about these classes.

To access the LiveConnect classes, place the `.jar` or `.zip` file in the CLASSPATH of the JDK compiler in either of the following ways:

- Create a CLASSPATH environment variable to specify the path and name of `.jar` or `.zip` file.

- Specify the location of `.jar` or `.zip` file when you compile by using the -classpath command line parameter.

For example, in Navigator 4.0 for Windows NT, the classes are delivered in the `java40.jar` file in the Program\Java\Classes directory beneath the Navigator directory.

To specify an environment variable in Windows NT:

1. Double-click the **System** icon in the Control Panel.

2. Create a user environment variable called CLASSPATH with a value similar to the following:

        D:\Navigator\Program\Java\Classes\java40.jar

See the Sun JDK documentation for more information about CLASSPATH.

Note: Because Java is a strongly typed language and JavaScript is weakly typed, the JavaScript runtime engine converts argument values into the appropriate data types for the other language when you use LiveConnect. See *Data Type Conversions* (on page 197), for complete information.

Using the LiveConnect Classes

All JavaScript objects appear within Java code as instances of netscape.javascript.JSObject. When you call a method in your Java code, you can pass it a JavaScript object as one of its argument. To do so, you must define the corresponding formal parameter of the method to be of type JSObject.

Also, any time you use JavaScript objects in your Java code, you should put the call to the JavaScript object inside a `try...catch` statement that handles exceptions of type netscape.javascript.JSException. This allows your Java code to handle errors in JavaScript code execution that appear in Java as exceptions of type JSException.

Accessing JavaScript with JSObject

For example, suppose you are working with the Java class called JavaDog. As shown in the following code, the JavaDog constructor takes the JavaScript object jsDog, which is defined as type JSObject, as an argument:

```
import netscape.javascript.*;
public class JavaDog
{
   public String dogBreed;
   public String dogColor;
   public String dogSex;

   // define the class constructor
   public JavaDog(JSObject jsDog)
   {
       // use try...catch to handle JSExceptions here
       this.dogBreed = (String)jsDog.getMember("breed");
       this.dogColor = (String)jsDog.getMember("color");
       this.dogSex = (String)jsDog.getMember("sex");
   }
}
```

Note: The getMember method of JSObject is used to access the properties of the JavaScript object. The previous example uses getMember to assign the value of the JavaScript property jsDog.breed to the Java data member JavaDog.dogBreed.

A more realistic example would place the call to getMember inside a `try...catch` statement to handle errors of type JSException. See *Handling JavaScript Exceptions in Java* (on page 195), for more information.

To get a better sense of how getMember works, look at the definition of the custom JavaScript object Dog:

```
function Dog(breed,color,sex) {
   this.breed = breed
   this.color = color
   this.sex = sex
}
```

You can create a JavaScript instance of Dog called gabby as follows:

```
gabby = new Dog("lab","chocolate","female")
```

If you evaluate gabby.color, you can see that it has the value "chocolate". Now suppose you create an instance of JavaDog in your JavaScript code by passing the gabby object to the constructor as follows:

```
javaDog = new Packages.JavaDog(gabby)
```

If you evaluate javaDog.dogColor, you can see that it also has the value "chocolate", because the getMember method in the Java constructor assigns dogColor the value of gabby.color.

Handling JavaScript Exceptions in Java

When JavaScript code called from Java fails at run time, it throws an exception. If you are calling the JavaScript code from Java, you can catch this exception in a `try...catch` statement. The JavaScript exception is available to your Java code as an instance of netscape.javascript.JSException.

JSException is a Java wrapper around any exception type thrown by JavaScript, similar to the way that instances of JSObject are wrappers for JavaScript objects. Use JSException when you are evaluating JavaScript code in Java.

When you are evaluating JavaScript code in Java, the following situations can cause run-time errors:

- The JavaScript code is not evaluated, either due to a JavaScript compilation error or to some other error that occurred at run time.

  The JavaScript interpreter generates an error message that is converted into an instance of JSException.

- Java successfully evaluates the JavaScript code, but the JavaScript code executes an unhandled throw statement.

  JavaScript throws an exception that is wrapped as an instance of `JSException`. Use the `getWrappedException` method of `JSException` to unwrap this exception in Java.

For example, suppose the Java object eTest evaluates the string jsCode that you pass to it. You can respond to either type of run-time error the evaluation causes by implementing an exception handler such as the following:

```
import netscape.javascript.JSObject;
import netscape.javascript.JSException;
```

```
public class eTest {
  public static Object doit(JSObject obj, String jsCode) {
      try {
            obj.eval(jsCode);
      } catch (JSException e) {
            if (e.getWrappedException()==null)
            return e;
            return e.getWrappedException();
      }
      return null;
  }
}
```

In this example, the code in the try block attempts to evaluate the string jsCode that
you pass to it. Let's say you pass the string "myFunction()" as the value of jsCode. If
myFunction is not defined as a JavaScript function, the JavaScript interpreter cannot
evaluate jsCode. The interpreter generates an error message, the Java handler catches
the message, and the doit method returns an instance of
netscape.javascript.JSException.

However, suppose myFunction is defined in JavaScript as follows:

```
function myFunction() {
  try {
        if (theCondition == true) {
            return "Everything's ok";
        } else {
            throw "JavaScript error occurred" ;
        }
  } catch (e) {
        if (canHandle == true) {
            handleIt();
        } else {
            throw e;
        }
  }
}
```

If theCondition is false, the function throws an exception. The exception is caught in
the JavaScript code, and if canHandle is true, JavaScript handles it. If canHandle is
false, the exception is rethrown, the Java handler catches it, and the doit method
returns a Java string:

```
JavaScript error occurred
```

See *Exception Handling Statements* (on page 205), for complete information about JavaScript exceptions.

Backward Compatibility

In JavaScript 1.3 and earlier versions, the JSException class had three public constructors that optionally took a string argument, specifying the detail message or other information for the exception. The getWrappedException method was not available.

Use a `try...catch` statement such as the following to handle LiveConnect exceptions in JavaScript 1.3 and earlier versions:

```
try {
  global.eval("foo.bar = 999;");
} catch (Exception e) {
  if (e instanceof JSException) {
      jsCodeFailed()";
  } else {
      otherCodeFailed();
  }
}
```

In this example, the eval statement fails if foo is not defined. The catch block executes the jsCodeFailed method if the eval statement in the try block throws a JSException; the otherCodeFailed method executes if the try block throws any other error.

Data Type Conversions

Because Java is a strongly typed language and JavaScript is weakly typed, the JavaScript runtime engine converts argument values into the appropriate data types for the other language when you use LiveConnect.

JavaScript to Java Conversions

When you call a Java method and pass it parameters from JavaScript, the data types of the parameters you pass in are converted according to the rules described in the following sections:

- *Number Values* (on page 198)

- *Boolean Values* (on page 199)

- *String Values* (on page 200)

- *Undefined Values* (on page 200)
- *Null Values* (on page 201)
- *JavaArray and JavaObject Objects* (on page 201)
- *JavaClass Objects* (on page 203)
- *Other JavaScript Objects* (on page 203)

The return values of methods of netscape.javascript.JSObject are always converted to instances of java.lang.Object. The rules for converting these return values are also described in these sections.

For example, if JSObject.eval returns a JavaScript number, you can find the rules for converting this number to an instance of java.lang.Object in *Number Values* (on page 198).

Number Values

When you pass JavaScript number types as parameters to Java methods, Java converts the values according to the rules described in the following table:

Table 10: Number type conversion rules

Java parameter type	Conversion rules
double	The exact value is transferred to Java without rounding and without a loss of magnitude or sign.
lava.lang.Double java.lang.Object	A new instance of java.lang.Double is created, and the exact value is transferred to Java without rounding and without a loss of magnitude or sign.
float	Values are rounded to float precision. Values that are unrepresentably large or small are rounded to +infinity or -infinity.
byte char int long short	Values are rounded using round-to-negative-infinity mode. Values that are unrepresentably large or small result in a run-time error. NaN values are converted to zero.
java.lang.String	Values are converted to strings. For example: 237 becomes "237"
boolean	0 and NaN values are converted to false. Other values are converted to true.

When a JavaScript number is passed as a parameter to a Java method that expects an instance of java.lang.String, the number is converted to a string. Use the == operator to compare the result of this conversion with other string values.

Boolean Values

When you pass JavaScript Boolean types as parameters to Java methods, Java converts the values according to the rules described in the following table:

Table 11: Boolean type conversion rules

Java parameter type	Conversion rules
`boolean`	All values are converted directly to the Java equivalents.
lava.lang.Boolean java.lang.Object	A new instance of java.lang.Boolean is created. Each parameter creates a new instance, not one instance with the same primitive value.
java.lang.String	Values are converted to strings. For example: true becomes "true" false becomes "false"

`byte      char`
`double    float`
`int       long`
`short`
	true becomes 1
	
	false becomes 0

When a JavaScript Boolean is passed as a parameter to a Java method that expects an instance of java.lang.String, the Boolean is converted to a string. Use the == operator to compare the result of this conversion with other string values.

String Values

When you pass JavaScript string types as parameters to Java methods, Java converts the values according to the rules described in the following table:

Table 12: String type conversion rules

Java parameter type	Conversion rules
`lava.lang.String` `java.lang.Object`	JavaScript 1.4: A JavaScript string is converted to an instance of java.lang.String with a Unicode value. JavaScript 1.3 and earlier: A JavaScript string is converted to an instance of java.lang.String with an ASCII value.

`byte` `double`
`float` `int`
`long` `short`
	All values are converted to the appropriate numbers.

`char`	JavaScript 1.4: One-character strings are converted to Unicode characters. All other values are converted to numbers. JavaScript 1.3 and earlier: All values are converted to numbers.
`boolean`	The empty string becomes false. All other values become true.

Undefined Values

When you pass undefined JavaScript values as parameters to Java methods, Java converts the values according to the rules described in the following table:

Table 13: Undefined type conversion rules

Java parameter type	Conversion rules
`lava.lang.String` `java.lang.Object`	The value is converted to an instance of java.lang.String whose value is the string "undefined".
`boolean`	The value becomes false.
`double` `float`	The value becomes NaN.

Java parameter type	Conversion rules
byte char int long Short	The value becomes 0.

The undefined value conversion is possible in JavaScript 1.3 and later versions only. Earlier versions of JavaScript do not support undefined values.

When a JavaScript undefined value is passed as a parameter to a Java method that expects an instance of java.lang.String, the undefined value is converted to a string. Use the == operator to compare the result of this conversion with other string values.

Null Values

When you pass null JavaScript values as parameters to Java methods, Java converts the values according to the rules described in the following table:

Table 14: Null value conversion rules

Java parameter type	Conversion rules
Any class, any interface type	The value becomes null.
byte char double float int long short	The value becomes 0.
boolean	The value becomes false.

JavaArray and JavaObject Objects

In most situations, when you pass a JavaScript JavaArray or JavaObject as a parameter to a Java method, Java simply unwraps the object; in a few situations, the object is coerced into another data type according to the rules described in the following table:

Table 15: JavaArray and JavaObject type conversion rules

Java parameter type	Conversion rules
Any interface or class that is assignment-compatible with the unwrapped object.	The object is unwrapped.

Java parameter type	Conversion rules
`java.lang.String`	The object is unwrapped, the `toString` method of the unwrapped Java object is called, and the result is returned as a new instance of `java.lang.String`.

`byte`   `char`
`double`   `float`
`int`   `long`
`short`
	The object is unwrapped, and either of the following situations occur:
	
	• If the unwrapped Java object has a `doubleValue` method, the `JavaArray` or JavaObject is converted to the value returned by this method.
	
	• If the unwrapped Java object does not have a `doubleValue` method, an error occurs.

`boolean`	In JavaScript 1.3 and later versions, the object is unwrapped and either of the following situations occur: • If the object is null, it is converted to false. • If the object has any other value, it is converted to true. In JavaScript 1.2 and earlier versions, the object is unwrapped and either of the following situations occur: • If the unwrapped object has a `booleanValue` method, the source object is converted to the return value. • If the object does not have a `booleanValue` method, the conversion fails.

An interface or class is assignment-compatible with an unwrapped object if the unwrapped object is an instance of the Java parameter type. That is, the following statement must return true:

```
unwrappedObject instanceof parameterType
```

JavaClass Objects

When you pass a JavaScript JavaClass object as a parameter to a Java method, Java converts the object according to the rules described in the following table:

Table 16: JavaClass object conversion rules

Java parameter type	Conversion rules
`java.lang.Class`	The object is unwrapped.
`java.lang.JSObject` `java.lang.Object`	The JavaClass object is wrapped in a new instance of java.lang.JSObject.
`java.lang.String`	The object is unwrapped, the toString method of the unwrapped Java object is called, and the result is returned as a new instance of java.lang.String.
`boolean`	In JavaScript 1.3 and later versions, the object is unwrapped and either of the following situations occur: • If the object is null, it is converted to false. • If the object has any other value, it is converted to true. In JavaScript 1.2 and earlier versions, the object is unwrapped and either of the following situations occur: • If the unwrapped object has a booleanValue method, the source object is converted to the return value. • If the object does not have a booleanValue method, the conversion fails.

Other JavaScript Objects

When you pass any other JavaScript object as a parameter to a Java method, Java converts the object according to the rules described in the following table:

Table 17: Other JavaScript Object Conversion Rules

Java parameter type	Conversion rules
`java.lang.JSObject` `java.lang.Object`	The object is wrapped in a new instance of java.lang.JSObject.
`java.lang.String`	The object is unwrapped, the toString method of the unwrapped Java object is called, and the result is returned as a new instance of java.lang.String.

Java parameter type	Conversion rules
byte char Double float int long short	The object is converted to an appropriate value using the logic of the ToPrimitive operator. The *PreferredType* hint used with this operator is Number.
boolean	In JavaScript 1.3 and later versions, the object is unwrapped and either of the following situations occur: • If the object is null, it is converted to false. • If the object has any other value, it is converted to true. In JavaScript 1.2 and earlier versions, the object is unwrapped and either of the following situations occur: • If the unwrapped object has a `booleanValue` method, the source object is converted to the return value. • If the object does not have a `booleanValue` method, the conversion fails.

Java to JavaScript Conversions

Values passed from Java to JavaScript are converted as follows:

• Java byte, char, short, int, long, float, and double are converted to JavaScript numbers.

• A Java Boolean is converted to a JavaScript Boolean.

• An object of class `netscape.javascript.JSObject` is converted to the original JavaScript object.

• Java arrays are converted to a JavaScript pseudo-Array object; this object behaves just like a JavaScript Array object: you can access it with the syntax `arrayName[index]` (where index is an integer), and determine its length with `arrayName.length`.

• A Java object of any other class is converted to a JavaScript wrapper, which can be used to access methods and fields of the Java object:

  • Converting this wrapper to a string calls the toString method on the original object.

  • Converting to a number calls the doubleValue method, if possible, and fails otherwise.

- Converting to a Boolean in JavaScript 1.3 and later versions returns false if the object is null, and true otherwise.

- Converting to a Boolean in JavaScript 1.2 and earlier versions calls the booleanValue method, if possible, and fails otherwise.

Note: Instances of `java.lang.Double` and `java.lang.Integer` are converted to JavaScript objects, not to JavaScript numbers. Similarly, instances of `java.lang.String` are also converted to JavaScript objects, not to JavaScript strings.

Java String objects also correspond to JavaScript wrappers. If you call a JavaScript method that requires a JavaScript string and pass it this wrapper, you'll get an error. Instead, convert the wrapper to a JavaScript string by appending the empty string to it, as shown here:

```
var JavaString = JavaObj.methodThatReturnsAString();
var JavaScriptString = JavaString + "";
```

Exception Handling Statements

You can throw and catch exceptions using the `throw` and `try...catch` statements. You also use the `try...catch` statement to handle Java exceptions. See *Handling Java Exceptions in JavaScript* (on page 192), and *Handling JavaScript Exceptions in Java* (on page 195), for information.

The Throw Statement

Use the `throw` statement to throw an exception. When you throw an exception, you specify an expression containing the value of the exception:

```
throw expression
```
The following code throws several exceptions.

```
throw "Error2"  // generates an exception with a string value
throw 42        // generates an exception with the value 42
throw true      // generates an exception with the value true
```
You can specify an object when you throw an exception. You can then reference the object's properties in the `catch` block. The following example creates an object `myUserException` of type `UserException` and uses it in a `throw` statement.

```
// Create an object type UserException
function UserException (message) {
  this.message=message
  this.name="UserException"
}
```

```
// Create an instance of the object type and throw it
myUserException=new UserException("Value too high")
throw myUserException
```

The Try...Catch Statement

The `try...catch` statement marks a block of statements to try, and specifies a response should an exception be thrown. If an exception is thrown, the `try...catch` statement catches it.

Note: There are some exceptions that are not caught with the `try...catch` statement. This includes the HTTP errors, which are thrown by the HTTP server.

The `try...catch` statement consists of the following:

*   A `try` block, which contains one or more statements.

*   A `catch` block, containing statements that specify what to do if an exception is thrown in the `try` block.

    That is, you want the `try` block to succeed, and if it does not succeed, you want control to pass to the `catch` block.

    If any statement within the `try` block (or in a function called from within the `try` block) throws an exception, control immediately shifts to the `catch` block. If no exception is thrown in the `try` block, the `catch` block is skipped.

*   The `finally` block executes after the `try` and `catch` blocks execute but before the statements following the `try...catch` statement.

The following example uses a `try...catch` statement. The example calls a function that retrieves a month name from an array based on the value passed to the function. If the value does not correspond to a month number (1-12), an exception is thrown with the value `"InvalidMonthNo"` and the statements in the `catch` block set the `monthName` variable to `"unknown"`.

```
function getMonthName (mo) {
  // Adjust month number for array index (1=Jan, 12=Dec)
  mo=mo-1
  var months=new Array ("Jan","Feb","Mar","Apr","May",
      "Jun","Jul","Aug","Sep","Oct","Nov","Dec")
  if (months[mo] != null) {
      return months[mo]
  } else {
      throw "InvalidMonthNo"
  }
}
```

```
try {
  // statements to try
  monthName=getMonthName(myMonth)
  // function could throw exception
}
catch (e) {
  monthName="unknown"
  logMyErrors(e)
  // pass exception object to error handler
}
```

The Catch Block

Use the `try...catch` statement's `catch` block (*recovery* block) to execute error-handling code. A `catch` block looks as follows:

```
catch (catchID) {
   statements
}
```

Every `catch` block specifies an identifier (`catchID` in the preceding syntax) that holds the value specified by the `throw` statement; you can use this identifier to get information about the exception that was thrown. JavaScript creates this identifier when the `catch` block is entered; the identifier lasts only for the duration of the `catch` block; after the `catch` block finishes executing, the identifier is no longer available.

For example, the following code throws an exception. When the exception occurs, control transfers to the `catch` block.

```
try {
   throw "myException" // generates an exception
}
catch (e) {
  // statements to handle any exceptions
  logMyErrors(e) // pass exception object to error handler
}
```

The Finally Block

The `finally` block contains statements to execute after the `try` and `catch` blocks execute but before the statements following the `try...catch` statement. The `finally` block executes whether or not an exception is thrown. If an exception is thrown, the statements in the `finally` block execute even if no `catch` block handles the exception.

You can use the `finally` block to make your script fail gracefully when an exception occurs; for example, you may need to release a resource that your script has tied up. The following example opens a file and then executes statements that use the file (server-side JavaScript allows you to access files). If an exception is thrown while the file is open, the `finally` block closes the file before the script fails.

```
try {
  openMyFile()  // tie up a resource
  writeMyFile(theData)
}
finally {
  closeMyFile() // always close the resource
}
```

Nesting Try...Catch Statements

You can nest one or more `try...catch` statements. If an inner `try...catch` statement does not have a `catch` block, the enclosing `try...catch` statement's `catch` block is checked for a match.

JSException and JSObject Classes

This section describes the JSException and JSObject Classes.

JSException Class

The public class JSException extends RuntimeException.

```
java.lang.Object
  |
  +---java.lang.Throwable
        |
        +---java.lang.Exception
             |
             +---java.lang.RuntimeException
                  |
                  +----netscape.javascript.JSException
```

Description

JSException is an exception that is thrown when JavaScript code returns an error.

Constructor Summary

The netscape.javascript.JSException class has the following constructors:

Table 18: JSException constructors

Constructor	Description
JSException	Deprecated constructors optionally let you specify a detail message and other information.

Method Summary

The netscape.javascript.JSException class has the following method:

Table 19: JSException methods

Method	Description
GetWrappedException	Instance method getWrappedException.

The following sections show the declaration and usage of the constructors and method.

Backward Compatibility

JavaScript 1.1 through 1.3. JSException had three public constructors that optionally took a string argument, specifying the detail message or other information for the exception. The getWrappedException method was not available.

JSException Constructor

Constructors, deprecated in JavaScript 1.4. Construct a JSException with an optional detail message.

Declaration

```
1. public JSException()
2. public JSException(String s)
3. public JSException(String s,
        String filename,
        int lineno,
        String source,
        int tokenIndex)
```

Arguments

Table 20: JSException arguments

Argument Name	Description
s	The detail message.
filename	The URL of the file where the error occurred, if possible.
lineno	The line number if the file, if possible.
source	The string containing the JavaScript code being evaluated.
tokenIndex	The index into the source string where the error occurred.

GetWrappedException

Instance method getWrappedException.

Declaration

```
public Object getWrappedException()
```

JSObject Class

The public final class netscape.javascript.JSObject extends Object.

```
java.lang.Object
   |
   +----netscape.javascript.JSObject
```

Description

JavaScript objects are wrapped in an instance of the class netscape.javascript.JSObject and passed to Java. JSObject allows Java to manipulate JavaScript objects.

When a JavaScript object is sent to Java, the runtime engine creates a Java wrapper of type JSObject; when a JSObject is sent from Java to JavaScript, the runtime engine unwraps it to its original JavaScript object type. The JSObject class provides a way to invoke JavaScript methods and examine JavaScript properties.

Any JavaScript data brought into Java is converted to Java data types. When the JSObject is passed back to JavaScript, the object is unwrapped and can be used by JavaScript code. See *Data Type Conversions* (on page 197), for more information about data type conversions.

Method Summary

The netscape.javascript.JSObject class has the following methods:

Table 21: JSObject methods

Method	Description
`call`	Calls a JavaScript method.
`equals`	Determines if two JSObject objects refer to the same instance.
`eval`	Evaluates a JavaScript expression.
`getMember`	Retrieves the value of a property of a JavaScript object.
`getSlot`	Retrieves the value of an array element of a JavaScript object.
`removeMember`	Removes a property of a JavaScript object.
`setMember`	Sets the value of a property of a JavaScript object.
`setSlot`	Sets the value of an array element of a JavaScript object.
`toString`	Converts a JSObject to a string.
Static Method	
getWindow	Gets a JSObject for the window containing the given applet.

The following sections describe the declaration and usage of these methods.

call

Method. Calls a JavaScript method. Equivalent to

```
"this.methodName(args[0], args[1], ...)" in JavaScript.
```

Declaration

```
public Object call(String methodName, Object args[])
```

equals

Method. Determines if two JSObject objects refer to the same instance.

Overrides: equals in class java.lang.Object

Declaration

```
public boolean equals(Object obj)
```

Backward compatibility

JavaScript 1.3. In JavaScript 1.3 and earlier versions, you can use either the equals method of java.lang.Object or the == operator to evaluate two JSObject objects.

eval

Method. Evaluates a JavaScript expression. The expression is a string of JavaScript source code that will be evaluated in the context given by "this".

Declaration

```
public Object eval(String s)
```

getMember

Method. Retrieves the value of a property of a JavaScript object. Equivalent to "this.name" in JavaScript.

Declaration

```
public Object getMember(String name)
```

getSlot

Method. Retrieves the value of an array element of a JavaScript object. Equivalent to "this[index]" in JavaScript.

Declaration

```
public Object getSlot(int index)
```

getWindow

Static method. Returns a JSObject for the window containing the given applet. This method is useful in client-side JavaScript only.

Declaration

```
public static JSObject getWindow(Applet applet)
```

removeMember

Method. Removes a property of a JavaScript object.

Declaration

```
public void removeMember(String name)
```

setMember

Method. Sets the value of a property of a JavaScript object. Equivalent to "this.name = value" in JavaScript.

Declaration

```
public void setMember(String name, Object value)
```

setSlot

Method. Sets the value of an array element of a JavaScript object. Equivalent to "this[index] = value" in JavaScript.

Declaration

```
public void setSlot(int index, Object value)
```

toString

Method. Converts a JSObject to a String.

Overrides: toString in class java.lang.Object

Declaration

```
public String toString()
```

Load Engine CLI

C**Appendix C**

You can use the following CLI commands to run the Load Generator and then run a test simulating the load.

Note: The script name must always appear first in the command line. The order of the other parameters is not significant.

Option	Effect	Default
`<scenarios filename>`	Agenda or template name. This option must always appear first in the command line.	
`-r:<reports interval>`	Interval of reporting.	100,000 milliseconds (100 seconds)
`-b`	Batch mode.	Interactive (menu) mode
`-c:<clients>`	Virtual clients number.	1
`-t:<abs|rel|off>`	Timestamp type switcher: • Abs – absolute (PC time) • rel – relative (from application start) • off – no timestamps (suitable for functional tests)	Absolute time stamps
`-a`	Append log file if one exists.	Overwrite
`-u`	Attach auto-generated unique file name suffix: timestamp and/or PID.	No suffix
`-m`	Messages on. This option is only applicable in Interactive and Batch mode.	off
`-d`	Duplicate trace to console.	Silent (no console output)
`-s:<stats_file>`	Statistics file name. The Statistics file will appear in the log directory. This option is only applicable in Interactive and Batch mode.	No statistics output

The following is an example using the CLI commands:

```
wlloadengine.exe E:\google.js –r:10 -b -c:2 –d –m –
s:google.stats
```

```
Where:
```

- `Wlloadengine.exe` runs the Load Generator engine.

- `E:\google.js` is the script being recorded.

- `–r:10` indicates a 10 milliseconds interval of reporting.

- `-b` indicates that the Load Generator is working in batch mode.

- `-c:2` indicates that there are two virtual clients.

- `-d` indicates to duplicate a trace to the Console, creating Console output.

- `–m` indicates that the Messages features is on.

- `-s:google.stat` indicates the name of the statistics output file that appears in the log directory.

Index

A

A Typical Web Page And The Corresponding Parse Tree ▪ 128

Accessing Java Objects From JavaScript Agendas ▪ 97

Accessing JavaScript With JSObject ▪ 194

Accessing the JavaScript code within the Agenda Tree ▪ 15

Activating ActiveX Objects From A JavaScript Agenda ▪ 110

ActiveX Object ▪ 108
    ActiveXObject() constructor ▪ 111
    ActiveXObject() constructor (for RDS) ▪ 118
    automatic conversion to JavaScript data types ▪ 114
    casting functions ▪ 115
    remote data service (RDS) support ▪ 117
    Remote Data Service (RDS) Support example ▪ 119

ActiveX Object Interfaces ▪ 109
    IDispatch ▪ 109
    ITypeInfo ▪ 109
    IUnknown ▪ 109

ActiveX Object Limitations ▪ 121

ActiveXObject() (Constructor) ▪ 111

ActiveXObject() Constructor for RDS ▪ 118

Adding
    JavaScript object nodes, to Agenda Tree ▪ 19

Adding a CopyFile() Function ▪ 25

Adding an IncludeFile() Function ▪ 21

Adding Input File Commands to your Agenda Code ▪ 32

Adding JavaScript Object Nodes ▪ 19

Adding New Bookstore Data ▪ 82

Adding the wlOutputFile Object ▪ 27

Agenda Execution Sequence ▪ 11

Agenda Program Structure ▪ 8

Agenda Tree ▪ 7
    nodes ▪ 7
    structure ▪ 7

Agenda Tree Nodes ▪ 7

Agenda Tree structure ▪ 7

B

Backward Compatibility ▪ 197, 209

Basic Execution Sequence ▪ 11

Boolean Values ▪ 199

Browser Configuration
    executing commands ▪ 51
    search order precedence ▪ 62
    transaction-specific ▪ 51

Browser Emulation ▪ 40

C

Cache, SSL, Cleared at Round's End ▪ 13

call ▪ 211

Calling A WebLOAD API From A Java Application ▪ 104

CBool() Function ▪ 115

CByte() Function ▪ 115

CDbl() Function ▪ 115

CFlt() Function ▪ 115

Changing Bookstore Data ▪ 80

CInt() Function ▪ 115

CLASSPATH Environment Variable ▪ 95

Cleanup at the End of Each Round ▪ 13

ClientNum ▪ 65

ClientNum Variable ▪ 65

Clients
    example, identifying clients ▪ 67
    mixed ▪ 14
    scheduled ▪ 13
    unique number ▪ 65

Agendas
(see index entries)