

# Server-Side JavaScript Guide

Version 1.2

Netscape Communications Corporation ("Netscape") and its licensors retain all ownership rights to the software programs offered by Netscape (referred to herein as "Software") and related documentation. Use of the Software and related documentation is governed by the license agreement accompanying the Software and applicable copyright law.

Your right to copy this documentation is limited by copyright law. Making unauthorized copies, adaptations, or compilation works is prohibited and constitutes a punishable violation of the law. Netscape may revise this documentation from time to time without notice.

THIS DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. IN NO EVENT SHALL NETSCAPE BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY KIND ARISING FROM ANY ERROR IN THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION ANY LOSS OR INTERRUPTION OF BUSINESS, PROFITS, USE, OR DATA.

The Software and documentation are copyright ©1994-1998 Netscape Communications Corporation. All rights reserved.

Netscape, Netscape Navigator, Netscape Certificate Server, Netscape DevEdge, Netscape FastTrack Server, Netscape ONE, SuiteSpot and the Netscape N and Ship's Wheel logos are registered trademarks of Netscape Communications Corporation in the United States and other countries. Other Netscape logos, product names, and service names are also trademarks of Netscape Communications Corporation, which may be registered in other countries. JavaScript is a trademark of Sun Microsystems, Inc. used under license for technology invented and implemented by Netscape Communications Corporation. Other product and brand names are trademarks of their respective owners.

The downloading, exporting, or reexporting of Netscape software or any underlying information or technology must be in full compliance with all United States and other applicable laws and regulations. Any provision of Netscape software or documentation to the U.S. Government is with restricted rights as described in the license agreement accompanying Netscape software.



Recycled and Recyclable Paper

Version 1.2

©1998 Netscape Communications Corporation. All Rights Reserved

Printed in the United States of America. 00 99 98 5 4 3 2 1

Netscape Communications Corporation, 501 East Middlefield Road, Mountain View, CA 94043

# New Features in this Release

This section contains the following topics:

- Changes to Server-Side JavaScript
- Changes to Core JavaScript 1.2
- Upgrading from an Earlier Release
- Backward Compatibility with Earlier Releases

## Changes to Server-Side JavaScript

With the release of the 3.x versions of Netscape web servers, Netscape LiveWire 1.01 is fully integrated into the web servers. Since LiveWire database connectivity is now integrated as the LiveWire Database Service portion of server-side JavaScript, developers do not have to install LiveWire as a separate product. Simply turn on the JavaScript support in the Administration Server to make the necessary components available.

The following improvements have been made to server-side JavaScript:

- **Support for core JavaScript 1.2.** See “Changes to Core JavaScript 1.2” on page 5.
- **New Lock class.** The `Lock` class allows safe sharing of information with multiple incoming requests. See “Sharing Objects Safely with Locking” on page 279.
- **New SendMail class.** The `SendMail` class lets you generate email from JavaScript. See “Mail Service” on page 287.
- **Property value types.** Property values can be of any data type, rather than just strings, for the `project`, `server`, and `request` objects. In particular, you can now use `project` and `server` objects to store references to other objects. See “The project Object” on page 260, “The server Object” on page 261, and “The request Object” on page 249.

- **Direct access to HTTP request and response headers.** See “Request and Response Manipulation” on page 302.
- **Access Java classes.** You can access Java classes using LiveConnect. See Chapter 21, “LiveConnect Overview.”
- **IIOP.** You can access legacy applications using IIOP. See Chapter 22, “Accessing CORBA Services.”
- **Multiple simultaneous connections.** LiveWire has support for multiple simultaneous connections to multiple databases. See Chapter 15, “Connecting to a Database.”
- **Span multiple client requests.** LiveWire database connections and transactions (and the objects used with them) can span multiple client requests instead of having to be restarted for each request. See “Individual Database Connections” on page 323 and “Managing Transactions” on page 348.
- **Stored procedures.** LiveWire has support for stored procedures. See “Calling Stored Procedures” on page 354.
- **ODBC support.** LiveWire has ODBC support under Unix. See “Supported Database Clients and ODBC Drivers” on page 372.
- **Multithreading.** LiveWire supports multithreading of Informix, Oracle, and Sybase database client libraries for improved performance and scalability. See “Supported Database Clients and ODBC Drivers” on page 372. This support is available only if the underlying platform supports multithreading. For information on which platforms support it, see *Enterprise Server 3.x Release Notes*

# Changes to Core JavaScript 1.2

JavaScript version 1.2 provides the following new features and enhancements:

- **Create objects with initializers.** You can create objects using object initializers. See “Object Literals” on page 83 and “Creating New Objects” on page 141.
- **Changes to arrays.** Array objects can be created using literal notation. See “Creating an Array” on page 148.
- **Regular expressions.** Regular expressions are patterns used to match character combinations in strings. You create a regular expression as an object that has methods used to execute a match against a string. You can also pass the regular expression as an argument to the `String` methods `match`, `replace`, `search`, and `split`. The `RegExp` object has properties most of which are set when a match is successful, such as `lastMatch` which specifies the last successful match. The `Array` object has new properties that provide information about a successful match such as `input` which specifies the original input string against which the match was executed. See Chapter 6, “Regular Expressions” for information.
- **New top-level functions `Number` and `String`.** The `Number` function converts an object to a number. The `String` function converts an object to a string. See “Number and String Functions” on page 137.
- **Changes to `eval`.** `eval` is no longer a method of individual objects; it is available only as a top-level function. See “eval Function” on page 135.
- **New and changed operators.**
  - The new `delete` operator deletes an object, an object’s property, or an element at a specified index in an array. See “delete” on page 98.
  - If the `<SCRIPT>` tag uses `LANGUAGE=JavaScript1.2`, the equality operators `==` and `!=` do not attempt to convert operands from one type to another, and always compare identity of like-typed operands. See “Comparison Operators” on page 90.

- **New and changed statements.**

- The `break` and `continue` statements can now be used with the new `labeled` statement. See “break Statement” on page 126 and “continue Statement” on page 127.
- `do...while` repeats a loop until the test condition evaluates to false. See “do...while Statement” on page 124.
- `label` allows the program to break outside nested loops or to continue a loop outside the current loop. See “label Statement” on page 125.
- `switch` allows the program to test several conditions easily. See “switch Statement” on page 121.

See the *Server-Side JavaScript Reference* for information on additional features.

## Upgrading from an Earlier Release

If you have previously installed a 2.0 version of a Netscape web server, you should migrate the server settings when you install a 3.x version of a Netscape web server. For information on how to install the server and migrate settings, see the administrator's guide for your web server. If you do not migrate old server settings when you install the server, you can migrate them later, using the “Migrate from previous version” link on the Netscape Server Administration Page. Information on this link is also in the administrator's guide for your web server.

If you have previously created JavaScript applications using LiveWire 1.x, you should be aware of the following changes that occur when you upgrade to 3.x and migrate old server settings:

- **Settings preserved.** If the 2.x server had LiveWire turned on, the 3.x server will have server-side JavaScript turned on. Whether or not the Application Manager requires a password is also preserved. For more information, see “Configuration Information” on page 49.

- **Config file upgraded and renamed.** The existing `livewire.conf` file is upgraded and renamed `jsa.conf`. The new `jsa.conf` file points to the new Application Manager and the new sample applications. It also contains entries for all other applications you had in the old `livewire.conf` file. For details of the `jsa.conf` file, see “Application Manager Details” on page 71.
- **Applications are not moved.** Upgrading server settings does not move your applications nor does it recompile them for use with the 3.x web server. If your existing applications are in the `LiveWire/docs` directory, you must move (or copy) them to a new directory. In addition, you must manually recompile user-defined applications before you can use them with a 3.x web server, as described in “Backward Compatibility with Earlier Releases” on page 8. Be aware that an application can’t be used with Enterprise Server 2.0 after recompiling. If you want to use an application with both servers, you should copy the application instead of moving it.
- **New versions of the sample applications.** Many of the sample applications that shipped with LiveWire 1.x have been changed. The upgrade process installs new versions of the `world`, `hangman`, `cipher`, `dbadmin`, and `viewer` sample applications. In addition, the sample application `lwccall` has been updated and renamed `jsaccall`. The sample application `video` has been updated and renamed `oldvideo`; a new version of this application, using new LiveWire Database Service features, is named `videoapp`. Finally, there are several new sample applications, `bank`, `bugbase`, `flexi`, and `sendmail`, that demonstrate other new server-side JavaScript features. For information on the sample applications, see Chapter 11, “Quick Start with the Sample Applications.”  
  
If you modified the old sample applications in the old `samples` directory and you want to transfer your changes to the new server, you must move (or copy) them and recompile them, as you do your own applications.
- **Changes you should make.** For information on changes you may have to make in your code when upgrading, see the next section, “Backward Compatibility with Earlier Releases.”

# Backward Compatibility with Earlier Releases

You should be aware of the following changes in the behavior of server-side JavaScript applications.

- **Recompile applications.** Web files compiled with the earlier version will not run with 3.x Netscape web servers. You must recompile all of your existing JavaScript applications. In earlier releases, the JavaScript application compiler was called `lwcomp`. It is now called `jsac` and has additional options. For information on using the compiler, see “Compiling an Application” on page 59. Once you recompile your applications, they will not work under LiveWire 1.x.
- **Change JavaScript code.** Some changes in core and client-side JavaScript may require you to change your JavaScript source code. For information on these changes, see “Changes to Core JavaScript 1.2” on page 5.
- **Change property references.** In earlier releases, you could refer to an object’s properties by their property name or by their ordinal index. In this release, however, if you initially define a property by its name, you must always refer to it by its name, and if you initially define a property by an index, you must always refer to it by its index. So, the following code is now illegal:

```
obj = new Object();
obj.prop = 42;
write(obj[0] == 42); //Illegal! Cannot refer to obj.prop as obj[0]
```

- **Variable with no value returns undefined.** In earlier releases, if you referred to a defined variable for which you had provided no value, it returned `NULL`. In this release, it returns `undefined`. Consider this code:

```
<server>
var myVar;
write("The value of myVar is: " + myVar);
</server>
```

In earlier releases, that code would produce this output:

```
The value of myVar is: NULL
```

Now it produces this output:

```
The value of myVar is: undefined
```



- **client and request objects not created automatically.** In earlier releases, the runtime engine created `client` and `request` objects for an application's initial page. The properties of this `client` object were not available on other pages. In this release, the runtime engine creates neither a `client` object nor a `request` object for an application's initial page. You can use the following statements to create these objects:

```
client = new Object();
request = new Object();
```

Note, however, that if you create these objects, their properties are still not available on any other pages of the application.

- **Site Manager removed.** LiveWire 1.x included Site Manager for managing your web sites. This functionality was removed from the 3.x web servers. You can instead use the Web Publisher to publish your documents, and you must use the command-line compiler, `jsac`, to compile your applications.
- **Changes to the lock method.** The behavior of the `lock` method for the `project` and `server` objects has changed. In earlier releases, if you called `project.lock` or `server.lock`, no other thread (for either the same or a different application) could make any changes to the `project` or `server` object until you called `project.unlock` or `server.unlock`. That is, the locking did not require any cooperation.

In this release, cooperation among different applications or pages in the same application is required. If one thread calls `project.lock` or `server.lock`, and if another thread then calls the same method, that method will wait until the first thread calls `project.unlock` or `server.unlock` or until a specified timeout period has elapsed. If, however, the second thread does not call `project.lock` or `server.lock`, it can make changes to those objects. For more information on locking, see “Sharing Objects Safely with Locking” on page 279.

- **Changes to LiveWire Database Service.** There are several changes in how you can use the LiveWire Database Service to connect your JavaScript application to a relational database:
  - **Very Important:** In this release, if your database server and web server are not on the same machine, you must install a database client to use the LiveWire Database Service. In earlier releases, this was optional. In addition, the required version of the client library may be newer than that required in LiveWire 1.x. For more information, see “Supported Database Clients and ODBC Drivers” on page 372.
  - In earlier releases, you could leave a database connection or cursor open and allow the system to close it for you. In this release, the system no longer does this. When finished with them, your code must release all connections opened with `DbPool` objects and close all cursors opened either with `database` or `Connection` objects. For information on managing connections, see Chapter 15, “Connecting to a Database.” For information on cursors, see “Manipulating Query Results with Cursors” on page 338.
  - In earlier releases, you could choose to modify a row with an updatable cursor without first starting an explicit transaction by calling `beginTransaction`. In this release, you must always use explicit transaction control (with the `beginTransaction`, `commitTransaction`, and `rollbackTransaction` methods) when using an updatable cursor and making changes to the database. For information on cursors, see “Manipulating Query Results with Cursors” on page 338. For information on transactions, see “Managing Transactions” on page 348.
  - In earlier releases, if a JavaScript error occurred while a transaction was in progress, that transaction was committed. In this release, if the transaction is through the `database` object, the transaction is rolled back. If the transaction is through a `DbPool` object, the value of the `commitFlag` parameter when the connection was established determines whether the transaction is committed or rolled back. For information on establishing connections, see Chapter 15, “Connecting to a Database.”

# Contents

<b>New Features in this Release</b> .....	3
Changes to Server-Side JavaScript .....	3
Changes to Core JavaScript 1.2 .....	5
Upgrading from an Earlier Release .....	6
Backward Compatibility with Earlier Releases .....	8
<b>About this Book</b> .....	23
New Features in this Release .....	23
What You Should Already Know .....	23
JavaScript Versions .....	24
Where to Find JavaScript Information .....	25
Document Conventions .....	26
<b>Chapter 1 JavaScript Overview</b> .....	29
What Is JavaScript? .....	30
Core, Client-Side, and Server-Side JavaScript .....	31
Core JavaScript .....	32
Client-Side JavaScript .....	32
Server-Side JavaScript .....	34
JavaScript and Java .....	36
Debugging JavaScript .....	37
Visual JavaScript .....	38
JavaScript and the ECMA Specification .....	38
Relationship Between JavaScript and ECMA Versions .....	39
JavaScript Documentation vs. the ECMA Specification .....	40
JavaScript and ECMA Terminology .....	40

## Part 1 Developing Server Applications

<b>Chapter 2 Getting Started</b> .....	43
Architecture of JavaScript Applications .....	43
System Requirements .....	47
Configuration Information .....	49
Enabling Server-Side JavaScript .....	49
Protecting the Application Manager .....	49
Setting Up for LiveConnect .....	50
Locating the Compiler .....	51
<b>Chapter 3 How to Develop Server Applications</b> .....	53
Basic Steps in Building an Application .....	54
JavaScript Application Manager Overview .....	56
Creating Application Source Files .....	58
Compiling an Application .....	59
Installing a New Application .....	61
Application URLs .....	64
Controlling Access to an Application .....	65
Modifying Installation Fields .....	66
Removing an Application .....	66
Starting, Stopping, and Restarting an Application .....	66
Running an Application .....	67
Debugging an Application .....	68
Using the Application Manager .....	68
Using Debug URLs .....	69
Using the debug Function .....	70
Deploying an Application .....	70
Application Manager Details .....	71
Configuring Default Settings .....	71
Under the Hood .....	73

## Part 2 Core Language Features

<b>Chapter 4 Values, Variables, and Literals</b>	77
Values	77
Data Type Conversion	78
Variables	79
Declaring Variables	79
Evaluating Variables	79
Variable Scope	80
Literals	81
Array Literals	81
Boolean Literals	82
Floating-Point Literals	83
Integers	83
Object Literals	83
String Literals	84
<b>Chapter 5 Expressions and Operators</b>	87
Expressions	87
Operators	88
Assignment Operators	89
Comparison Operators	90
Arithmetic Operators	91
Bitwise Operators	92
Logical Operators	95
String Operators	96
Special Operators	97
Operator Precedence	102
<b>Chapter 6 Regular Expressions</b>	103
Creating a Regular Expression	104
Writing a Regular Expression Pattern	104
Using Simple Patterns	105
Using Special Characters	105
Using Parentheses	110

Working with Regular Expressions .....	110
Using Parenthesized Substring Matches .....	113
Executing a Global Search and Ignoring Case .....	115
Examples .....	116
Changing the Order in an Input String .....	116
Using Special Characters to Verify Input .....	117
<b>Chapter 7 Statements</b> .....	119
Conditional Statements .....	120
if...else Statement .....	120
switch Statement .....	121
Loop Statements .....	122
for Statement .....	122
do...while Statement .....	124
while Statement .....	124
label Statement .....	125
break Statement .....	126
continue Statement .....	127
Object Manipulation Statements .....	128
for...in Statement .....	128
with Statement .....	129
Comments .....	130
<b>Chapter 8 Functions</b> .....	131
Defining Functions .....	131
Calling Functions .....	132
Using the arguments Array .....	133
Predefined Functions .....	134
eval Function .....	135
isFinite Function .....	135
isNaN Function .....	136
parseInt and parseFloat Functions .....	136
Number and String Functions .....	137
escape and unescape Functions .....	138

<b>Chapter 9 Working with Objects</b>	139
Objects and Properties	140
Creating New Objects	141
Using Object Initializers	141
Using a Constructor Function	142
Indexing Object Properties	144
Defining Properties for an Object Type	144
Defining Methods	145
Using this for Object References	146
Deleting Objects	147
Predefined Core Objects	147
Array Object	147
Boolean Object	151
Date Object	151
Function Object	155
Math Object	156
Number Object	158
RegExp Object	158
String Object	159
<b>Chapter 10 Details of the Object Model</b>	161
Class-Based vs. Prototype-Based Languages	162
Defining a Class	162
Subclasses and Inheritance	163
Adding and Removing Properties	163
Summary of Differences	163
The Employee Example	165
Creating the Hierarchy	166
Object Properties	169
Inheriting Properties	169
Adding Properties	171
More Flexible Constructors	173

Property Inheritance Revisited .....	178
Local versus Inherited Values .....	178
Determining Instance Relationships .....	180
Global Information in Constructors .....	181
No Multiple Inheritance .....	183

## Part 3 Server-Side JavaScript Features

<b>Chapter 11 Quick Start with the Sample Applications</b> .....	187
Hello World .....	190
What Hello World Does .....	191
Looking at the Source Script .....	192
Modifying Hello World .....	195
Hangman .....	196
Looking at the Source Files .....	198
Debugging Hangman .....	201
<b>Chapter 12 Basics of Server-Side JavaScript</b> .....	203
What to Do Where .....	204
Overview of Runtime Processing .....	206
Server-Side Language Overview .....	208
Core Language .....	209
Usage .....	211
Environment .....	211
Classes and Objects .....	213
Embedding JavaScript in HTML .....	216
The SERVER tag .....	217
Backquotes .....	218
When to Use Each Technique .....	220
Runtime Processing on the Server .....	220
Constructing the HTML Page .....	225
Generating HTML .....	226
Flushing the Output Buffer .....	226
Changing to a New Client Request .....	227



Accessing CGI Variables .....	228
Communicating Between Server and Client .....	232
Sending Values from Client to Server .....	232
Sending Values from Server to Client .....	237
Using Cookies .....	239
Garbage Collection .....	242
<b>Chapter 13 Session Management Service .....</b>	<b>245</b>
Overview of the Predefined Objects .....	246
The request Object .....	249
Properties .....	250
Working with Image Maps .....	252
The client Object .....	252
Properties .....	253
Uniquely Referring to the client Object .....	255
Creating a Custom client Object .....	256
The project Object .....	260
Properties .....	260
Sharing the project Object .....	261
The server Object .....	261
Properties .....	262
Sharing the server Object .....	263
Techniques for Maintaining the client Object .....	263
Comparing Client-Maintenance Techniques .....	264
Client-Side Techniques .....	268
Server-Side Techniques .....	271
The Lifetime of the client Object .....	275
Manually Appending client Properties to URLs .....	277
Sharing Objects Safely with Locking .....	279
Using Instances of Lock .....	280
Special Locks for project and server Objects .....	283
Avoiding Deadlock .....	284

<b>Chapter 14 Other JavaScript Functionality</b>	287
Mail Service	287
File System Service	290
Security Considerations	290
Creating a File Object	291
Opening and Closing a File	291
Locking Files	292
Working with Files	293
Example	297
Working with External Libraries	297
Guidelines for Writing Native Functions	299
Identifying Library Files	299
Registering Native Functions	300
Using Native Functions in JavaScript	300
Request and Response Manipulation	302
Request Header	303
Request Body	304
Response Header	305

## Part 4 LiveWire Database Service

<b>Chapter 15 Connecting to a Database</b>	309
Interactions with Databases	310
Approaches to Connecting	311
Database Connection Pools	314
Single-Threaded and Multithreaded Databases	316
Managing Connection Pools	318
Sharing a Fixed Set of Connection Pools	320
Sharing an Array of Connection Pools	321
Individual Database Connections	323
Maintaining a Connection Across Requests	325
Waiting for a Connection	327
Retrieving an Idle Connection	328

<b>Chapter 16 Working with a Database</b>	335
Automatically Displaying Query Results	336
Executing Arbitrary SQL Statements	337
Manipulating Query Results with Cursors	338
Creating a Cursor	339
Displaying Record Values	341
Displaying Expressions and Aggregate Functions	343
Navigating with Cursors	344
Working with Columns	345
Changing Database Information	346
Managing Transactions	348
Using the Transaction-Control Methods	349
Working with Binary Data	351
Calling Stored Procedures	354
Exchanging Information	354
Steps for Using Stored Procedures	356
Registering the Stored Procedure	357
Defining a Prototype for a Stored Procedure	358
Executing the Stored Procedure	358
Working with Result Sets	360
Working with Return Values	366
Working with Output Parameters	367
Informix and Sybase Exceptions	368
<b>Chapter 17 Configuring Your Database</b>	369
Checking Your Database Configuration	370
Supported Database Clients and ODBC Drivers	372
DB2	376
DB2 Remote	376
DB2 Local	377
Informix	378
Informix Remote	378
Informix Local	379

ODBC .....	379
ODBC Data Source Names (NT only) .....	380
OpenLink ODBC Driver (Solaris only) .....	380
Visigenic ODBC Driver (Unix only) .....	381
Oracle .....	381
Oracle Remote .....	382
Oracle Local .....	383
Sybase .....	383
Sybase Remote .....	384
Sybase Local .....	384
Sybase (Unix only) .....	385
<b>Chapter 18 Data Type Conversion .....</b>	<b>387</b>
Working with Dates and Databases .....	388
Data-Type Conversion by Database .....	388
<b>Chapter 19 Error Handling for LiveWire .....</b>	<b>391</b>
Return Values .....	392
Number .....	392
Object .....	393
Boolean .....	394
String .....	395
Void .....	395
Error Methods .....	396
Status Codes .....	397
<b>Chapter 20 Videoapp and Oldvideo Sample Applications .....</b>	<b>399</b>
Configuring Your Environment .....	400
Connecting to the Database and Recompiling .....	400
Creating the Database .....	401
Running Videoapp .....	406
Looking at the Source Files .....	407
Application Architecture .....	408
Modifying videoapp .....	411

## Part 5 Working with LiveConnect

<b>Chapter 21 LiveConnect Overview</b>	415
What Is LiveConnect?	416
Working with Wrappers	417
JavaScript to Java Communication	418
The Packages Object	419
Working with Java Arrays	420
Package and Class References	420
Arguments of Type char	421
Example of JavaScript Calling Java	421
Java to JavaScript Communication	422
Using the LiveConnect Classes	423
Accessing Server-Side JavaScript	426
Data Type Conversions	429
JavaScript to Java Conversions	429
Java to JavaScript Conversions	435
<b>Chapter 22 Accessing CORBA Services</b>	437
About CORBA Services	437
Flexi Sample Application	439
CORBA Client and Server Processes	440
Starting FlexiServer	441
Starting Flexi	442
Using Flexi	442
Looking at the Source Files	443
Deployment Alternatives	447
<b>Glossary</b>	449
<b>Index</b>	455



# About this Book

JavaScript is Netscape's cross-platform, object-based scripting language for client and server applications. This book explains everything you need to know to begin creating server-side JavaScript applications.

This preface contains the following sections:

- New Features in this Release
- What You Should Already Know
- JavaScript Versions
- Where to Find JavaScript Information
- Document Conventions

## New Features in this Release

For a summary of JavaScript 1.2 features, see “New Features in this Release” on page 3. Information on these features has been incorporated in this manual.

## What You Should Already Know

This book assumes you have the following basic background:

- A general understanding of the Internet and the World Wide Web (WWW).
- A general understanding of client-side JavaScript. This book does not duplicate client-side language information.
- Good working knowledge of HyperText Markup Language (HTML). Experience with forms and the Common Gateway Interface (CGI) is also useful.
- Some programming experience in Pascal, C, Perl, Visual Basic, or a similar language.

- Familiarity with relational databases and a working knowledge of Structured Query Language (SQL), if you're going to use the LiveWire Database Service.

## JavaScript Versions

Each version of Navigator supports a different version of JavaScript. To help you write scripts that are compatible with multiple versions of Navigator, this manual lists the JavaScript version in which each feature was implemented.

The following table lists the JavaScript version supported by different Navigator versions. Versions of Navigator prior to 2.0 do not support JavaScript.

Table 1 JavaScript and Navigator versions

JavaScript version	Navigator version
JavaScript 1.0	Navigator 2.0
JavaScript 1.1	Navigator 3.0
JavaScript 1.2	Navigator 4.0–4.05

Each version of the Netscape Enterprise Server also supports a different version of JavaScript. To help you write scripts that are compatible with multiple versions of the Enterprise Server, this manual uses an abbreviation to indicate the server version in which each feature was implemented.

Table 2 JavaScript and Netscape Enterprise Server versions

Abbreviation	Enterpriser Server version
NES 2.0	Netscape Enterprise Server 2.0
NES 3.0	Netscape Enterprise Server 3.0



# Where to Find JavaScript Information

The server-side JavaScript documentation includes the following books:

- The *Server-Side JavaScript Guide* (this book) provides information about the JavaScript language and its objects. This book contains information for both core and server-side JavaScript. Some core language features work differently on the client than on the server; these differences are discussed in this book.
- The *Server-Side JavaScript Reference* provides reference material for the JavaScript language, including both core and server-side JavaScript.

If you are new to JavaScript, start with Chapter 1, “JavaScript Overview,” then continue with the rest of the book. Once you have a firm grasp of the fundamentals, you can use the *Server-Side JavaScript Reference* to get more details on individual objects and statements.

Use the material in this book to familiarize yourself with core and server-side JavaScript. Use the *Client-Side JavaScript Guide* and *Client-Side JavaScript Reference* for information on scripting HTML pages.

The *Netscape Enterprise Server Programmer’s Bookshelf* summarizes the different programming interfaces available with the 3.x versions of Netscape web servers. Use this guide as a roadmap or starting point for exploring the Enterprise Server documentation for developers.

The Netscape web site contains information that can be useful when you’re working with JavaScript. The following URLs are of particular interest:

- [http://home.netscape.com/one\\_stop/intranet\\_apps/index.html](http://home.netscape.com/one_stop/intranet_apps/index.html)

The Netscape AppFoundry Online home page is a source for starter applications, technical information, tools, and expert forums for quickly building and dynamically deploying open intranet applications. This site also includes troubleshooting information in the resources section and extra help on setting up your JavaScript environment.

- <http://help.netscape.com/products/tools/livewire/>

Netscape’s technical support page for information on the LiveWire Database Service contains many useful pointers to information on using LiveWire in JavaScript applications.

- <http://developer.netscape.com/tech/javascript/ssjs/index.html>

Netscape's support page for server-side JavaScript contains news and resources related to server-side JavaScript. For quick access to this URL, click the Documentation link on the Netscape Enterprise Server Application Manager.

- <http://developer.netscape.com/viewsource/index.html>

View Source Magazine, Netscape's online technical magazine for developers, is updated every other week and frequently contains articles of interest to JavaScript developers.

## Document Conventions

JavaScript applications run on many operating systems; the information in this book applies to all versions. File and directory paths are given in Windows format (with backslashes separating directory names). For Unix versions, the directory paths are the same, except that you use slashes instead of backslashes to separate directories.

This book uses uniform resource locators (URLs) of the following form:

`http://server.domain/path/file.html`

In these URLs, *server* represents the name of the server on which you run your application, such as `research1` or `www`; *domain* represents your Internet domain name, such as `netscape.com` or `uiuc.edu`; *path* represents the directory structure on the server; and *file.html* represents an individual file name. In general, items in italics in URLs are placeholders and items in normal monospace font are literals. If your server has Secure Sockets Layer (SSL) enabled, you would use `https` instead of `http` in the URL.

This book uses the following font conventions:

- The `monospace font` is used for sample code and code listings, API and language elements (such as method names and property names), file names, path names, directory names, HTML tags, and any text that must be typed on the screen. (*Monospace italic font* is used for placeholders embedded in code.)
- *Italic type* is used for book titles, emphasis, variables and placeholders, and words used in the literal sense.
- **Boldface type** is used for glossary terms.



# JavaScript Overview

This chapter introduces JavaScript and discusses some of its fundamental concepts.

This chapter contains the following sections:

- What Is JavaScript?
- Core, Client-Side, and Server-Side JavaScript
- JavaScript and Java
- Debugging JavaScript
- Visual JavaScript
- JavaScript and the ECMA Specification

# What Is JavaScript?

JavaScript is Netscape's cross-platform, object-oriented scripting language. Core JavaScript contains a core set of objects, such as `Array`, `Date`, and `Math`, and a core set of language elements such as operators, control structures, and statements. Core JavaScript can be extended for a variety of purposes by supplementing it with additional objects; for example:

- *Client-side JavaScript* extends the core language by supplying objects to control a browser (Navigator or another web browser) and its Document Object Model (DOM). For example, client-side extensions allow an application to place elements on an HTML form and respond to user events such as mouse clicks, form input, and page navigation.
- *Server-side JavaScript* extends the core language by supplying objects relevant to running JavaScript on a server. For example, server-side extensions allow an application to communicate with a relational database, provide continuity of information from one invocation to another of the application, or perform file manipulations on a server.

JavaScript lets you create applications that run over the Internet. Client applications run in a browser, such as Netscape Navigator, and server applications run on a server, such as Netscape Enterprise Server. Using JavaScript, you can create dynamic HTML pages that process user input and maintain persistent data using special objects, files, and relational databases.

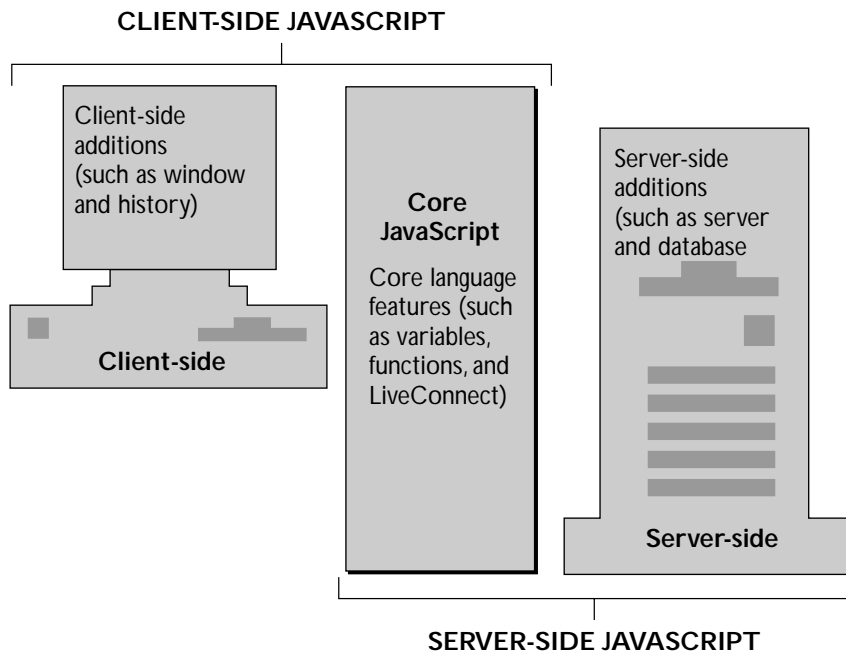
Through JavaScript's LiveConnect functionality, you can let Java and JavaScript code communicate with each other. From JavaScript, you can instantiate Java objects and access their public methods and fields. From Java, you can access JavaScript objects, properties, and methods.

Netscape invented JavaScript, and JavaScript was first used in Netscape browsers.

# Core, Client-Side, and Server-Side JavaScript

The components of JavaScript are illustrated in the following figure.

Figure 1.1 The JavaScript language



The following sections introduce the workings of JavaScript on the client and on the server.

## Core JavaScript

Client-side and server-side JavaScript have the following elements in common:

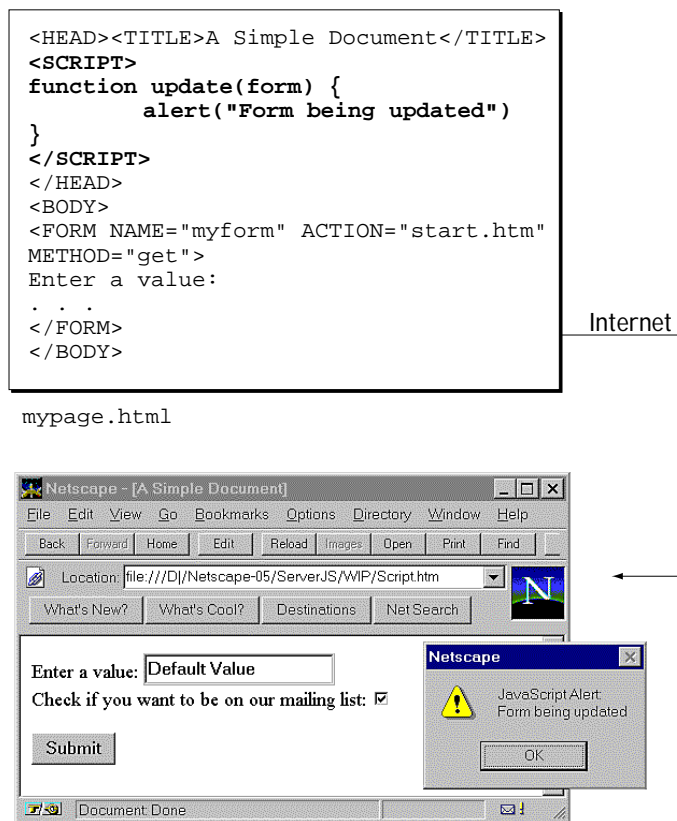
- Keywords
- Statement syntax and grammar
- Rules for expressions, variables, and literals
- Underlying object model (although client-side and server-side JavaScript have different sets of predefined objects)
- Predefined objects and functions, such as `Array`, `Date`, and `Math`

## Client-Side JavaScript

Web browsers such as Navigator (2.0 and later versions) can interpret client-side JavaScript statements embedded in an HTML page. When the browser (or *client*) requests such a page, the server sends the full content of the document, including HTML and JavaScript statements, over the network to the client. The browser reads the page from top to bottom, displaying the results of the HTML and executing JavaScript statements as they are encountered. This process, illustrated in the following figure, produces the results that the user sees.



Figure 1.2 Client-side JavaScript



Client-side JavaScript statements embedded in an HTML page can respond to user events such as mouse clicks, form input, and page navigation. For example, you can write a JavaScript function to verify that users enter valid information into a form requesting a telephone number or zip code. Without any network transmission, the embedded JavaScript on the HTML page can check the entered data and display a dialog box if the user enters invalid data.

Different versions of JavaScript work with specific versions of Navigator. For example, JavaScript 1.2 is for Navigator 4.0. Some features available in JavaScript 1.2 are not available in JavaScript 1.1 and hence are not available in Navigator 3.0. For information on JavaScript and Navigator versions, see "JavaScript Versions" on page 24.

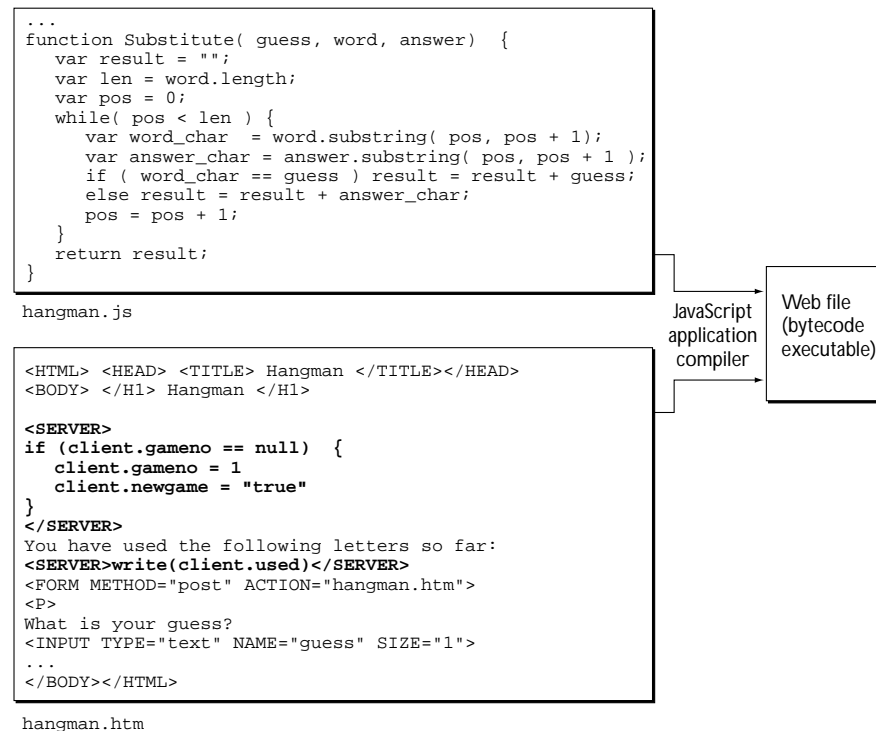
## Server-Side JavaScript

On the server, you also embed JavaScript in HTML pages. The server-side statements can connect to relational databases from different vendors, share information across users of an application, access the file system on the server, or communicate with other applications through LiveConnect and Java. HTML pages with server-side JavaScript can also include client-side JavaScript.

In contrast to pure client-side JavaScript pages, HTML pages that use server-side JavaScript are compiled into bytecode executable files. These application executables are run by a web server that contains the JavaScript runtime engine. For this reason, creating JavaScript applications is a two-stage process.

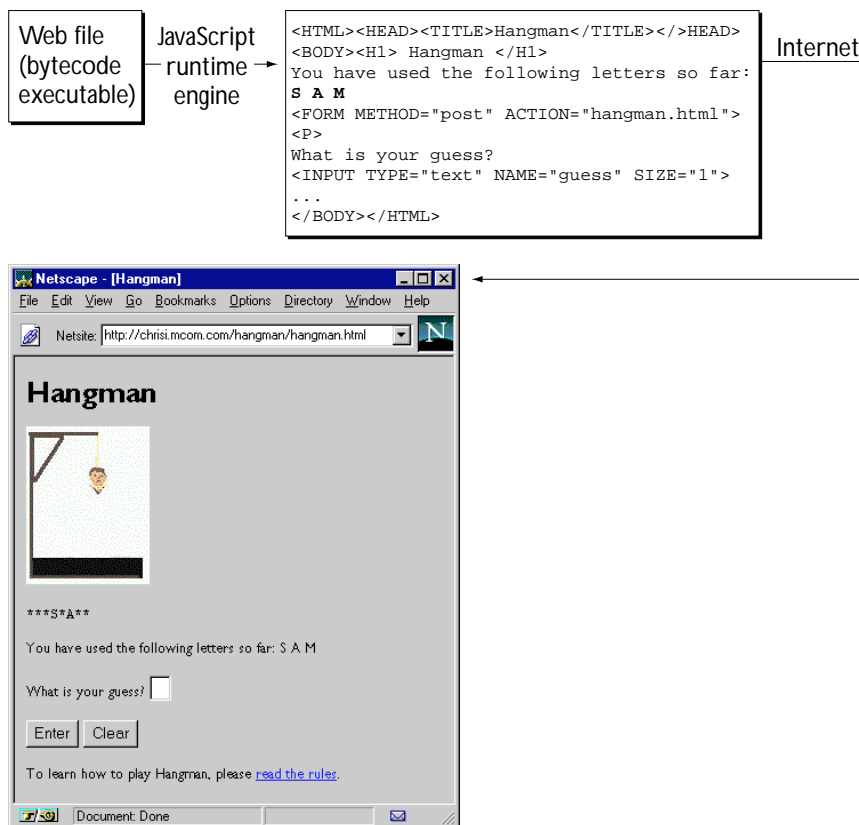
In the first stage, shown in Figure 1.3, you create HTML pages (which can contain both client-side and server-side JavaScript statements) and JavaScript files. You then compile all of those files into a single executable.

Figure 1.3 Server-side JavaScript during development



In the second stage, shown in Figure 1.4, a page in the application is requested by a client browser. The runtime engine uses the application executable to look up the source page and dynamically generate the HTML page to return. It runs any server-side JavaScript statements found on the page. The result of those statements might add new HTML or client-side JavaScript statements to the HTML page. The run-time engine then sends the resulting page over the network to the Navigator client, which runs any client-side JavaScript and displays the results.

Figure 1.4 Server-side JavaScript during runtime



In contrast to standard Common Gateway Interface (CGI) programs, all JavaScript source is integrated directly into HTML pages, facilitating rapid development and easy maintenance. Server-side JavaScript's Session Management Service contains objects you can use to maintain data that persists across client requests, multiple clients, and multiple applications. Server-side JavaScript's LiveWire Database Service provides objects for database access that serve as an interface to Structured Query Language (SQL) database servers.

## JavaScript and Java

JavaScript and Java are similar in some ways but fundamentally different in others. The JavaScript language resembles Java but does not have Java's static typing and strong type checking. JavaScript supports most Java expression syntax and basic control-flow constructs.

In contrast to Java's compile-time system of classes built by declarations, JavaScript supports a runtime system based on a small number of data types representing numeric, Boolean, and string values. JavaScript has a prototype-based object model instead of the more common class-based object model. The prototype-based model provides dynamic inheritance; that is, what is inherited can vary for individual objects. JavaScript also supports functions without any special declarative requirements. Functions can be properties of objects, executing as loosely typed methods.

JavaScript is a very free-form language compared to Java. You do not have to declare all variables, classes, and methods. You do not have to be concerned with whether methods are public, private, or protected, and you do not have to implement interfaces. Variables, parameters, and function return types are not explicitly typed.

Java is a class-based programming language designed for fast execution and type safety. Type safety means, for instance, that you can't cast a Java integer into an object reference or access private memory by corrupting Java bytecodes. Java's class-based model means that programs consist exclusively of classes and their methods. Java's class inheritance and strong typing generally require tightly coupled object hierarchies. These requirements make Java programming more complex than JavaScript authoring.

In contrast, JavaScript descends in spirit from a line of smaller, dynamically typed languages such as HyperTalk and dBASE. These scripting languages offer programming tools to a much wider audience because of their easier syntax, specialized built-in functionality, and minimal requirements for object creation.

Table 1.1 JavaScript compared to Java

JavaScript	Java
Interpreted (not compiled) by client.	Compiled bytecodes downloaded from server, executed on client.
Object-oriented. No distinction between types of objects. Inheritance is through the prototype mechanism, and properties and methods can be added to any object dynamically.	Class-based. Objects are divided into classes and instances with all inheritance through the class hierarchy. Classes and instances cannot have properties or methods added dynamically.
Code integrated with, and embedded in, HTML.	Applets distinct from HTML (accessed from HTML pages).
Variable data types not declared (dynamic typing).	Variable data types must be declared (static typing).
Cannot automatically write to hard disk.	Cannot automatically write to hard disk.

For more information on the differences between JavaScript and Java, see Chapter 10, “Details of the Object Model.”

## Debugging JavaScript

JavaScript allows you to write complex computer programs. As with all languages, you may make mistakes while writing your scripts. The Netscape JavaScript Debugger allows you to debug your scripts.

For information on using the Debugger, see *Getting Started with Netscape JavaScript Debugger*.

# Visual JavaScript

Netscape Visual JavaScript is a component-based visual development tool for the Netscape Open Network Environment (ONE) platform. It is primarily intended for use by application developers who want to build cross-platform, standards-based, web applications from ready-to-use components with minimal programming effort. The applications are based on HTML, JavaScript, and Java.

For information on Visual JavaScript, see the *Visual JavaScript Developer's Guide*.

## JavaScript and the ECMA Specification

Netscape invented JavaScript, and JavaScript was first used in Netscape browsers. However, Netscape is working with ECMA (European Computer Manufacturers Association) to deliver a standardized, international programming language based on core JavaScript. ECMA is an international standards association for information and communication systems. This standardized version of JavaScript, called ECMAScript, behaves the same way in all applications that support the standard. Companies can use the open standard language to develop their implementation of JavaScript. The first version of the ECMA standard is documented in the ECMA-262 specification.

The ECMA-262 standard is also approved by the ISO (International Organization for Standards) as ISO-16262. You can find a PDF version of ECMA-262 at Netscape DevEdge Online. You can also find the specification on the ECMA web site. The ECMA specification does not describe the Document Object Model (DOM), which is being standardized by the World Wide Web Consortium (W3C). The DOM defines the way in which HTML document objects are exposed to your script.

## Relationship Between JavaScript and ECMA Versions

Netscape works closely with ECMA to produce the ECMA specification. The following table describes the relationship between JavaScript and ECMA versions.

Table 1.2 JavaScript and ECMA versions

JavaScript version	Relationship to ECMA version
JavaScript 1.1	ECMA-262 is based on JavaScript 1.1.
JavaScript 1.2	<p>ECMA-262 was not complete when JavaScript 1.2 was released. JavaScript 1.2 is not fully compatible with ECMA-262 for the following reasons:</p> <ul style="list-style-type: none"><li>• Netscape developed additional features in JavaScript 1.2 that were not considered for ECMA-262.</li><li>• ECMA-262 adds two new features: internationalization using Unicode, and uniform behavior across all platforms. Several features of JavaScript 1.2, such as the <code>Date</code> object, were platform-dependent and used platform-specific behavior.</li></ul>

The *Server-Side JavaScript Reference* indicates which features of the language are ECMA-compliant.

JavaScript will always include features that are not part of the ECMA specification; JavaScript is compatible with ECMA, while providing additional features.

## JavaScript Documentation vs. the ECMA Specification

The ECMA specification is a set of requirements for implementing ECMAScript; it is useful if you want to determine whether a JavaScript feature is supported under ECMA. If you plan to write JavaScript code that uses only features supported by ECMA, then you may need to review the ECMA specification.

The ECMA document is not intended to help script programmers; use the JavaScript documentation for information on writing scripts.

## JavaScript and ECMA Terminology

The ECMA specification uses terminology and syntax that may be unfamiliar to a JavaScript programmer. Although the description of the language may differ in ECMA, the language itself remains the same. JavaScript supports all functionality outlined in the ECMA specification.

The JavaScript documentation describes aspects of the language that are appropriate for a JavaScript programmer. For example:

- The global object is not discussed in the JavaScript documentation because you do not use it directly. The methods and properties of the global object, which you do use, are discussed in the JavaScript documentation but are called top-level functions and properties.
- The no parameter (zero-argument) constructor with the `Number` and `String` objects is not discussed in the JavaScript documentation, because what is generated is of little use. A `Number` constructor without an argument returns `+0`, and a `String` constructor without an argument returns `""` (an empty string).



# 1

## *Developing Server Applications*

- Getting Started
- How to Develop Server Applications



# Getting Started

This chapter provides an overview of what a typical server-side JavaScript application looks like, and it shows you how to set up your system for server-side development.

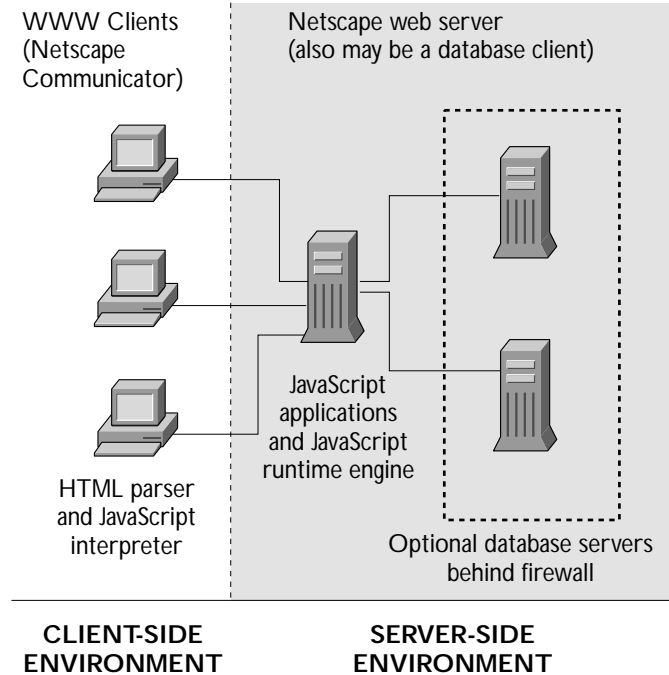
This chapter contains the following sections:

- Architecture of JavaScript Applications
- System Requirements
- Configuration Information

## Architecture of JavaScript Applications

As discussed in earlier sections, JavaScript applications have portions that run on the client and on the server. In addition, many JavaScript applications use the LiveWire Database Service to connect the application to a relational database. For this reason, you can think of JavaScript applications as having a three-tier client-server architecture, as illustrated in Figure 2.1.

Figure 2.1 Architecture of the client-server JavaScript application environment



The three tiers are:

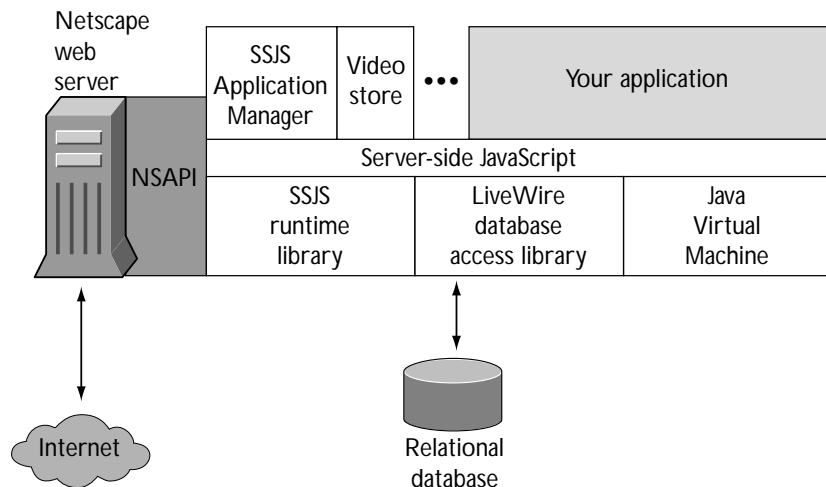
- *WWW clients (such as Netscape Navigator clients):* This tier provides a cross-platform end-user interface to the application. This tier can also contain some application logic, such as data-validation rules implemented in client-side JavaScript. Clients can be inside or outside the corporate firewall.
- *Netscape WWW server/database client:* This tier consists of a Netscape server, with server-side JavaScript enabled. It contains the application logic, manages security, and controls access to the application by multiple users, using server-side JavaScript. This tier allows clients both inside and outside the firewall to access the application. The WWW server also acts as a client to any installed database servers.

- Database servers:** This tier consists of SQL database servers, typically running on high-performance workstations. It contains all the database data, metadata, and referential integrity rules required by the application. This tier typically is inside the corporate firewall and can provide a layer of security in addition to that provided by the WWW server. Netscape Enterprise Server supports the use of ODBC, DB2, Informix, Oracle, and Sybase database servers. Netscape FastTrack Server supports only ODBC. For further information about the LiveWire Database Service, see Part 4, “LiveWire Database Service.”

The JavaScript client-side environment runs as part of WWW clients, and the JavaScript server-side environment runs as part of a Netscape web server with access to one or more database servers. Figure 2.2 shows more detail of how the server-side JavaScript environment, and applications built for this environment, fit into the Netscape web server.

The top part of Figure 2.2 shows how server-side JavaScript fits into a Netscape web server. Inside the web server, the server-side JavaScript runtime environment is built from three main components which are listed below. The JavaScript Application Manager then runs on top of server-side JavaScript, as do the sample applications provided by Netscape (such as the `videoapp` application) and any applications you create.

Figure 2.2 Server-side JavaScript in the Netscape server environment



These are the three primary components of the JavaScript runtime environment:

- *The JavaScript runtime library:* This component provides basic JavaScript functionality. An example is the Session Management Service, which provides predefined objects to help manage your application and share information between the client and the server and between multiple applications. The Session Management Service is described in Chapter 13, “Session Management Service.”
- *The LiveWire database access library:* This component extends the base server-side JavaScript functionality with classes and objects that provide seamless access to external database servers. It is described in Part 4, “LiveWire Database Service.”
- *The Java virtual machine:* Unlike the other components, the Java virtual machine is not only for use with JavaScript; any Java application running on the server uses this virtual machine. The Java virtual machine has been augmented to allow JavaScript applications to access Java classes, using JavaScript’s LiveConnect functionality. LiveConnect is described in Chapter 21, “LiveConnect Overview.”

In general, a JavaScript application can contain statements interpreted by the client (with the JavaScript interpreter provided with Netscape Navigator or some other web browser) and by the server (with the JavaScript runtime engine just discussed).

When you run a JavaScript application, a variety of things occur, some on the server and some on the client. Although the end user does not need to know the details, it is important for you, the application developer, to understand what happens “under the hood.”

In creating your application, you write HTML pages that can contain both server-side and client-side JavaScript statements. In the source code HTML, client-side JavaScript is delimited by the `SCRIPT` tag and server-side JavaScript by the `SERVER` tag.

You can also write files that contain only JavaScript statements and no HTML tags. A JavaScript file can contain either client-side JavaScript or server-side JavaScript; a single file cannot contain both client-side and server-side objects or functions.

If the HTML and JavaScript files contain server-side JavaScript, you then compile them into a single JavaScript application executable file. The executable is called a web file and has the extension `.web`. The JavaScript application compiler turns the source code HTML into platform-independent bytecodes, parsing and compiling server-side JavaScript statements.

Finally, you deploy your application on your web server and use the JavaScript Application Manager to install and start the application, so that users can access your application.

At runtime, when a client requests a page from a server-side JavaScript application, the runtime engine locates the representation of that file in the application's web file. It runs all the server code found and creates an HTML page to send to the client. That page can contain both regular HTML tags and client-side JavaScript statements. All server code is run on the server, before the page goes to the client and before any of the HTML or client-side JavaScript is executed. Consequently, your server-side code cannot use any client-side objects, nor can your client-side code use any server-side objects.

For more details, see Chapter 12, "Basics of Server-Side JavaScript."

## System Requirements

To develop and run JavaScript applications that take advantage of both client-side and server-side JavaScript, you need appropriate development and deployment environments. In general, it is recommended that you develop applications on a system other than your deployment (production) server because development consumes resources (for example, communications ports, bandwidth, processor cycles, and memory). Development might also disrupt end-user applications that have already been deployed.

A JavaScript development environment consists of

- *Development tools* for authoring and compiling JavaScript applications. These tools typically are resident on the development machine.
- *A development machine with a web server* for running JavaScript applications that are under development.
- *A deployment machine with a web server* for deploying finished applications. End users access completed applications on this server.

The development tools needed include:

- A JavaScript-enabled browser, such as Netscape Navigator, included in Netscape Communicator.
- A JavaScript application compiler, such as the one bundled with Netscape web servers.
- An editor, such as Emacs or Notepad.

The development and deployment machines require the following software:

- A web server.
- A JavaScript runtime engine, such as the one bundled with Netscape web servers.
- A way to configure your server to run JavaScript applications, as provided in the JavaScript Application Manager bundled with Netscape web servers.

In addition, if your application uses JavaScript's LiveWire Database Service, you need the following:

- Relational database server software on your database server machine. For more information, refer to your database server documentation. In some cases, you may want to install the web server and the database server on the same machine. For specific requirements for server-side JavaScript, see Chapter 17, "Configuring Your Database."
- Your database's client and networking software on your web server machine. If you use one machine as both your database server and web server, typically the necessary database client software is installed when the database server is installed. Otherwise, you must ensure that the database client software is installed on the same machine as the web server, so that it can access the database as a client. For more information on database client software requirements, refer to the database vendor's documentation.



# Configuration Information

This section provides configuration information for using server-side JavaScript. For additional information on setting up your database to work with the LiveWire Database Service, see Chapter 17, “Configuring Your Database.”

## Enabling Server-Side JavaScript

To run JavaScript applications on your server, you must enable the JavaScript runtime engine from your Server Manager by clicking Programs and then choosing server-side JavaScript. At the prompt “Activate the JavaScript application environment?”, choose Yes and click OK. You are also asked about restricting access to the Application Manager. For more information, see “Protecting the Application Manager” on page 49.

**Note** If you do not enable the JavaScript runtime engine, JavaScript applications cannot run on the server.

Once you activate the JavaScript application environment, you must stop and restart your web server for the associated environment variables to take effect. If you do not, JavaScript applications that use the LiveWire Database Service will not run.

## Protecting the Application Manager

The Application Manager provides control over JavaScript applications. Because of its special capabilities, you should protect it from unauthorized access. If you don’t restrict access to the Application Manager, anyone can add, remove, modify, start, and stop applications on your server. This can have undesirable consequences.

You (the JavaScript application developer) need to have permission to use the Application Manager on your development server, because you use it to work with the application as you develop it. Your web server administrator, however, may choose to not give you this access to the deployment server.

When you enable the JavaScript runtime engine in the Server Manager, a prompt asks you whether to restrict access to the Application Manager. Choose Yes to do so, then click OK. (Yes is the default.) After this point, anyone

attempting to access the Application Manager must enter the Server Manager user name and password to use the Application Manager and the `dbadmin` sample application. For more information, see the administrator's guide for your web server.

If your server is not using the Secure Sockets Layer (SSL), the user name and password for the Application Manager are transmitted unencrypted over the network. An intruder who intercepts this data can get access to the Application Manager. If you use the same password for your administration server, the intruder will also have control of your server. Therefore, it is recommended that you do not use the Application Manager outside your firewall unless you use SSL. For instructions on how to turn on SSL for your server, see the administrator's guide for your web server.

## Setting Up for LiveConnect

Netscape web servers include Java classes you can use with JavaScript. The installation procedures for these servers put those classes in the `$NSHOME\js\samples` directory, where `$NSHOME` is the directory in which you installed the server. The installation procedure also modifies the web server's `CLASSPATH` environment variable to automatically include this directory.

You must either install your Java classes in this same directory or modify the `CLASSPATH` environment variable of the server to include the location of your Java classes. In addition, the `CLASSPATH` environment variable of the process in which you compile the Java classes associated with your JavaScript application must also include the location of your Java classes.

Remember, if you use the Admin Server to start your web server, you'll have to set `CLASSPATH` before you start the Admin Server. Alternatively, you can directly modify the `obj.conf` file for your web server. For information on this file, see your web server's administrator's guide.

On NT, if you modify `CLASSPATH` and you start the server using the Services panel of the control panel, you must reboot your machine after you set `CLASSPATH` in the System panel of the control panel.

## Locating the Compiler

Installation of a Netscape server does not change your `PATH` environment variable to include the directory in which the JavaScript application compiler is installed. If you want to be able to easily refer to the location of the compiler, you must modify this environment variable.

On Unix systems, you have various choices on how to change your `PATH` environment variable. You can add `$NSHOME/bin/https`, where `$NSHOME` is the directory in which you installed the server. See your system administrator for information on how to do so.

To change your NT system path, start the Control Panel application, locate the System dialog box, and set the `PATH` variable in the Environment settings to include the `%NSHOME%\bin\https`, where `NSHOME` is the directory in which you installed the server.

If you move the JavaScript application compiler to a different directory, you must add that directory to your `PATH` environment variable.



# How to Develop Server Applications

This chapter describes the process of developing your application, such as how to use the JavaScript application compiler and how to use the Application Manager of Netscape servers to install or debug your application. For information on using only client-side JavaScript, see the *Client-Side JavaScript Guide*.

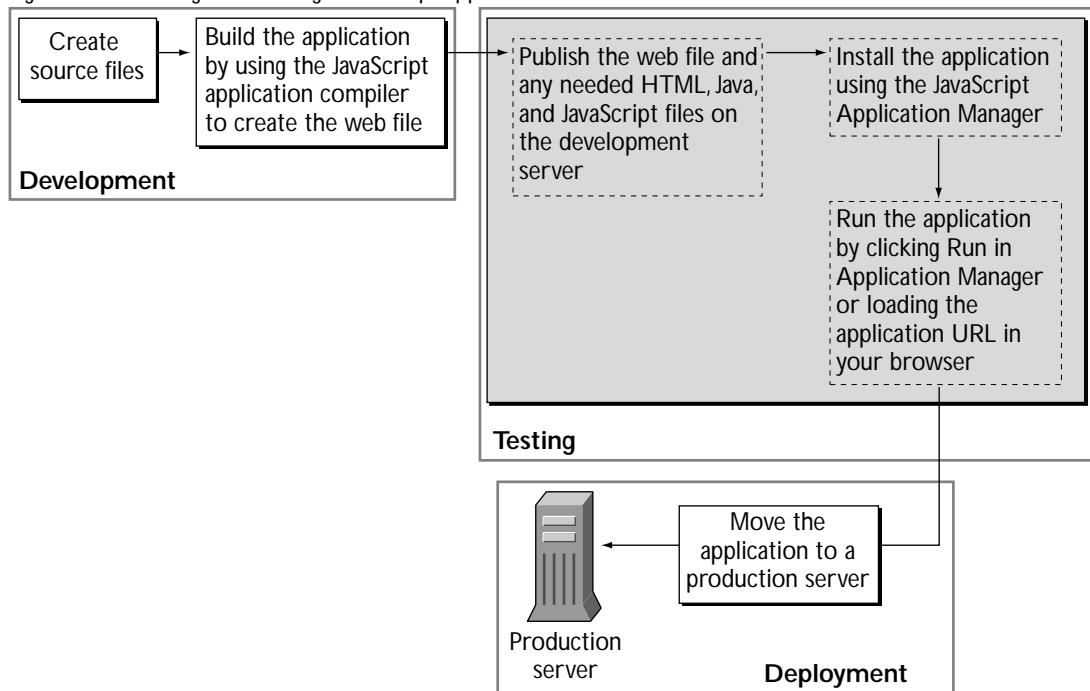
This chapter contains the following sections:

- Basic Steps in Building an Application
- JavaScript Application Manager Overview
- Creating Application Source Files
- Compiling an Application
- Installing a New Application
- Controlling Access to an Application
- Modifying Installation Fields
- Removing an Application
- Starting, Stopping, and Restarting an Application
- Running an Application
- Debugging an Application
- Deploying an Application
- Application Manager Details

# Basic Steps in Building an Application

Normally, HTML is static: after you write an HTML page, its content is fixed. The fixed content is transmitted from the server to the client when the client accesses the page's URL. With JavaScript, you can create HTML pages that change based on changing data and user actions. Figure 3.1 shows the basic procedure for creating and running a JavaScript application.

Figure 3.1 Creating and running a JavaScript application



You take these basic steps to build a JavaScript application:

1. Create the source files. The source files can be HTML files with embedded JavaScript, files containing only JavaScript, or Java source files. (See “Creating Application Source Files” on page 58.)
2. Build the application by using the JavaScript application compiler to create the bytecode executable (.web file). (See “Compiling an Application” on page 59.) Compile Java source files into class files.

3. Publish the web file, any needed uncompiled HTML, image, and client-side JavaScript files, and compiled Java class files in appropriate directories on the server. You can use the Netscape Web Publisher to publish your files, as described in the *Web Publisher User's Guide*.
4. Install the application for the first time (see “Installing a New Application” on page 61) using the JavaScript Application Manager. You also use the Application Manager to restart an application after rebuilding it (see “Starting, Stopping, and Restarting an Application” on page 66). Installing or restarting the application enables the JavaScript runtime engine to run it.  
  
After installing an application, you may want to protect it. See “Deploying an Application” on page 70. You do not need to restart an application after you initially install it.
5. Run the application by clicking Run in the Application Manager or loading the application URL in your browser. (See “Running an Application” on page 67 and “Application URLs” on page 64.) For example, to run Hello World, load `http://server.domain/world/`. You can also debug the application by clicking Debug in the Application Manager. (See “Debugging an Application” on page 68.)
6. After you have completed developing and testing your application, you need to deploy it to make it available to users. Deploying generally involves installing it on a production server and changing access restrictions. (See “Deploying an Application” on page 70.)

Before you can develop JavaScript applications, you need to enable the runtime engine on the server and should protect the JavaScript Application Manager from unauthorized access. For more information, see “Configuration Information” on page 49.

# JavaScript Application Manager Overview

Before learning how to create JavaScript applications, you should become familiar with the JavaScript Application Manager. You can use the Application Manager to accomplish these tasks:

- Add a new JavaScript application.
- Modify any of the attributes of an installed application.
- Stop, start, and restart an installed application.
- Run and debug an active application.
- Remove an installed application.

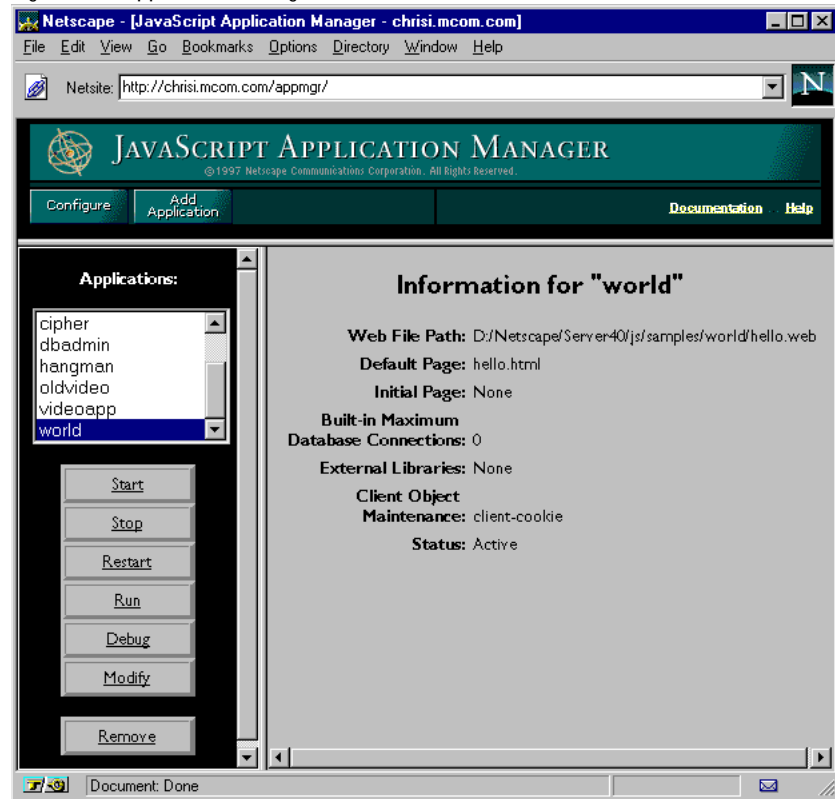
The Application Manager is itself a JavaScript application that demonstrates the power and flexibility of JavaScript. You start the JavaScript Application Manager from the following URL in Navigator:

`http://server.domain/appmgr`

In response, the Application Manager displays the page shown in Figure 3.2.



Figure 3.2 Application Manager



The Application Manager displays, in a scrolling list in the left frame, all JavaScript applications currently installed on the server. Select an application by clicking its name in the scrolling list.

For the selected application, the right frame displays the

- Application name at the top of the frame
- Path of the application web file on the server
- Default and initial pages for the application
- Maximum number of database connections allowed for the predefined database object
- External libraries (if any)

- `Client` object maintenance technique
- Status of the application: *active* or *stopped* (Users can run only active applications. Stopped applications are not accessible.)

For a description of these fields, see “Installing a New Application” on page 61.

Click the Add Application button in the top frame to add a new application. Click the task buttons in the left frame to perform the indicated action on the selected application. For example, to modify the installation fields of the selected application, click Modify.

Click Configure to configure the default settings for the Application Manager. Click Documentation to reach Netscape’s technical support page for server-side JavaScript, including links to all sorts of documentation about it. Click Help for more instructions on using the Application Manager.

## Creating Application Source Files

The first step in building a JavaScript application is to create and edit the source files. The web file for a JavaScript application can contain two kinds of source files:

- Files with standard HTML or JavaScript embedded in HTML. These files have the file extension (suffix) `.html` or `.htm`.
- Files with JavaScript functions only. These files have the file extension `.js`.

When you use JavaScript in an HTML file, you must follow the rules outlined in “Embedding JavaScript in HTML” on page 216.

Do not use any special tags in `.js` files; the JavaScript application compiler on the server and the JavaScript interpreter on the client assume everything in the file is JavaScript. While an HTML file is used on both the client and the server, a single JavaScript file must be either for use on the server or on the client; it cannot be used on both. Consequently, a JavaScript file can contain either client-side JavaScript or server-side JavaScript, but a single file cannot contain both client-side and server-side objects or functions.

The JavaScript application compiler compiles and links the HTML and JavaScript files that contain server-side JavaScript into a single platform-independent bytecode web file, with the file extension `.web`, as described in “Compiling an Application” on page 59.

You install a web file to be run with the JavaScript runtime engine, as described in “Installing a New Application” on page 61.

## Compiling an Application

You compile a JavaScript application using the JavaScript application compiler, `jsac`. The compiler creates a web file from HTML and JavaScript source files.

For ease of accessing the compiler, you may want to add the directory in which it is installed to your `PATH` environment variable. For information on how to do so, see “Locating the Compiler” on page 51.

You only need to compile those pages containing server-side JavaScript or both client-side and server-side JavaScript. You do not need to compile pages that contain only client-side JavaScript. You can do so, but runtime performance is better if you leave them uncompiled.

The compiler is available from any command prompt. Use the following command-line syntax to compile and link JavaScript applications on the server:

```
jsac [-h] [-c] [-v] [-d] [-l]
      [-o outfile.web]
      [-i inputFile]
      [-p pathName]
      [-f includeFile]
      [-r errorFile]
      [-a 1.2]
      script1.html [...scriptN.html]
      [funct1.js ... functN.js]
```

Items enclosed in square brackets are optional. The syntax is shown on multiple lines for clarity. The `scriptN.html` and `functN.js` files are the input files to the compiler. There must be at least one HTML file. By default, the HTML and JavaScript files are relative to the current directory. Files you specify must be either JavaScript files or HTML files; you cannot specify other files, such as GIF files.

On all platforms, you may use either the dash (-) or the forward slash (/) to indicate a command-line option. That is, the following lines are equivalent:

```
jsac -h
jsac /h
```

Note that because the forward slash indicates a command-line option, an input file cannot start with a forward slash to indicate that it is an absolute pathname. That is, the following call is illegal:

```
jsac -o myapp.web /usr/vpg/myapp.html
```

This restriction does not apply to any of the pathnames you supply as arguments to command-line options; only to the input files. On NT, you can instead use backslash (\) to indicate an absolute pathname in an input file, as in the following call:

```
jsac -o myapp.web \usr\vpg\myapp.html
```

On Unix, you must use the -i command-line option to specify an absolute pathname, as described below.

The following command-line options are available:

- -h: Displays compiler syntax help. If you supply this option, don't use any other options.
- -c: Checks syntax only; does not generate a web file. If you supply this option, you do not need to supply the -o option.
- -v: (Verbose) Displays information about the running of the compiler.
- -d: Displays generated JavaScript contents.
- -l: Specifies the character set to use when compiling (such as iso-8859-1, x-sjis, or euc-kr)
- -o *outfile*: Creates a bytecode-format web file, named *outfile.web*. If you do not supply this option, the compiler does not generate a web file. (Omit this option only if you're using the -c option to check syntax or -h to get help.)
- -i *inputFile*: Allows you to specify an input file using its full pathname instead of a relative pathname. You can provide only one filename to this option. If you need to specify multiple filenames using full pathnames, use the -f option.

- `-p pathName`: Specifies a directory to be the root of all relative pathnames used during compilation. (Use before the `-f` option.) You can provide only one pathname to this option.
- `-f includeFile`: Specifies a file that is actually a list of input files, allowing you to circumvent the character limit for a command line. You can provide only one filename to this option. The list of input files in *includeFile* is white-space delimited. If a filename contains a space, you must enclose the filename in double quotes.
- `-r errorFile`: Redirects standard output (including error messages) to the specified file. You can provide only one filename to this option.
- `-a 1.2`: Changes how the compiler handles comparison operators on the server. For more information, see “Comparison Operators” on page 209.

For example, the following command compiles and links two JavaScript-enhanced HTML pages, `main.html` and `hello.html`, and a server-side JavaScript file, `support.js`, creating a binary executable named `myapp.web`. In addition, during compilation, the compiler prints progress information to the command line.

```
jsac -v -o myapp.web main.html hello.html support.js
```

As a second example, the following command compiles the files listed in the file `looksee.txt` into a binary executable called `looksee.web`:

```
jsac -f looksee.txt -o looksee.web
```

Here, `looksee.txt` might contain the following:

```
looksee1.html
looksee2.html
\myapps\jsplace\common.js
looksee3.html
```

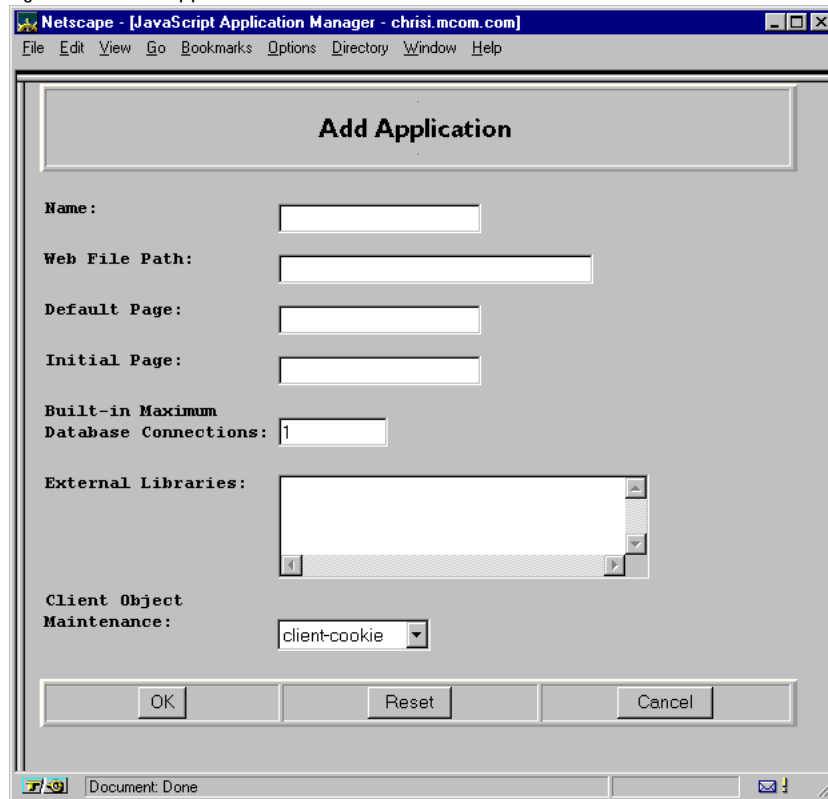
## Installing a New Application

You cannot run an application and clients cannot access it until you install it. Installing an application identifies it to the server. After you have installed the application, you can rebuild and run it any number of times. You need to reinstall it only if you subsequently remove it. You can install up to 120 JavaScript applications on one server.

Before you install, you must move all application-related files to the correct directory, by publishing the files. Otherwise, you'll get an error when you install the application. For security reasons, you may not want to publish your JavaScript source files on your deployment server. See "Application URLs" on page 64 for restrictions on where you can place your files.

To install a new application with the Application Manager, click Add Application. In response, the Application Manager displays, in its right frame, the form shown in Figure 3.3.

Figure 3.3 Add Application form



The screenshot shows a Netscape browser window titled "Netscape - [JavaScript Application Manager - chrisi.mcom.com]". The address bar shows "File Edit View Go Bookmarks Options Directory Window Help". The main content area displays a form titled "Add Application". The form contains the following fields and controls:

- Name:** A text input field.
- Web File Path:** A text input field.
- Default Page:** A text input field.
- Initial Page:** A text input field.
- Built-in Maximum Database Connections:** A text input field containing the value "1".
- External Libraries:** A text area with scrollbars.
- Client Object Maintenance:** A dropdown menu currently showing "client-cookie".

At the bottom of the form are three buttons: "OK", "Reset", and "Cancel". The status bar at the bottom of the browser window shows "Document: Done".

Fill in the fields in the Add Application form, as follows:

- *Name*: the name of the application. This name defines the application URL. For example, the name of the Hello World application is “world,” and its application URL is `http://server.domain/world`. This is a required field, and the name you type must be different from all other application names on the server. See “Application URLs” on page 64.
- *Web File Path*: the full pathname of the application web file. This is a required field. For example, if you installed the Netscape server in `c:\nshome`, the web file path for the Hello World application is `c:\nshome\js\samples\world\hello.web`.
- *Default Page*: the page that the JavaScript runtime engine serves if the user does not indicate a specific page in the application. This page is analogous to `index.html` for a standard URL.
- *Initial Page*: the page that the JavaScript runtime engine executes when you start the application in the Application Manager. This page is executed exactly once per running of the application. It is generally used to initialize values, create locks, and establish database connections. Any JavaScript source on this page cannot use either of the predefined `request` or `client` objects. This is an optional field.
- *Built-in Maximum Database Connections*: the default value for the maximum number of database connections that this application can have at one time using the predefined `database` object. JavaScript code can override what you specify in this setting when it calls the `database.connect` method.
- *External Libraries*: the pathnames of external libraries to be used with the application. If you specify multiple libraries, delimit the names with either commas or semicolons. This is an optional field. Libraries installed for one application can be used by all applications on the server. See “Working with External Libraries” on page 297.
- *Client Object Maintenance*: the technique used to save the properties of the `client` object. This can be client cookie, client URL, server IP, server cookie, or server URL. See “Techniques for Maintaining the client Object” on page 263.

After you have provided all the required information, click Enter to install the application, Reset to clear all the fields, or Cancel to cancel the operation.

You must stop and restart your server after you add or change the external libraries for an application. You can restart a server from your Server Manager; see the administrator's guide for your web server for more information.

# Application URLs

When you install an application, you must supply a name for it. This name determines the base application URL, the URL that clients use to access the default page of a JavaScript application. The base application URL is of the form

```
http://server.domain/appName
```

Here, *server* is the name of the HTTP server, *domain* is the Internet domain (including any subdomains), and *appName* is the application name you enter when you install it. Individual pages within the application are accessed by application URLs of the form

```
http://server.domain/appName/page.html
```

Here, *page* is the name of a page in the application. For example, if your server is named *coyote* and your domain name is *royalairways.com*, the base application URL for the *hangman* sample application is

```
http://coyote.royalairways.com/hangman
```

When a client requests this URL, the server generates HTML for the default page in the application and sends it to the client. The application URL for the winning page in this application is

```
http://coyote.royalairways.com/hangman/youwon.html
```

**Important** Before you install an application, be sure the application name you choose does not usurp an existing URL on your server. The JavaScript runtime engine routes all client requests for URLs that match the application URL to the directory specified for the web file. This circumvents the server's normal document root.

For instance, suppose a client requests a URL that starts with this prefix from the previous example:

```
http://coyote.royalairways.com/hangman
```



In this case, the runtime engine on the server looks for a document in the `samples\hangman` directory and not in your server's normal document root. The server also serves pages in the directory that are not compiled into the application.

You can place your source (uncompiled) server-side JavaScript files in the same directory as the web file; however, you should do so only for debugging purposes. When you deploy your application to the public, for security reasons, you should not publish uncompiled server-side JavaScript files.

## Controlling Access to an Application

When you install an application, you may want to restrict the users who can access it, particularly if the application provides access to sensitive information or capabilities.

If you work on a development server inside a firewall, then you may not need to worry about restricting access while developing the application. It is convenient to have unrestricted access during development, and you may be able to assume that the application is safe from attack inside the firewall. If you use sample data during the development phase, then the risk is even less. However, if you leave your application open, you should be aware that anyone who knows or guesses the application URL can use the application.

When you finish development and are ready to deploy your application, you should reconsider how you want to protect it. You can restrict access by applying a server configuration style to the application. For information on configuration styles, see the administrator's guide for your web server.

**Note** Controlling access to applications with configuration styles is available only with Netscape 2.0 servers and later versions.

## Modifying Installation Fields

To modify an application's installation fields, select the application name in the left frame of the Application Manager and click Modify.

You can change any of the fields defined when you installed the application, except the application name. To change the name of an application, you must remove the application and then reinstall it.

If you modify the fields of a stopped application, the Application Manager automatically starts it. When you modify fields of an active application, the Application Manager automatically stops and restarts it.

## Removing an Application

To remove the selected application, click Remove in the Application Manager. The Application Manager removes the application so that it cannot be run on the server. Clients are no longer able to access the application. If you delete an application and subsequently want to run it, you must install it again.

Although clients can no longer use the application, removing it with the Application Manager does not delete the application's files from the server. If you want to delete them as well, you must do so manually.

## Starting, Stopping, and Restarting an Application

After you first install an application, you must start it to run it. Select the application in the Application Manager and click Start. If the application successfully starts, its status, indicated in the right frame, changes from Stopped to Active.

You can also start an application by loading the following URL:

```
http://server.domain/appmgr/control.html?name=appName&cmd=start
```

Here, *appName* is the application name. You cannot use this URL unless you have access privileges for the Application Manager.

If you want to stop an application and thereby make it inaccessible to users, select the application name in the Application Manager and click Stop. The application's status changes to Stopped and clients can no longer run the application. You must stop an application if you want to move the web file or update an application from a development server to a deployment server.

You can also stop an application by loading the following URL:

```
http://server.domain/appmgr/control.html?name=appName&cmd=stop
```

Here, *appName* is the application name. You cannot use this URL unless you have access privileges for the Application Manager.

You must restart an application each time you rebuild it. To restart an active application, select it in the Application Manager and click Restart. Restarting essentially reinstalls the application; the software looks for the specified web file. If there is not a valid web file, then the Application Manager generates an error.

You can also restart an application by loading the following URL:

```
http://server.domain/appmgr/control.html?name=appName&cmd=restart
```

Here, *appName* is the application name. You cannot use this URL unless you have access privileges for the Application Manager.

## Running an Application

Once you have compiled and installed an application, you can run it in one of two ways:

- Select the application name in the Application Manager, and then click Run. In response, the Application Manager opens a new Navigator window to access the application.
- Load the base application URL in Navigator by typing it in the Location field.

The server then generates HTML for the specified application page and sends it to the client.

# Debugging an Application

To debug an application, do one of the following:

- Select the application name in the Application Manager, and then click Debug, as described in “Using the Application Manager” on page 68.
- Load the application’s debug URL, as described in “Using Debug URLs” on page 69.

You can use the `debug` function to display debugging information, as described in “Using the debug Function” on page 70.

Once you’ve started debugging a JavaScript application in this way, you may not be able to stop or restart it. In this situation, the Application Manager displays the warning “Trace is active.” If this occurs, do the following:

1. Close any windows running the debugger.
2. Close any windows running the affected application.
3. In the Application Manager, select the affected application and click Run.

You can now stop or restart the application.

## Using the Application Manager

To debug an application, select it in the left frame of the Application Manager and then click Debug. In response, the Application Manager opens a new Navigator window in which the application runs. The trace utility also appears, either in a separate frame of the window containing the application or in another window altogether. (You can determine the appearance of the debug window when you configure the default settings for the Application Manager, as described in “Configuring Default Settings” on page 71.)

The trace utility displays this debugging information:

- Values of object properties and arguments of debug functions called by the application
- Property values of the `request` and `client` objects, before and after generating HTML for the page

- Property values of the `project` and `server` objects
- Indication of when the application assigns new values to properties
- Indication of when the runtime engine sends content to the client

The following figure shows what you might see if you debug the Hangman application.

Figure 3.4 Debugging Hangman



## Using Debug URLs

Instead of using the Application Manager, you may find it more convenient to use an application's debug URL. To display an application's trace utility in a separate window, enter the following URL:

```
http://server.domain/appmgr/trace.html?name=appName
```

Here, *appName* is the name of the application. To display the trace utility in the same window as the application (but in a separate frame), enter this URL:

```
http://server.domain/appmgr/debug.html?name=appName
```

You cannot use these two URLs unless you have access privileges to run the Application Manager. You may want to bookmark the debug URL for convenience during development.

## Using the debug Function

You can use the `debug` function in your JavaScript application to help trace problems with the application. The `debug` function displays values to the application trace utility. For example, the following statement displays the value of the `guess` property of the `request` object in the trace window along with some identifying text:

```
debug ("Current Guess is ", request.guess);
```

## Deploying an Application

After you have finished developing and testing your application, you are ready to deploy it so that it is available to its intended users. This involves two steps:

- Moving the application from the development server to the deployment (production) server that is accessible to end users
- Applying or changing access restrictions to the application, as appropriate

You should move the application web file to the deployment server, along with any images and uncompiled HTML and JavaScript files that are needed. For more information on how to deploy your application files, see the *Web Publisher User's Guide*.

**Note** In general, for security reasons, you should not deploy source files.

Depending on the application, you might want to restrict access to certain groups or individuals. In some cases, you might want anyone to be able to run the application; in these cases you don't need to apply any restrictions at all. If

the application displays sensitive information or provides access to the server file system, you should restrict access to authorized users who have the proper user name and password.

You restrict access to an application by applying a server configuration style from your Server Manager. For information on using Server Manager and configuration styles, see the administrator's guide for your web server.

## Application Manager Details

This section shows how to change default settings for the Application Manager. In addition, it talks about the format of the underlying file in which the Application Manager stores information.

### Configuring Default Settings

To configure default settings for the Application Manager, click Configure in the Application Manager's top frame. In response, the Application Manager displays the form shown in Figure 3.5.

You can specify these default values:

- *Web File Path*: A default directory path for your development area.
- *Default Page*: A default name for the default page of a new application.
- *Initial Page*: A default name for the initial page of a new application.
- *Built-in Maximum Database Connections*: A default value for the maximum number of database connections you can make for the predefined database object.
- *External Libraries*: A default directory path for native executables libraries.
- *Client Object Maintenance*: A default maintenance technique for the `client` object properties.

When you install a new application, the default installation fields are used for the initial settings.

In addition, you can specify these preferences:

- *Confirm on:* Whether you are prompted to confirm your action when you remove, start, stop, or restart an application.
- *Debug Output:* Whether, when debugging an application, the application trace appears in the same window as the application but in another frame, or in a window separate from the application.

Figure 3.5 Default Settings form

The screenshot shows a Netscape browser window titled "Netscape - [JavaScript Application Manager - chrisi.mcom.com]". The address bar shows "File Edit View Go Bookmarks Options Directory Window Help". The main content area is titled "Default Values When Adding Applications" and contains the following fields:

- Web File Path:** A text input field.
- Default Page:** A text input field.
- Initial Page:** A text input field.
- Built-in Maximum Database Connections:** A text input field with the value "1".
- External Libraries:** A text area with a scroll bar.
- Client Object Maintenance:** A dropdown menu with "client-cookie" selected.

Below these fields is a section titled "Preferences" with the following options:

- Confirm On:** A group of checkboxes: ☒ Remove, ☐ Start, ☐ Stop, ☐ Restart.
- Debug Output:** A group of radio buttons: ☒ Same Window, ☐ Other Window.

At the bottom of the form are three buttons: "OK", "Reset", and "Cancel". The status bar at the bottom of the browser window shows "Document: Done".



## Under the Hood

The Application Manager is a convenient interface for modifying the configuration file `$NSHOME\https-serverID\config\jsa.conf`, where `$NSHOME` is the directory in which you installed the server and *serverID* is the server's ID. In case of catastrophic errors, you may need to edit this file yourself. In general, this is not recommended, but the information is provided here for troubleshooting purposes.

Each line in `jsa.conf` corresponds to an application. The first item on each line is the application name. The remaining items are in the format `name=value`, where `name` is the name of the installation field, and `value` is its value. The possible values for `name` are:

- `uri`: the application name portion of the base application URL
- `object`: path to the application web file
- `home`: application default page
- `start`: application initial page
- `maxdbconnect`: default maximum number of database connections allowed for the predefined database object
- `library`: paths to external libraries, separated by commas or semicolons
- `client-mode`: technique for maintaining the `client` object

The `jsa.conf` file is limited to 1024 lines, and each line is limited to 1024 characters. If the fields entered in the Application Manager cause a line to exceed this limit, the line is truncated. This usually results in loss of the last item, external library files. If this occurs, reduce the number of external libraries entered for the application, and add the libraries to other applications. Because installed libraries are accessible to all applications, the application can still use them.

A line that starts with `#` indicates a comment. That entire line is ignored. You can also include empty lines in the file.

Do not include multiple lines specifying the same application name. Doing so causes errors in the Application Manager.



# 2

## *Core Language Features*

- **Values, Variables, and Literals**
- **Expressions and Operators**
- **Regular Expressions**
- **Statements**
- **Functions**
- **Working with Objects**
- **Details of the Object Model**



# Values, Variables, and Literals

This chapter discusses values that JavaScript recognizes and describes the fundamental building blocks of JavaScript expressions: variables and literals.

This chapter contains the following sections:

- Values
- Variables
- Literals

## Values

JavaScript recognizes the following types of values:

- Numbers, such as 42 or 3.14159.
- Logical (Boolean) values, either `true` or `false`.
- Strings, such as “Howdy!”.
- `null`, a special keyword denoting a null value; `null` is also a primitive value. Because JavaScript is case sensitive, `null` is not the same as `Null`, `NULL`, or any other variant.
- `undefined`, a top-level property whose value is undefined; `undefined` is also a primitive value.

This relatively small set of types of values, or *data types*, enables you to perform useful functions with your applications. There is no explicit distinction between integer and real-valued numbers. Nor is there an explicit date data type in JavaScript. However, you can use the `Date` object and its methods to handle dates.

Objects and functions are the other fundamental elements in the language. You can think of objects as named containers for values, and functions as procedures that your application can perform.

## Data Type Conversion

JavaScript is a dynamically typed language. That means you do not have to specify the data type of a variable when you declare it, and data types are converted automatically as needed during script execution. So, for example, you could define a variable as follows:

```
var answer = 42
```

And later, you could assign the same variable a string value, for example,

```
answer = "Thanks for all the fish..."
```

Because JavaScript is dynamically typed, this assignment does not cause an error message.

In expressions involving numeric and string values with the `+` operator, JavaScript converts numeric values to strings. For example, consider the following statements:

```
x = "The answer is " + 42 // returns "The answer is 42"  
y = 42 + " is the answer" // returns "42 is the answer"
```

In statements involving other operators, JavaScript does not convert numeric values to strings. For example:

```
"37" - 7 // returns 30  
"37" + 7 // returns 377
```

# Variables

You use variables as symbolic names for values in your application. You give variables names by which you refer to them and which must conform to certain rules.

A JavaScript identifier, or *name*, must start with a letter or underscore (“\_”); subsequent characters can also be digits (0-9). Because JavaScript is case sensitive, letters include the characters “A” through “Z” (uppercase) and the characters “a” through “z” (lowercase).

Some examples of legal names are `Number_hits`, `temp99`, and `_name`.

## Declaring Variables

You can declare a variable in two ways:

- By simply assigning it a value. For example, `x = 42`
- With the keyword `var`. For example, `var x = 42`

## Evaluating Variables

A variable or array element that has not been assigned a value has the value `undefined`. The result of evaluating an unassigned variable depends on how it was declared:

- If the unassigned variable was declared without `var`, the evaluation results in a runtime error.
- If the unassigned variable was declared with `var`, the evaluation results in the undefined value, or NaN in numeric contexts.

The following code demonstrates evaluating unassigned variables.

```
function f1() {
    return y - 2;
}
f1() //Causes runtime error

function f2() {
    return var y - 2;
}
f2() //returns NaN
```

You can use `undefined` to determine whether a variable has a value. In the following code, the variable `input` is not assigned a value, and the `if` statement evaluates to `true`.

```
var input;
if(input === undefined){
    doThis();
} else {
    doThat();
}
```

The `undefined` value behaves as `false` when used as a Boolean value. For example, the following code executes the function `myFunction` because the array element is not defined:

```
myArray=new Array()
if (!myArray[0])
    myFunction()
```

When you evaluate a null variable, the null value behaves as 0 in numeric contexts and as `false` in Boolean contexts. For example:

```
var n = null
n * 32 //returns 0
```

## Variable Scope

When you set a variable identifier by assignment outside of a function, it is called a *global* variable, because it is available everywhere in the current document. When you declare a variable within a function, it is called a *local* variable, because it is available only within the function.

Using `var` to declare a global variable is optional. However, you must use `var` to declare a variable inside a function.



You can access global variables declared in one window or frame from another window or frame by specifying the window or frame name. For example, if a variable called `phoneNumber` is declared in a `FRAMESET` document, you can refer to this variable from a child frame as `parent.phoneNumber`.

## Literals

You use literals to represent values in JavaScript. These are fixed values, not variables, that you *literally* provide in your script. This section describes the following types of literals:

- Array Literals
- Boolean Literals
- Floating-Point Literals
- Integers
- Object Literals
- String Literals

## Array Literals

An array literal is a list of zero or more expressions, each of which represents an array element, enclosed in square brackets (`[]`). When you create an array using an array literal, it is initialized with the specified values as its elements, and its length is set to the number of arguments specified.

The following example creates the `coffees` array with three elements and a length of three:

```
coffees = ["French Roast", "Columbian", "Kona"]
```

**Note** An array literal is a type of object initializer. See “Using Object Initializers” on page 141.

If an array is created using a literal in a top-level script, JavaScript interprets the array each time it evaluates the expression containing the array literal. In addition, a literal used in a function is created each time the function is called.

Array literals are also `Array` objects. See “Array Object” on page 147 for details on `Array` objects.

## Extra Commas in Array Literals

You do not have to specify all elements in an array literal. If you put two commas in a row, the array is created with spaces for the unspecified elements. The following example creates the `fish` array:

```
fish = ["Lion", , "Angel"]
```

This array has two elements with values and one empty element (`fish[0]` is "Lion", `fish[1]` is undefined, and `fish[2]` is "Angel"):

If you include a trailing comma at the end of the list of elements, the comma is ignored. In the following example, the length of the array is three. There is no `myList[3]`. All other commas in the list indicate a new element.

```
myList = ['home', , 'school', ];
```

In the following example, the length of the array is four, and `myList[0]` is missing.

```
myList = [ , 'home', , 'school'];
```

In the following example, the length of the array is four, and `myList[3]` is missing. Only the last comma is ignored. This trailing comma is optional.

```
myList = ['home', , 'school', , ];
```

## Boolean Literals

The Boolean type has two literal values: `true` and `false`.

Do not confuse the primitive Boolean values `true` and `false` with the `true` and `false` values of the Boolean object. The Boolean object is a wrapper around the primitive Boolean data type. See "Boolean Object" on page 151 for more information.

## Floating-Point Literals

A floating-point literal can have the following parts:

- A decimal integer
- A decimal point (“.”)
- A fraction (another decimal number)
- An exponent

The exponent part is an “e” or “E” followed by an integer, which can be signed (preceded by “+” or “-”). A floating-point literal must have at least one digit and either a decimal point or “e” (or “E”).

Some examples of floating-point literals are 3.1415, -3.1E12, .1e12, and 2E-12

## Integers

Integers can be expressed in decimal (base 10), hexadecimal (base 16), and octal (base 8). A decimal integer literal consists of a sequence of digits without a leading 0 (zero). A leading 0 (zero) on an integer literal indicates it is in octal; a leading 0x (or 0X) indicates hexadecimal. Hexadecimal integers can include digits (0-9) and the letters a-f and A-F. Octal integers can include only the digits 0-7.

Some examples of integer literals are: 42, 0xFFFF, and -345.

## Object Literals

An object literal is a list of zero or more pairs of property names and associated values of an object, enclosed in curly braces ({}). You should not use an object literal at the beginning of a statement. This will lead to an error.

The following is an example of an object literal. The first element of the `car` object defines a property, `myCar`; the second element, the `getCar` property, invokes a function (`Cars("honda")`); the third element, the `special` property, uses an existing variable (`Sales`).

```

var Sales = "Toyota";

function CarTypes(name) {
    if(name == "Honda")
        return name;
    else
        return "Sorry, we don't sell " + name + ".";
}

car = {myCar: "Saturn", getCar: CarTypes("Honda"), special: Sales}

document.write(car.myCar); // Saturn
document.write(car.getCar); // Honda
document.write(car.special); // Toyota

```

Additionally, you can use an index for the object, the `index` property (for example, 7), or nest an object inside another. The following example uses these options. These features, however, may not be supported by other ECMA-compliant browsers.

```

car = {manyCars: {a: "Saab", b: "Jeep"}, 7: "Mazda"}

document.write(car.manyCars.b); // Jeep
document.write(car[7]); // Mazda

```

## String Literals

A string literal is zero or more characters enclosed in double (") or single (') quotation marks. A string must be delimited by quotation marks of the same type; that is, either both single quotation marks or both double quotation marks. The following are examples of string literals:

- "blah"
- 'blah'
- "1234"
- "one line \n another line"

You can call any of the methods of the `String` object on a string literal value—JavaScript automatically converts the string literal to a temporary `String` object, calls the method, then discards the temporary `String` object. You can also use the `String.length` property with a string literal.

You should use string literals unless you specifically need to use a `String` object. See “String Object” on page 159 for details on `String` objects.

## Using Special Characters in Strings

In addition to ordinary characters, you can also include special characters in strings, as shown in the following example.

```
"one line \n another line"
```

The following table lists the special characters that you can use in JavaScript strings.

Table 4.1 JavaScript special characters

Character	Meaning
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Tab
\'	Apostrophe or single quote
\"	Double quote
\\	Backslash character (\)
\xxx	The character with the Latin-1 encoding specified by up to three octal digits <i>XXX</i> between 0 and 377. For example, \251 is the octal sequence for the copyright symbol.
\xxx	The character with the Latin-1 encoding specified by the two hexadecimal digits <i>XX</i> between 00 and FF. For example, \xA9 is the hexadecimal sequence for the copyright symbol.

## Escaping Characters

For characters not listed in Table 4.1, a preceding backslash is ignored, with the exception of a quotation mark and the backslash character itself.

You can insert a quotation mark inside a string by preceding it with a backslash. This is known as *escaping* the quotation mark. For example,

```
var quote = "He read \"The Cremation of Sam McGee\" by R.W. Service."  
document.write(quote)
```

The result of this would be

He read "The Cremation of Sam McGee" by R.W. Service.

To include a literal backslash inside a string, you must escape the backslash character. For example, to assign the file path `c:\temp` to a string, use the following:

```
var home = "c:\\temp"
```

# Expressions and Operators

This chapter describes JavaScript expressions and operators, including assignment, comparison, arithmetic, bitwise, logical, string, and special operators.

This chapter contains the following sections:

- Expressions
- Operators

## Expressions

An *expression* is any valid set of literals, variables, operators, and expressions that evaluates to a single value; the value can be a number, a string, or a logical value.

Conceptually, there are two types of expressions: those that assign a value to a variable, and those that simply have a value. For example, the expression `x = 7` is an expression that assigns `x` the value seven. This expression itself evaluates to seven. Such expressions use *assignment operators*. On the other hand, the expression `3 + 4` simply evaluates to seven; it does not perform an assignment. The operators used in such expressions are referred to simply as *operators*.

JavaScript has the following types of expressions:

- Arithmetic: evaluates to a number, for example 3.14159
- String: evaluates to a character string, for example, “Fred” or “234”
- Logical: evaluates to true or false

# Operators

JavaScript has the following types of operators. This section describes the operators and contains information about operator precedence.

- Assignment Operators
- Comparison Operators
- Arithmetic Operators
- Bitwise Operators
- Logical Operators
- String Operators
- Special Operators

JavaScript has both *binary* and *unary* operators. A binary operator requires two operands, one before the operator and one after the operator:

*operand1 operator operand2*

For example, 3+4 or x\*y.

A unary operator requires a single operand, either before or after the operator:

*operator operand*

or

*operand operator*

For example, x++ or ++x.

In addition, JavaScript has one ternary operator, the conditional operator. A ternary operator requires three operands.



## Assignment Operators

An assignment operator assigns a value to its left operand based on the value of its right operand. The basic assignment operator is equal (`=`), which assigns the value of its right operand to its left operand. That is, `x = y` assigns the value of `y` to `x`.

The other assignment operators are shorthand for standard operations, as shown in the following table.

Table 5.1 Assignment operators

Shorthand operator	Meaning
<code>x += y</code>	<code>x = x + y</code>
<code>x -= y</code>	<code>x = x - y</code>
<code>x *= y</code>	<code>x = x * y</code>
<code>x /= y</code>	<code>x = x / y</code>
<code>x %= y</code>	<code>x = x % y</code>
<code>x &lt;&lt;= y</code>	<code>x = x &lt;&lt; y</code>
<code>x &gt;&gt;= y</code>	<code>x = x &gt;&gt; y</code>
<code>x &gt;&gt;&gt;= y</code>	<code>x = x &gt;&gt;&gt; y</code>
<code>x &amp;= y</code>	<code>x = x &amp; y</code>
<code>x ^= y</code>	<code>x = x ^ y</code>
<code>x  = y</code>	<code>x = x   y</code>

# Comparison Operators

A comparison operator compares its operands and returns a logical value based on whether the comparison is true. The operands can be numerical or string values. Strings are compared based on standard lexicographical ordering. The following table describes the comparison operators.

Table 5.2 Comparison operators

Operator	Description	Examples returning true <sup>a</sup>
Equal (==)	Returns true if the operands are equal.	<code>3 == var1</code>
Not equal (!=)	Returns true if the operands are not equal.	<code>var1 != 4</code>
Greater than (>)	Returns true if the left operand is greater than the right operand.	<code>var2 &gt; var1</code>
Greater than or equal (>=)	Returns true if the left operand is greater than or equal to the right operand.	<code>var2 &gt;= var1</code> <code>var1 &gt;= 3</code>
Less than (<)	Returns true if the left operand is less than the right operand.	<code>var1 &lt; var2</code>
Less than or equal (<=)	Returns true if the left operand is less than or equal to the right operand.	<code>var1 &lt;= var2</code> <code>var2 &lt;= 5</code>

a. These examples assume that `var1` has been assigned the value 3 and `var2` has been assigned the value 4.

## Arithmetic Operators

Arithmetic operators take numerical values (either literals or variables) as their operands and return a single numerical value. The standard arithmetic operators are addition (+), subtraction (-), multiplication (\*), and division (/). These operators work as they do in most other programming languages, except the / operator returns a floating-point division in JavaScript, not a truncated division as it does in languages such as C or Java. For example:

```
1/2 //returns 0.5 in JavaScript
1/2 //returns 0 in Java
```

In addition, JavaScript provides the arithmetic operators listed in the following table.

Table 5.3 Arithmetic Operators

Operator	Description	Example
% (Modulus)	Binary operator. Returns the integer remainder of dividing the two operands.	12 % 5 returns 2.
++ (Increment)	Unary operator. Adds one to its operand. If used as a prefix operator (++x), returns the value of its operand after adding one; if used as a postfix operator (x++), returns the value of its operand before adding one.	If x is 3, then ++x sets x to 4 and returns 4, whereas x++ sets x to 4 and returns 3.
-- (Decrement)	Unary operator. Subtracts one to its operand. The return value is analogous to that for the increment operator.	If x is 3, then --x sets x to 2 and returns 2, whereas x-- sets x to 2 and returns 3.
- (Unary negation)	Unary operator. Returns the negation of its operand.	If x is 3, then -x returns -3.

# Bitwise Operators

Bitwise operators treat their operands as a set of 32 bits (zeros and ones), rather than as decimal, hexadecimal, or octal numbers. For example, the decimal number nine has a binary representation of 1001. Bitwise operators perform their operations on such binary representations, but they return standard JavaScript numerical values.

The following table summarizes JavaScript’s bitwise operators.

Table 5.4 Bitwise operators

Operator	Usage	Description
Bitwise AND	<i>a &amp; b</i>	Returns a one in each bit position for which the corresponding bits of both operands are ones.
Bitwise OR	<i>a   b</i>	Returns a one in each bit position for which the corresponding bits of either or both operands are ones.
Bitwise XOR	<i>a ^ b</i>	Returns a one in each bit position for which the corresponding bits of either but not both operands are ones.
Bitwise NOT	<i>~ a</i>	Inverts the bits of its operand.
Left shift	<i>a &lt;&lt; b</i>	Shifts <i>a</i> in binary representation <i>b</i> bits to left, shifting in zeros from the right.
Sign-propagating right shift	<i>a &gt;&gt; b</i>	Shifts <i>a</i> in binary representation <i>b</i> bits to right, discarding bits shifted off.
Zero-fill right shift	<i>a &gt;&gt;&gt; b</i>	Shifts <i>a</i> in binary representation <i>b</i> bits to the right, discarding bits shifted off, and shifting in zeros from the left.

## Bitwise Logical Operators

Conceptually, the bitwise logical operators work as follows:

- The operands are converted to thirty-two-bit integers and expressed by a series of bits (zeros and ones).
- Each bit in the first operand is paired with the corresponding bit in the second operand: first bit to first bit, second bit to second bit, and so on.
- The operator is applied to each pair of bits, and the result is constructed bitwise.

For example, the binary representation of nine is 1001, and the binary representation of fifteen is 1111. So, when the bitwise operators are applied to these values, the results are as follows:

- $15 \ \& \ 9$  yields 9 ( $1111 \ \& \ 1001 = 1001$ )
- $15 \ | \ 9$  yields 15 ( $1111 \ | \ 1001 = 1111$ )
- $15 \ ^ \ 9$  yields 6 ( $1111 \ ^ \ 1001 = 0110$ )

## Bitwise Shift Operators

The bitwise shift operators take two operands: the first is a quantity to be shifted, and the second specifies the number of bit positions by which the first operand is to be shifted. The direction of the shift operation is controlled by the operator used.

Shift operators convert their operands to thirty-two-bit integers and return a result of the same type as the left operator.

The shift operators are listed in the following table.

Table 5.5 Bitwise shift operators

Operator	Description	Example
<< (Left shift)	This operator shifts the first operand the specified number of bits to the left. Excess bits shifted off to the left are discarded. Zero bits are shifted in from the right.	9<<2 yields 36, because 1001 shifted 2 bits to the left becomes 100100, which is 36.
>> (Sign-propagating right shift)	This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Copies of the leftmost bit are shifted in from the left.	9>>2 yields 2, because 1001 shifted 2 bits to the right becomes 10, which is 2. Likewise, -9>>2 yields -3, because the sign is preserved.
>>> (Zero-fill right shift)	This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Zero bits are shifted in from the left.	19>>>2 yields 4, because 10011 shifted 2 bits to the right becomes 100, which is 4. For non-negative numbers, zero-fill right shift and sign-propagating right shift yield the same result.

# Logical Operators

Logical operators are typically used with Boolean (logical) values; when they are, they return a Boolean value. However, the `&&` and `||` operators actually return the value of one of the specified operands, so if these operators are used with non-Boolean values, they may return a non-Boolean value. The logical operators are described in the following table.

Table 5.6 Logical operators

Operator	Usage	Description
<code>&amp;&amp;</code>	<code>expr1 &amp;&amp; expr2</code>	(Logical AND) Returns <code>expr1</code> if it can be converted to false; otherwise, returns <code>expr2</code> . Thus, when used with Boolean values, <code>&amp;&amp;</code> returns true if both operands are true; otherwise, returns false.
<code>  </code>	<code>expr1    expr2</code>	(Logical OR) Returns <code>expr1</code> if it can be converted to true; otherwise, returns <code>expr2</code> . Thus, when used with Boolean values, <code>  </code> returns true if either operand is true; if both are false, returns false.
<code>!</code>	<code>!expr</code>	(Logical NOT) Returns false if its single operand can be converted to true; otherwise, returns true.

Examples of expressions that can be converted to false are those that evaluate to null, 0, the empty string (“”), or undefined.

The following code shows examples of the `&&` (logical AND) operator.

```
a1=true && true      // t && t returns true
a2=true && false     // t && f returns false
a3=false && true      // f && t returns false
a4=false && (3 == 4)  // f && f returns false
a5="Cat" && "Dog"     // t && t returns Dog
a6=false && "Cat"     // f && t returns false
a7="Cat" && false     // t && f returns false
```

The following code shows examples of the `||` (logical OR) operator.

```
o1=true || true      // t || t returns true
o2=false || true     // f || t returns true
o3=true || false     // t || f returns true
o4=false || (3 == 4) // f || f returns false
o5="Cat" || "Dog"    // t || t returns Cat
o6=false || "Cat"    // f || t returns Cat
o7="Cat" || false    // t || f returns Cat
```

The following code shows examples of the `!` (logical NOT) operator.

```
n1=!true           // !t returns false
n2=!false          // !f returns true
n3=! "Cat"         // !t returns false
```

## Short-Circuit Evaluation

As logical expressions are evaluated left to right, they are tested for possible “short-circuit” evaluation using the following rules:

- `false && anything` is short-circuit evaluated to false.
- `true || anything` is short-circuit evaluated to true.

The rules of logic guarantee that these evaluations are always correct. Note that the *anything* part of the above expressions is not evaluated, so any side effects of doing so do not take effect.

## String Operators

In addition to the comparison operators, which can be used on string values, the concatenation operator `(+)` concatenates two string values together, returning another string that is the union of the two operand strings. For example, `"my " + "string"` returns the string `"my string"`.

The shorthand assignment operator `+=` can also be used to concatenate strings. For example, if the variable `mystring` has the value `"alpha,"` then the expression `mystring += "bet"` evaluates to `"alphabet"` and assigns this value to `mystring`.



## Special Operators

JavaScript provides the following special operators:

- conditional operator
- comma operator
- delete
- new
- this
- typeof
- void

### conditional operator

The conditional operator is the only JavaScript operator that takes three operands. The operator can have one of two values based on a condition. The syntax is:

```
condition ? val1 : val2
```

If *condition* is true, the operator has the value of *val1*. Otherwise it has the value of *val2*. You can use the conditional operator anywhere you would use a standard operator.

For example,

```
status = (age >= 18) ? "adult" : "minor"
```

This statement assigns the value “adult” to the variable *status* if *age* is eighteen or more. Otherwise, it assigns the value “minor” to *status*.

### comma operator

The comma operator (,) simply evaluates both of its operands and returns the value of the second operand. This operator is primarily used inside a `for` loop, to allow multiple variables to be updated each time through the loop.

For example, if *a* is a 2-dimensional array with 10 elements on a side, the following code uses the comma operator to increment two variables at once. The code prints the values of the diagonal elements in the array:

```
for (var i=0, j=9; i <= 9; i++, j--)
    document.writeln("a["+i+", "+j+"] = " + a[i,j])
```

## delete

The delete operator deletes an object, an object's property, or an element at a specified index in an array. Its syntax is:

```
delete objectName
delete objectName.property
delete objectName[index]
delete property // legal only within a with statement
```

where *objectName* is the name of an object, *property* is an existing property, and *index* is an integer representing the location of an element in an array.

The fourth form is legal only within a `with` statement, to delete a property from an object.

You can use the delete operator to delete variables declared implicitly but not those declared with the `var` statement.

If the delete operator succeeds, it sets the property or element to `undefined`. The delete operator returns `true` if the operation is possible; it returns `false` if the operation is not possible.

```
x=42
var y= 43
myobj=new Number()
myobj.h=4      // create property h
delete x       // returns true (can delete if declared implicitly)
delete y       // returns false (cannot delete if declared with var)
delete Math.PI // returns false (cannot delete predefined properties)
delete myobj.h // returns true (can delete user-defined properties)
delete myobj   // returns true (can delete user-defined object)
```

## Deleting array elements

When you delete an array element, the array length is not affected. For example, if you delete `a[3]`, `a[4]` is still `a[4]` and `a[3]` is `undefined`.

When the delete operator removes an array element, that element is no longer in the array. In the following example, `trees[3]` is removed with delete.

```
trees=new Array("redwood","bay","cedar","oak","maple")
delete trees[3]
if (3 in trees) {
    // this does not get executed
}
```

If you want an array element to exist but have an undefined value, use the `undefined` keyword instead of the `delete` operator. In the following example, `trees[3]` is assigned the value `undefined`, but the array element still exists:

```
trees=new Array("redwood","bay","cedar","oak","maple")
trees[3]=undefined
if (3 in trees) {
    // this gets executed
}
```

## new

You can use the `new` operator to create an instance of a user-defined object type or of one of the predefined object types `Array`, `Boolean`, `Date`, `Function`, `Image`, `Number`, `Object`, `Option`, `RegExp`, or `String`. On the server, you can also use it with `DbPool`, `Lock`, `File`, or `SendMail`. Use `new` as follows:

```
objectName = new objectType ( param1 [,param2] ...[,paramN] )
```

You can also create objects using object initializers, as described in “Using Object Initializers” on page 141.

See `new` in the *Server-Side JavaScript Reference* for more information.

## this

Use the `this` keyword to refer to the current object. In general, `this` refers to the calling object in a method. Use `this` as follows:

```
this[.propertyName]
```

**Example 1.** Suppose a function called `validate` validates an object’s value property, given the object and the high and low values:

```
function validate(obj, lowval, hival) {
    if ((obj.value < lowval) || (obj.value > hival))
        alert("Invalid Value!")
}
```

You could call `validate` in each form element’s `onChange` event handler, using `this` to pass it the form element, as in the following example:

```
<B>Enter a number between 18 and 99:</B>
<INPUT TYPE = "text" NAME = "age" SIZE = 3
    onChange="validate(this, 18, 99)">
```

**Example 2.** When combined with the `form` property, `this` can refer to the current object's parent form. In the following example, the form `myForm` contains a `Text` object and a button. When the user clicks the button, the value of the `Text` object is set to the form's name. The button's `onClick` event handler uses `this.form` to refer to the parent form, `myForm`.

```
<FORM NAME="myForm">
Form name:<INPUT TYPE="text" NAME="text1" VALUE="Beluga">
<P>
<INPUT NAME="button1" TYPE="button" VALUE="Show Form Name"
onClick="this.form.text1.value=this.form.name">
</FORM>
```

## typeof

The `typeof` operator is used in either of the following ways:

1. `typeof operand`
2. `typeof (operand)`

The `typeof` operator returns a string indicating the type of the unevaluated operand. `operand` is the string, variable, keyword, or object for which the type is to be returned. The parentheses are optional.

Suppose you define the following variables:

```
var myFun = new Function("5+2")
var shape="round"
var size=1
var today=new Date()
```

The `typeof` operator returns the following results for these variables:

```
typeof myFun is object
typeof shape is string
typeof size is number
typeof today is object
typeof dontExist is undefined
```

For the keywords `true` and `null`, the `typeof` operator returns the following results:

```
typeof true is boolean
typeof null is object
```

For a number or string, the `typeof` operator returns the following results:

```
typeof 62 is number
typeof 'Hello world' is string
```

For property values, the `typeof` operator returns the type of value the property contains:

```
typeof document.lastModified is string
typeof window.length is number
typeof Math.LN2 is number
```

For methods and functions, the `typeof` operator returns results as follows:

```
typeof blur is function
typeof eval is function
typeof parseInt is function
typeof shape.split is function
```

For predefined objects, the `typeof` operator returns results as follows:

```
typeof Date is function
typeof Function is function
typeof Math is function
typeof Option is function
typeof String is function
```

## void

The void operator is used in either of the following ways:

1. `void (expression)`
2. `void expression`

The void operator specifies an expression to be evaluated without returning a value. `expression` is a JavaScript expression to evaluate. The parentheses surrounding the expression are optional, but it is good style to use them.

You can use the void operator to specify an expression as a hypertext link. The expression is evaluated but is not loaded in place of the current document.

The following code creates a hypertext link that does nothing when the user clicks it. When the user clicks the link, `void(0)` evaluates to 0, but that has no effect in JavaScript.

```
<A HREF="javascript:void(0)">Click here to do nothing</A>
```

The following code creates a hypertext link that submits a form when the user clicks it.

```
<A HREF="javascript:void(document.form.submit())">
Click here to submit</A>
```

# Operator Precedence

The *precedence* of operators determines the order they are applied when evaluating an expression. You can override operator precedence by using parentheses.

The following table describes the precedence of operators, from lowest to highest.

Table 5.7 Operator precedence

Operator type	Individual operators
comma	,
assignment	= += -= *= /= %= <<= >>= >>>= &= ^=  =
conditional	? :
logical-or	
logical-and	&&
bitwise-or	
bitwise-xor	^
bitwise-and	&
equality	== !=
relational	< <= > >=
bitwise shift	<< >> >>>
addition/subtraction	+ -
multiply/divide	* / %
negation/increment	! ~ - + ++ -- typeof void delete
call	( )
create instance	new
member	. [ ]

# Regular Expressions

Regular expressions are patterns used to match character combinations in strings. In JavaScript, regular expressions are also objects. These patterns are used with the `exec` and `test` methods of `RegExp`, and with the `match`, `replace`, `search`, and `split` methods of `String`. This chapter describes JavaScript regular expressions.

**JavaScript 1.1 and earlier.** Regular expressions are not available in JavaScript 1.1 and earlier.

This chapter contains the following sections:

- Creating a Regular Expression
- Writing a Regular Expression Pattern
- Working with Regular Expressions
- Examples

## Creating a Regular Expression

You construct a regular expression in one of two ways:

- Using an object initializer, as follows:

```
re = /ab+c/
```

Object initializers provide compilation of the regular expression when the script is evaluated. When the regular expression will remain constant, use this for better performance. Object initializers are discussed in “Using Object Initializers” on page 141.

- Calling the constructor function of the `RegExp` object, as follows:

```
re = new RegExp("ab+c")
```

Using the constructor function provides runtime compilation of the regular expression. Use the constructor function when you know the regular expression pattern will be changing, or you don't know the pattern and are getting it from another source, such as user input. Once you have a defined regular expression, if the regular expression is used throughout the script, and if its source changes, you can use the `compile` method to compile a new regular expression for efficient reuse.

## Writing a Regular Expression Pattern

A regular expression pattern is composed of simple characters, such as `/abc/`, or a combination of simple and special characters, such as `/ab*c/` or `/Chapter (\d+)\.\d*/`. The last example includes parentheses which are used as a memory device. The match made with this part of the pattern is remembered for later use, as described in “Using Parenthesized Substring Matches” on page 113.



## Using Simple Patterns

Simple patterns are constructed of characters for which you want to find a direct match. For example, the pattern `/abc/` matches character combinations in strings only when exactly the characters 'abc' occur together and in that order. Such a match would succeed in the strings "Hi, do you know your abc's?" and "The latest airplane designs evolved from slabcraft." In both cases the match is with the substring 'abc'. There is no match in the string "Grab crab" because it does not contain the substring 'abc'.

## Using Special Characters

When the search for a match requires something more than a direct match, such as finding one or more b's, or finding whitespace, the pattern includes special characters. For example, the pattern `/ab*c/` matches any character combination in which a single 'a' is followed by zero or more 'b's (\* means 0 or more occurrences of the preceding character) and then immediately followed by 'c'. In the string "cbbabbbbcdebc," the pattern matches the substring 'abbbbc'.

The following table provides a complete list and description of the special characters that can be used in regular expressions.

Table 6.1 Special characters in regular expressions.

Character	Meaning
\	<p>Either of the following:</p> <ul style="list-style-type: none"><li>For characters that are usually treated literally, indicates that the next character is special and not to be interpreted literally. For example, <code>/b/</code> matches the character 'b'. By placing a backslash in front of b, that is by using <code>/\b/</code>, the character becomes special to mean match a word boundary.</li><li>For characters that are usually treated specially, indicates that the next character is not special and should be interpreted literally. For example, <code>*</code> is a special character that means 0 or more occurrences of the preceding character should be matched; for example, <code>/a*/</code> means match 0 or more a's. To match <code>*</code> literally, precede the it with a backslash; for example, <code>/a\*/</code> matches 'a*'. </li></ul>
^	<p>Matches beginning of input or line.</p> <p>For example, <code>/^A/</code> does not match the 'A' in "an A," but does match it in "An A."</p>
\$	<p>Matches end of input or line.</p> <p>For example, <code>/t\$/</code> does not match the 't' in "eater", but does match it in "eat"</p>
*	<p>Matches the preceding character 0 or more times.</p> <p>For example, <code>/bo*/</code> matches 'boooo' in "A ghost boooooed" and 'b' in "A bird warbled", but nothing in "A goat grunted".</p>
+	<p>Matches the preceding character 1 or more times. Equivalent to <code>{1, }</code>.</p> <p>For example, <code>/a+/</code> matches the 'a' in "candy" and all the a's in "caaaaaaandy."</p>
?	<p>Matches the preceding character 0 or 1 time.</p> <p>For example, <code>/e?le?/</code> matches the 'el' in "angel" and the 'le' in "angle."</p>

Table 6.1 Special characters in regular expressions. (Continued)

Character	Meaning
.	<p>(The decimal point) matches any single character except the newline character.</p> <p>For example, <code>/ .n/</code> matches 'an' and 'on' in "nay, an apple is on the tree", but not 'nay'.</p>
( <i>x</i> )	<p>Matches 'x' and remembers the match.</p> <p>For example, <code>/(foo)/</code> matches and remembers 'foo' in "foo bar." The matched substring can be recalled from the resulting array's elements <code>[1]</code>, ..., <code>[n]</code>, or from the predefined <code>RegExp</code> object's properties <code>\$1</code>, ..., <code>\$9</code>.</p>
<i>x</i>   <i>y</i>	<p>Matches either 'x' or 'y'.</p> <p>For example, <code>/green red/</code> matches 'green' in "green apple" and 'red' in "red apple."</p>
{ <i>n</i> }	<p>Where <i>n</i> is a positive integer. Matches exactly <i>n</i> occurrences of the preceding character.</p> <p>For example, <code>/a{2}/</code> doesn't match the 'a' in "candy," but it matches all of the a's in "caandy," and the first two a's in "caaandy."</p>
{ <i>n</i> , }	<p>Where <i>n</i> is a positive integer. Matches at least <i>n</i> occurrences of the preceding character.</p> <p>For example, <code>/a{2,}/</code> doesn't match the 'a' in "candy", but matches all of the a's in "caandy" and in "caaaaaaandy."</p>
{ <i>n</i> , <i>m</i> }	<p>Where <i>n</i> and <i>m</i> are positive integers. Matches at least <i>n</i> and at most <i>m</i> occurrences of the preceding character.</p> <p>For example, <code>/a{1,3}/</code> matches nothing in "cndy", the 'a' in "candy," the first two a's in "caandy," and the first three a's in "caaaaaaandy" Notice that when matching "caaaaaaandy", the match is "aaa", even though the original string had more a's in it.</p>
[ <i>xyz</i> ]	<p>A character set. Matches any one of the enclosed characters. You can specify a range of characters by using a hyphen.</p> <p>For example, <code>[abcd]</code> is the same as <code>[a-d]</code>. They match the 'b' in "brisket" and the 'c' in "ache".</p>

Table 6.1 Special characters in regular expressions. (Continued)

Character	Meaning
[^xyz]	<p>A negated or complemented character set. That is, it matches anything that is not enclosed in the brackets. You can specify a range of characters by using a hyphen.</p> <p>For example, [^abc] is the same as [^a-c]. They initially match 'r' in "brisket" and 'h' in "chop."</p>
[\b]	Matches a backspace. (Not to be confused with \b.)
\b	<p>Matches a word boundary, such as a space or a newline character. (Not to be confused with [\b].)</p> <p>For example, /\bn\b/ matches the 'no' in "noonday"; /\wy\b/ matches the 'ly' in "possibly yesterday."</p>
\B	<p>Matches a non-word boundary.</p> <p>For example, /\w\Bn/ matches 'on' in "noonday", and /\y\B\b/ matches 'ye' in "possibly yesterday."</p>
\cX	<p>Where <i>X</i> is a control character. Matches a control character in a string.</p> <p>For example, /\cM/ matches control-M in a string.</p>
\d	<p>Matches a digit character. Equivalent to [0-9].</p> <p>For example, /\d/ or /[0-9]/ matches '2' in "B2 is the suite number."</p>
\D	<p>Matches any non-digit character. Equivalent to [^0-9].</p> <p>For example, /\D/ or /^[^0-9]/ matches 'B' in "B2 is the suite number."</p>
\f	Matches a form-feed.
\n	Matches a linefeed.
\r	Matches a carriage return.
\s	<p>Matches a single white space character, including space, tab, form feed, line feed. Equivalent to [ \f\n\r\t\v].</p> <p>For example, /\s\w*/ matches ' bar' in "foo bar."</p>

Table 6.1 Special characters in regular expressions. (Continued)

Character	Meaning
<code>\s</code>	Matches a single character other than white space. Equivalent to <code>[^\f\n\r\t\v]</code> .  For example, <code>/\s\w*/</code> matches 'foo' in "foo bar."
<code>\t</code>	Matches a tab
<code>\v</code>	Matches a vertical tab.
<code>\w</code>	Matches any alphanumeric character including the underscore. Equivalent to <code>[A-Za-z0-9_]</code> .  For example, <code>/\w/</code> matches 'a' in "apple," '5' in "\$5.28," and '3' in "3D."
<code>\W</code>	Matches any non-word character. Equivalent to <code>[^A-Za-z0-9_]</code> .  For example, <code>/\W/</code> or <code>/[^A-Za-z0-9_]/</code> matches '%' in "50%."
<code>\n</code>	Where <i>n</i> is a positive integer. A back reference to the last substring matching the <i>n</i> parenthetical in the regular expression (counting left parentheses).  For example, <code>/apple(,)\sorange\1/</code> matches 'apple, orange,' in "apple, orange, cherry, peach." A more complete example follows this table.  <b>Note:</b> If the number of left parentheses is less than the number specified in <code>\n</code> , the <code>\n</code> is taken as an octal escape as described in the next row.
<code>\o<sub>octal</sub></code> <code>\x<sub>hex</sub></code>	Where <code>\o<sub>octal</sub></code> is an octal escape value or <code>\x<sub>hex</sub></code> is a hexadecimal escape value. Allows you to embed ASCII codes into regular expressions.

## Using Parentheses

Parentheses around any part of the regular expression pattern cause that part of the matched substring to be remembered. Once remembered, the substring can be recalled for other use, as described in “Using Parenthesized Substring Matches” on page 113.

For example, the pattern `/Chapter (\d+)\.\d*/` illustrates additional escaped and special characters and indicates that part of the pattern should be remembered. It matches precisely the characters 'Chapter ' followed by one or more numeric characters (`\d` means any numeric character and `+` means 1 or more times), followed by a decimal point (which in itself is a special character; preceding the decimal point with `\` means the pattern must look for the literal character `.`), followed by any numeric character 0 or more times (`\d` means numeric character, `*` means 0 or more times). In addition, parentheses are used to remember the first matched numeric characters.

This pattern is found in "Open Chapter 4.3, paragraph 6" and '4' is remembered. The pattern is not found in "Chapter 3 and 4", because that string does not have a period after the '3'.

## Working with Regular Expressions

Regular expressions are used with the `RegExp` methods `test` and `exec` and with the `String` methods `match`, `replace`, `search`, and `split`. These methods are explained in detail in the *Server-Side JavaScript Reference*.

Table 6.2 Methods that use regular expressions

Method	Description
<code>exec</code>	A <code>RegExp</code> method that executes a search for a match in a string. It returns an array of information.
<code>test</code>	A <code>RegExp</code> method that tests for a match in a string. It returns true or false.
<code>match</code>	A <code>String</code> method that executes a search for a match in a string. It returns an array of information or null on a mismatch.
<code>search</code>	A <code>String</code> method that tests for a match in a string. It returns the index of the match, or -1 if the search fails.

Table 6.2 Methods that use regular expressions

Method	Description
<code>replace</code>	A <code>String</code> method that executes a search for a match in a string, and replaces the matched substring with a replacement substring.
<code>split</code>	A <code>String</code> method that uses a regular expression or a fixed string to break a string into an array of substrings.

When you want to know whether a pattern is found in a string, use the `test` or `search` method; for more information (but slower execution) use the `exec` or `match` methods. If you use `exec` or `match` and if the match succeeds, these methods return an array and update properties of the associated regular expression object and also of the predefined regular expression object, `RegExp`. If the match fails, the `exec` method returns `null` (which converts to `false`).

In the following example, the script uses the `exec` method to find a match in a string.

```
<SCRIPT LANGUAGE="JavaScript1.2">
myRe=/d(b+)d/g;
myArray = myRe.exec("cdbbdsbz");
</SCRIPT>
```

If you do not need to access the properties of the regular expression, an alternative way of creating `myArray` is with this script:

```
<SCRIPT LANGUAGE="JavaScript1.2">
myArray = /d(b+)d/g.exec("cdbbdsbz");
</SCRIPT>
```

If you want to be able to recompile the regular expression, yet another alternative is this script:

```
<SCRIPT LANGUAGE="JavaScript1.2">
myRe= new RegExp ("d(b+)d", "g");
myArray = myRe.exec("cdbbdsbz");
</SCRIPT>
```

With these scripts, the match succeeds and returns the array and updates the properties shown in the following table.

Table 6.3 Results of regular expression execution.

Object	Property or index	Description	In this example
myArray		The matched string and all remembered substrings	[ "dbbd" , "bb" ]
	index	The 0-based index of the match in the input string	1
	input	The original string	"cdbbdsbz"
	[0]	The last matched characters	"dbbd"
myRe	lastIndex	The index at which to start the next match. (This property is set only if the regular expression uses the <code>g</code> option, described in “Executing a Global Search and Ignoring Case” on page 115.)	5
	source	The text of the pattern	"d(b+)d"
RegExp	lastMatch	The last matched characters	"dbbd"
	leftContext	The substring preceding the most recent match	"c"
	rightContext	The substring following the most recent match	"bsbz"

`RegExp.leftContext` and `RegExp.rightContext` can be computed from the other values. `RegExp.leftContext` is equivalent to:

```
myArray.input.substring(0, myArray.index)
```

and `RegExp.rightContext` is equivalent to:

```
myArray.input.substring(myArray.index + myArray[0].length)
```

As shown in the second form of this example, you can use the a regular expression created with an object initializer without assigning it to a variable. If you do, however, every occurrence is a new regular expression. For this reason, if you use this form without assigning it to a variable, you cannot subsequently access the properties of that regular expression. For example, assume you have this script:

```
<SCRIPT LANGUAGE="JavaScript1.2">
myRe=/d(b+)d/g;
myArray = myRe.exec("cdbbdsbz");
document.writeln("The value of lastIndex is " + myRe.lastIndex);
</SCRIPT>
```



This script displays:

The value of `lastIndex` is 5

However, if you have this script:

```
<SCRIPT LANGUAGE="JavaScript1.2">
myArray = /d(b+)d/g.exec("cdbbdsbz");
document.writeln("The value of lastIndex is " + /d(b+)d/g.lastIndex);
</SCRIPT>
```

It displays:

The value of `lastIndex` is 0

The occurrences of `/d(b+)d/g` in the two statements are different regular expression objects and hence have different values for their `lastIndex` property. If you need to access the properties of a regular expression created with an object initializer, you should first assign it to a variable.

## Using Parenthesized Substring Matches

Including parentheses in a regular expression pattern causes the corresponding submatch to be remembered. For example, `/a(b)c/` matches the characters 'abc' and remembers 'b'. To recall these parenthesized substring matches, use the `RegExp` properties `$1`, ..., `$9` or the `Array` elements `[1]`, ..., `[n]`.

The number of possible parenthesized substrings is unlimited. The predefined `RegExp` object holds up to the last nine and the returned array holds all that were found. The following examples illustrate how to use parenthesized substring matches.

**Example 1.** The following script uses the `replace` method to switch the words in the string. For the replacement text, the script uses the values of the `$1` and `$2` properties.

```
<SCRIPT LANGUAGE="JavaScript1.2">
re = /(\w+)\s(\w+)/;
str = "John Smith";
newstr = str.replace(re, "$2, $1");
document.write(newstr)
</SCRIPT>
```

This prints "Smith, John".

**Example 2.** In the following example, `RegExp.input` is set by the `Change` event. In the `getInfo` function, the `exec` method uses the value of `RegExp.input` as its argument. Note that `RegExp` must be prepended to its `$` properties (because they appear outside the replacement string). (Example 3 is a more efficient, though possibly more cryptic, way to accomplish the same thing.)

```
<HTML>

<SCRIPT LANGUAGE="JavaScript1.2">
function getInfo(){
    re = /(\w+)\s(\d+)/
    re.exec();
    window.alert(RegExp.$1 + ", your age is " + RegExp.$2);
}
</SCRIPT>

Enter your first name and your age, and then press Enter.

<FORM>
<INPUT TYPE="text" NAME="NameAge" onChange="getInfo(this);">
</FORM>

</HTML>
```

**Example 3.** The following example is similar to Example 2. Instead of using the `RegExp.$1` and `RegExp.$2`, this example creates an array and uses `a[1]` and `a[2]`. It also uses the shortcut notation for using the `exec` method.

```
<HTML>

<SCRIPT LANGUAGE="JavaScript1.2">
function getInfo(){
    a = /(\w+)\s(\d+)/();
    window.alert(a[1] + ", your age is " + a[2]);
}
</SCRIPT>

Enter your first name and your age, and then press Enter.

<FORM>
<INPUT TYPE="text" NAME="NameAge" onChange="getInfo(this);">
</FORM>

</HTML>
```

## Executing a Global Search and Ignoring Case

Regular expressions have two optional flags that allow for global and case insensitive searching. To indicate a global search, use the `g` flag. To indicate a case insensitive search, use the `i` flag. These flags can be used separately or together in either order, and are included as part of the regular expression.

To include a flag with the regular expression, use this syntax:

```
re = /pattern/[g|i|gi]
re = new RegExp("pattern", ['g'|'i'|'gi'])
```

Note that the flags, `i` and `g`, are an integral part of a regular expression. They cannot be added or removed later.

For example, `re = /\w+\s/g` creates a regular expression that looks for one or more characters followed by a space, and it looks for this combination throughout the string.

```
<SCRIPT LANGUAGE="JavaScript1.2">
re = /\w+\s/g;
str = "fee fi fo fum";
myArray = str.match(re);
document.write(myArray);
</SCRIPT>
```

This displays ["fee ", "fi ", "fo "]. In this example, you could replace the line:

```
re = /\w+\s/g;
```

with:

```
re = new RegExp("\\w+\\s", "g");
```

and get the same result.

# Examples

The following examples show some uses of regular expressions.

## Changing the Order in an Input String

The following example illustrates the formation of regular expressions and the use of `string.split()` and `string.replace()`. It cleans a roughly formatted input string containing names (first name first) separated by blanks, tabs and exactly one semicolon. Finally, it reverses the name order (last name first) and sorts the list.

```
<SCRIPT LANGUAGE="JavaScript1.2">

// The name string contains multiple spaces and tabs,
// and may have multiple spaces between first and last names.
names = new String ( "Harry Trump ;Fred Barney; Helen Rigby ;\
    Bill Abel ;Chris Hand ")

document.write ("----- Original String" + "<BR>" + "<BR>")
document.write (names + "<BR>" + "<BR>")

// Prepare two regular expression patterns and array storage.
// Split the string into array elements.

// pattern: possible white space then semicolon then possible white space
pattern = /\s*;\s*/

// Break the string into pieces separated by the pattern above and
// and store the pieces in an array called nameList
nameList = names.split (pattern)

// new pattern: one or more characters then spaces then characters.
// Use parentheses to "memorize" portions of the pattern.
// The memorized portions are referred to later.
pattern = /(\w+)\s+(\w+)/

// New array for holding names being processed.
bySurnameList = new Array;

// Display the name array and populate the new array
// with comma-separated names, last first.
//
// The replace method removes anything matching the pattern
// and replaces it with the memorized string—second memorized portion
// followed by comma space followed by first memorized portion.
//
// The variables $1 and $2 refer to the portions
// memorized while matching the pattern.
```

```

document.write ("----- After Split by Regular Expression" + "<BR>")
for ( i = 0; i < nameList.length; i++) {
    document.write (nameList[i] + "<BR>")
    bySurnameList[i] = nameList[i].replace (pattern, "$2, $1")
}

// Display the new array.
document.write ("----- Names Reversed" + "<BR>")
for ( i = 0; i < bySurnameList.length; i++) {
    document.write (bySurnameList[i] + "<BR>")
}

// Sort by last name, then display the sorted array.
bySurnameList.sort()
document.write ("----- Sorted" + "<BR>")
for ( i = 0; i < bySurnameList.length; i++) {
    document.write (bySurnameList[i] + "<BR>")
}

document.write ("----- End" + "<BR>")

</SCRIPT>

```

## Using Special Characters to Verify Input

In the following example, a user enters a phone number. When the user presses Enter, the script checks the validity of the number. If the number is valid (matches the character sequence specified by the regular expression), the script posts a window thanking the user and confirming the number. If the number is invalid, the script posts a window informing the user that the phone number is not valid.

The regular expression looks for zero or one open parenthesis `\(?`, followed by three digits `\d{3}`, followed by zero or one close parenthesis `\)?`, followed by one dash, forward slash, or decimal point and when found, remember the character `([-\/\.] )`, followed by three digits `\d{3}`, followed by the remembered match of a dash, forward slash, or decimal point `\1`, followed by four digits `\d{4}`.

The Change event activated when the user presses Enter sets the value of `RegExp.input`.

```
<HTML>
<SCRIPT LANGUAGE = "JavaScript1.2">

re = /\(?\d{3}\)?([-\./])\d{3}\1\d{4}/

function testInfo() {
    OK = re.exec()
    if (!OK)
        window.alert (RegExp.input +
            " isn't a phone number with area code!")
    else
        window.alert ("Thanks, your phone number is " + OK[0])
}

</SCRIPT>

Enter your phone number (with area code) and then press Enter.
<FORM>
<INPUT TYPE="text" NAME="Phone" onChange="testInfo(this);">
</FORM>

</HTML>
```

# Statements

JavaScript supports a compact set of statements that you can use to incorporate a great deal of interactivity in Web pages. This chapter provides an overview of these statements.

This chapter contains the following sections, which provide a brief overview of each statement:

- Conditional Statements: `if...else` and `switch`
- Loop Statements: `for`, `while`, `do while`, `label`, `break`, and `continue` (`label` is not itself a looping statement, but is frequently used with these statements)
- Object Manipulation Statements: `for...in` and `with`
- Comments

Any expression is also a statement. See Chapter 5, “Expressions and Operators,” for complete information about statements.

Use the semicolon (;) character to separate statements in JavaScript code.

See the *Server-Side JavaScript Reference* for details about the statements in this chapter.

# Conditional Statements

A conditional statement is a set of commands that executes if a specified condition is true. JavaScript supports two conditional statements: `if...else` and `switch`.

## if...else Statement

Use the `if` statement to perform certain statements if a logical condition is true; use the optional `else` clause to perform other statements if the condition is false. An `if` statement looks as follows:

```
if (condition) {  
    statements1  
}  
[else {  
    statements2  
} ]
```

The condition can be any JavaScript expression that evaluates to true or false. The statements to be executed can be any JavaScript statements, including further nested `if` statements. If you want to use more than one statement after an `if` or `else` statement, you must enclose the statements in curly braces, `{}`.

**Example.** In the following example, the function `checkData` returns true if the number of characters in a `Text` object is three; otherwise, it displays an alert and returns false.

```
function checkData () {  
    if (document.form1.threeChar.value.length == 3) {  
        return true  
    } else {  
        alert("Enter exactly three characters. " +  
            document.form1.threeChar.value + " is not valid.")  
        return false  
    }  
}
```



## switch Statement

A `switch` statement allows a program to evaluate an expression and attempt to match the expression's value to a case label. If a match is found, the program executes the associated statement. A `switch` statement looks as follows:

```
switch (expression){
    case label :
        statement;
        break;
    case label :
        statement;
        break;
    ...
    default : statement;
}
```

The program first looks for a label matching the value of expression and then executes the associated statement. If no matching label is found, the program looks for the optional default statement, and if found, executes the associated statement. If no default statement is found, the program continues execution at the statement following the end of `switch`.

The optional `break` statement associated with each case label ensures that the program breaks out of `switch` once the matched statement is executed and continues execution at the statement following `switch`. If `break` is omitted, the program continues execution at the next statement in the `switch` statement.

**Example.** In the following example, if `expr` evaluates to "Bananas", the program matches the value with case "Bananas" and executes the associated statement. When `break` is encountered, the program terminates `switch` and executes the statement following `switch`. If `break` were omitted, the statement for case "Cherries" would also be executed.

```
switch (expr) {
    case "Oranges" :
        document.write("Oranges are $0.59 a pound.<BR>");
        break;
    case "Apples" :
        document.write("Apples are $0.32 a pound.<BR>");
        break;
    case "Bananas" :
        document.write("Bananas are $0.48 a pound.<BR>");
        break;
    case "Cherries" :
        document.write("Cherries are $3.00 a pound.<BR>");
        break;
}
```

```

    default :
        document.write("Sorry, we are out of " + i + "<BR>");
    }
    document.write("Is there anything else you'd like?<BR>");

```

## Loop Statements

A loop is a set of commands that executes repeatedly until a specified condition is met. JavaScript supports the `for`, `do while`, `while`, and `label` loop statements (`label` is not itself a looping statement, but is frequently used with these statements). In addition, you can use the `break` and `continue` statements within loop statements.

Another statement, `for...in`, executes statements repeatedly but is used for object manipulation. See “Object Manipulation Statements” on page 128.

### for Statement

A `for` loop repeats until a specified condition evaluates to false. The JavaScript `for` loop is similar to the Java and C `for` loop. A `for` statement looks as follows:

```

for ([initialExpression]; [condition]; [incrementExpression]) {
    statements
}

```

When a `for` loop executes, the following occurs:

1. The initializing expression `initial-expression`, if any, is executed. This expression usually initializes one or more loop counters, but the syntax allows an expression of any degree of complexity.
2. The `condition` expression is evaluated. If the value of `condition` is true, the loop statements execute. If the value of `condition` is false, the `for` loop terminates.
3. The `statements` execute.
4. The update expression `incrementExpression` executes, and control returns to Step 2.

**Example.** The following function contains a `for` statement that counts the number of selected options in a scrolling list (a `Select` object that allows multiple selections). The `for` statement declares the variable `i` and initializes it to zero. It checks that `i` is less than the number of options in the `Select` object, performs the succeeding `if` statement, and increments `i` by one after each pass through the loop.

```
<SCRIPT>
function howMany(selectObject) {
    var numberSelected=0
    for (var i=0; i < selectObject.options.length; i++) {
        if (selectObject.options[i].selected==true)
            numberSelected++
    }
    return numberSelected
}
</SCRIPT>

<FORM NAME="selectForm">
<P><B>Choose some music types, then click the button below:</B>
<BR><SELECT NAME="musicTypes" MULTIPLE>
<OPTION SELECTED> R&B
<OPTION> Jazz
<OPTION> Blues
<OPTION> New Age
<OPTION> Classical
<OPTION> Opera
</SELECT>
<P><INPUT TYPE="button" VALUE="How many are selected?"
onClick="alert ('Number of options selected: ' +
howMany(document.selectForm.musicTypes))">
</FORM>
```

## do...while Statement

The `do...while` statement repeats until a specified condition evaluates to false. A `do...while` statement looks as follows:

```
do {  
    statement  
} while (condition)
```

`statement` executes once before the condition is checked. If `condition` returns `true`, the statement executes again. At the end of every execution, the condition is checked. When the condition returns `false`, execution stops and control passes to the statement following `do...while`.

**Example.** In the following example, the `do` loop iterates at least once and reiterates until `i` is no longer less than 5.

```
do {  
    i+=1;  
    document.write(i);  
} while (i<5);
```

## while Statement

A `while` statement executes its statements as long as a specified condition evaluates to true. A `while` statement looks as follows:

```
while (condition) {  
    statements  
}
```

If the condition becomes false, the statements within the loop stop executing and control passes to the statement following the loop.

The condition test occurs before the statements in the loop are executed. If the condition returns true, the statements are executed and the condition is tested again. If the condition returns false, execution stops and control is passed to the statement following `while`.

**Example 1.** The following `while` loop iterates as long as `n` is less than three:

```
n = 0
x = 0
while( n < 3 ) {
    n ++
    x += n
}
```

With each iteration, the loop increments `n` and adds that value to `x`. Therefore, `x` and `n` take on the following values:

- After the first pass: `n = 1` and `x = 1`
- After the second pass: `n = 2` and `x = 3`
- After the third pass: `n = 3` and `x = 6`

After completing the third pass, the condition `n < 3` is no longer true, so the loop terminates.

**Example 2: infinite loop.** Make sure the condition in a loop eventually becomes false; otherwise, the loop will never terminate. The statements in the following `while` loop execute forever because the condition never becomes false:

```
while (true) {
    alert("Hello, world") }
```

## label Statement

A label provides a statement with an identifier that lets you refer to it elsewhere in your program. For example, you can use a label to identify a loop, and then use the `break` or `continue` statements to indicate whether a program should interrupt the loop or continue its execution.

The syntax of the label statement looks like the following:

```
label :
    statement
```

The value of `label` may be any JavaScript identifier that is not a reserved word. The `statement` that you identify with a label may be any type.

**Example.** In this example, the label `markLoop` identifies a `while` loop.

```
markLoop:
while (theMark == true)
    doSomething();
}
```

## break Statement

Use the `break` statement to terminate a loop, `switch`, or label statement.

- When you use `break` with a `while`, `do-while`, `for`, or `switch` statement, `break` terminates the innermost enclosing loop or `switch` immediately and transfers control to the following statement.
- When you use `break` within an enclosing label statement, it terminates the statement and transfers control to the following statement. If you specify a label when you issue the `break`, the `break` statement terminates the specified statement.

The syntax of the `break` statement looks like the following:

1. `break`
2. `break [label]`

The first form of the syntax terminates the innermost enclosing loop, `switch`, or label; the second form of the syntax terminates the specified enclosing label statement.

**Example.** The following example iterates through the elements in an array until it finds the index of an element whose value is `theValue`:

```
for (i = 0; i < a.length; i++) {
    if (a[i] = theValue);
        break;
}
```

## continue Statement

The `continue` statement can be used to restart a `while`, `do-while`, `for`, or `label` statement.

- In a `while` or `for` statement, `continue` terminates the current loop and continues execution of the loop with the next iteration. In contrast to the `break` statement, `continue` does not terminate the execution of the loop entirely. In a `while` loop, it jumps back to the condition. In a `for` loop, it jumps to the `increment-expression`.
- In a `label` statement, `continue` is followed by a label that identifies a `label` statement. This type of `continue` restarts a `label` statement or continues execution of a labelled loop with the next iteration. `continue` must be in a looping statement identified by the label used by `continue`.

The syntax of the `continue` statement looks like the following:

```
1. continue
2. continue [label]
```

**Example 1.** The following example shows a `while` loop with a `continue` statement that executes when the value of `i` is three. Thus, `n` takes on the values one, three, seven, and twelve.

```
i = 0
n = 0
while (i < 5) {
    i++
    if (i == 3)
        continue
    n += i
}
```

**Example 2.** A statement labeled `checkiandj` contains a statement labeled `checkj`. If `continue` is encountered, the program terminates the current iteration of `checkj` and begins the next iteration. Each time `continue` is encountered, `checkj` reiterates until its condition returns `false`. When `false` is returned, the remainder of the `checkiandj` statement is completed, and `checkiandj` reiterates until its condition returns `false`. When `false` is returned, the program continues at the statement following `checkiandj`.

If `continue` had a label of `checkiandj`, the program would continue at the top of the `checkiandj` statement.

```
checkiandj :
  while (i<4) {
    document.write(i + "<BR>");
    i+=1;
    checkj :
      while (j>4) {
        document.write(j + "<BR>");
        j-=1;
        if ((j%2)==0);
          continue checkj;
        document.write(j + " is odd.<BR>");
      }
    document.write("i = " + i + "<br>");
    document.write("j = " + j + "<br>");
  }
```

## Object Manipulation Statements

JavaScript uses the `for...in` and `with` statements to manipulate objects.

### for...in Statement

The `for...in` statement iterates a specified variable over all the properties of an object. For each distinct property, JavaScript executes the specified statements. A `for...in` statement looks as follows:

```
for (variable in object) {
  statements }
```

**Example.** The following function takes as its argument an object and the object's name. It then iterates over all the object's properties and returns a string that lists the property names and their values.

```
function dump_props(obj, obj_name) {
  var result = ""
  for (var i in obj) {
    result += obj_name + "." + i + " = " + obj[i] + "<BR>"
  }
  result += "<HR>"
  return result
}
```



For an object `car` with properties `make` and `model`, `result` would be:

```
car.make = Ford
car.model = Mustang
```

## with Statement

The `with` statement establishes the default object for a set of statements. JavaScript looks up any unqualified names within the set of statements to determine if the names are properties of the default object. If an unqualified name matches a property, then the property is used in the statement; otherwise, a local or global variable is used.

A `with` statement looks as follows:

```
with (object){
    statements
}
```

**Example.** The following `with` statement specifies that the `Math` object is the default object. The statements following the `with` statement refer to the `PI` property and the `cos` and `sin` methods, without specifying an object. JavaScript assumes the `Math` object for these references.

```
var a, x, y
var r=10
with (Math) {
    a = PI * r * r
    x = r * cos(PI)
    y = r * sin(PI/2)
}
```

# Comments

Comments are author notations that explain what a script does. Comments are ignored by the interpreter. JavaScript supports Java-style comments:

- Comments on a single line are preceded by a double-slash (//).
- Comments that span multiple lines are preceded by /\* and followed by \*/:

**Example.** The following example shows two comments:

```
// This is a single-line comment.  
  
/* This is a multiple-line comment. It can be of any length, and  
you can put whatever you want here. */
```

# Functions

Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure—a set of statements that performs a specific task. To use a function, you must first define it; then your script can call it.

This chapter contains the following sections:

- Defining Functions
- Calling Functions
- Using the arguments Array
- Predefined Functions

## Defining Functions

A function definition consists of the `function` keyword, followed by

- The name of the function.
- A list of arguments to the function, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, enclosed in curly braces, `{ }`. The statements in a function can include calls to other functions defined in the current application.

For example, the following code defines a simple function named `square`:

```
function square(number) {
    return number * number;
}
```

The function `square` takes one argument, called `number`. The function consists of one statement that indicates to return the argument of the function multiplied by itself. The `return` statement specifies the value returned by the function.

```
return number * number
```

All parameters are passed to functions *by value*; the value is passed to the function, but if the function changes the value of the parameter, this change is not reflected globally or in the calling function. However, if you pass an object as a parameter to a function and the function changes the object's properties, that change is visible outside the function, as shown in the following example:

```
function myFunc(theObject) {
    theObject.make="Toyota"
}

mycar = {make:"Honda", model:"Accord", year:1998}
x=mycar.make      // returns Honda
myFunc(mycar)     // pass object mycar to the function
y=mycar.make      // returns Toyota (prop was changed by the function)
```

In addition to defining functions as described here, you can also define Function objects, as described in “Function Object” on page 155.

A *method* is a function associated with an object. You'll learn more about objects and methods in Chapter 9, “Working with Objects.”

## Calling Functions

In a server-side JavaScript application, you can use any function compiled with the application.

Defining a function does not execute it. Defining the function simply names the function and specifies what to do when the function is called. *Calling* the function actually performs the specified actions with the indicated parameters. For example, if you define the function `square`, you could call it as follows.

```
square(5)
```

The preceding statement calls the function with an argument of five. The function executes its statements and returns the value twenty-five.

The arguments of a function are not limited to strings and numbers. You can pass whole objects to a function, too. The `show_props` function (defined in “Objects and Properties” on page 140) is an example of a function that takes an object as an argument.

A function can even be recursive, that is, it can call itself. For example, here is a function that computes factorials:

```
function factorial(n) {
  if ((n == 0) || (n == 1))
    return 1
  else {
    result = (n * factorial(n-1) )
    return result
  }
}
```

You could then compute the factorials of one through five as follows:

```
a=factorial(1) // returns 1
b=factorial(2) // returns 2
c=factorial(3) // returns 6
d=factorial(4) // returns 24
e=factorial(5) // returns 120
```

## Using the arguments Array

The arguments of a function are maintained in an array. Within a function, you can address the parameters passed to it as follows:

```
arguments[i]
functionName.arguments[i]
```

where *i* is the ordinal number of the argument, starting at zero. So, the first argument passed to a function would be `arguments[0]`. The total number of arguments is indicated by `arguments.length`.

Using the `arguments` array, you can call a function with more arguments than it is formally declared to accept. This is often useful if you don't know in advance how many arguments will be passed to the function. You can use `arguments.length` to determine the number of arguments actually passed to the function, and then treat each argument using the `arguments` array.

For example, consider a function that concatenates several strings. The only formal argument for the function is a string that specifies the characters that separate the items to concatenate. The function is defined as follows:

```
function myConcat(separator) {
    result="" // initialize list
    // iterate through arguments
    for (var i=1; i<arguments.length; i++) {
        result += arguments[i] + separator
    }
    return result
}
```

You can pass any number of arguments to this function, and it creates a list using each argument as an item in the list.

```
// returns "red, orange, blue, "
myConcat(", ", "red", "orange", "blue")

// returns "elephant; giraffe; lion; cheetah;"
myConcat("; ", "elephant", "giraffe", "lion", "cheetah")

// returns "sage. basil. oregano. pepper. parsley. "
myConcat(". ", "sage", "basil", "oregano", "pepper", "parsley")
```

See the Function object in the *Server-Side JavaScript Reference* for more information.

## Predefined Functions

JavaScript has several top-level predefined functions:

- `eval`
- `isFinite`
- `isNaN`
- `parseInt` and `parseFloat`
- `Number` and `String`
- `escape` and `unescape`

The following sections introduce these functions. See the *Server-Side JavaScript Reference* for detailed information on all of these functions.

## eval Function

The `eval` function evaluates a string of JavaScript code without reference to a particular object. The syntax of `eval` is:

```
eval(expr)
```

where *expr* is a string to be evaluated.

If the string represents an expression, `eval` evaluates the expression. If the argument represents one or more JavaScript statements, `eval` performs the statements. Do not call `eval` to evaluate an arithmetic expression; JavaScript evaluates arithmetic expressions automatically.

## isFinite Function

The `isFinite` function evaluates an argument to determine whether it is a finite number. The syntax of `isFinite` is:

```
isFinite(number)
```

where *number* is the number to evaluate.

If the argument is NaN, positive infinity or negative infinity, this method returns `false`, otherwise it returns `true`.

The following code checks client input to determine whether it is a finite number.

```
if(isFinite(ClientInput) == true)
{
    /* take specific steps */
}
```

## isNaN Function

The `isNaN` function evaluates an argument to determine if it is “NaN” (not a number). The syntax of `isNaN` is:

```
isNaN(testValue)
```

where `testValue` is the value you want to evaluate.

The `parseFloat` and `parseInt` functions return “NaN” when they evaluate a value that is not a number. `isNaN` returns true if passed “NaN,” and false otherwise.

The following code evaluates `floatValue` to determine if it is a number and then calls a procedure accordingly:

```
floatValue=parseFloat(toFloat)

if (isNaN(floatValue)) {
    notFloat()
} else {
    isFloat()
}
```

## parseInt and parseFloat Functions

The two “parse” functions, `parseInt` and `parseFloat`, return a numeric value when given a string as an argument.

The syntax of `parseFloat` is

```
parseFloat(str)
```

where `parseFloat` parses its argument, the string `str`, and attempts to return a floating-point number. If it encounters a character other than a sign (+ or -), a numeral (0-9), a decimal point, or an exponent, then it returns the value up to that point and ignores that character and all succeeding characters. If the first character cannot be converted to a number, it returns “NaN” (not a number).

The syntax of `parseInt` is

```
parseInt(str [, radix])
```



`parseInt` parses its first argument, the string `str`, and attempts to return an integer of the specified `radix` (base), indicated by the second, optional argument, `radix`. For example, a radix of ten indicates to convert to a decimal number, eight octal, sixteen hexadecimal, and so on. For radices above ten, the letters of the alphabet indicate numerals greater than nine. For example, for hexadecimal numbers (base 16), A through F are used.

If `parseInt` encounters a character that is not a numeral in the specified radix, it ignores it and all succeeding characters and returns the integer value parsed up to that point. If the first character cannot be converted to a number in the specified radix, it returns “NaN.” The `parseInt` function truncates the string to integer values.

## Number and String Functions

The `Number` and `String` functions let you convert an object to a number or a string. The syntax of these functions is:

```
Number(objRef)
String(objRef)
```

where `objRef` is an object reference.

The following example converts the `Date` object to a readable string.

```
D = new Date (430054663215)
// The following returns
// "Thu Aug 18 04:37:43 GMT-0700 (Pacific Daylight Time) 1983"
x = String(D)
```

## escape and unescape Functions

The `escape` and `unescape` functions let you encode and decode strings. The `escape` function returns the hexadecimal encoding of an argument in the ISO Latin character set. The `unescape` function returns the ASCII string for the specified hexadecimal encoding value.

The syntax of these functions is:

```
escape(string)
unescape(string)
```

These functions are used primarily with server-side JavaScript to encode and decode name/value pairs in URLs.

# Working with Objects

JavaScript is designed on a simple object-based paradigm. An object is a construct with properties that are JavaScript variables or other objects. An object also has functions associated with it that are known as the object's *methods*. In addition to objects that are predefined in the Navigator client and the server, you can define your own objects.

This chapter describes how to use objects, properties, functions, and methods, and how to create your own objects.

This chapter contains the following sections:

- Objects and Properties
- Creating New Objects
- Predefined Core Objects

# Objects and Properties

A JavaScript object has properties associated with it. You access the properties of an object with a simple notation:

```
objectName.propertyName
```

Both the object name and property name are case sensitive. You define a property by assigning it a value. For example, suppose there is an object named `myCar` (for now, just assume the object already exists). You can give it properties named `make`, `model`, and `year` as follows:

```
myCar.make = "Ford"
myCar.model = "Mustang"
myCar.year = 1969;
```

An array is an ordered set of values associated with a single variable name. Properties and arrays in JavaScript are intimately related; in fact, they are different interfaces to the same data structure. So, for example, you could access the properties of the `myCar` object as follows:

```
myCar["make"] = "Ford"
myCar["model"] = "Mustang"
myCar["year"] = 1967
```

This type of array is known as an *associative array*, because each index element is also associated with a string value. To illustrate how this works, the following function displays the properties of the object when you pass the object and the object's name as arguments to the function:

```
function show_props(obj, obj_name) {
    var result = ""
    for (var i in obj)
        result += obj_name + "." + i + " = " + obj[i] + "\n"
    return result
}
```

So, the function call `show_props(myCar, "myCar")` would return the following:

```
myCar.make = Ford
myCar.model = Mustang
myCar.year = 1967
```

# Creating New Objects

JavaScript has a number of predefined objects. In addition, you can create your own objects. In JavaScript 1.2, you can create an object using an object initializer. Alternatively, you can first create a constructor function and then instantiate an object using that function and the `new` operator.

## Using Object Initializers

In addition to creating objects using a constructor function, you can create objects using an object initializer. Using object initializers is sometimes referred to as creating objects with literal notation. “Object initializer” is consistent with the terminology used by C++.

The syntax for an object using an object initializer is:

```
objectName = {property1:value1, property2:value2,..., propertyN:valueN}
```

where `objectName` is the name of the new object, each `propertyI` is an identifier (either a name, a number, or a string literal), and each `valueI` is an expression whose value is assigned to the `propertyI`. The `objectName` and assignment is optional. If you do not need to refer to this object elsewhere, you do not need to assign it to a variable.

If an object is created with an object initializer in a top-level script, JavaScript interprets the object each time it evaluates the expression containing the object literal. In addition, an initializer used in a function is created each time the function is called.

The following statement creates an object and assigns it to the variable `x` if and only if the expression `cond` is true.

```
if (cond) x = {hi:"there"}
```

The following example creates `myHonda` with three properties. Note that the `engine` property is also an object with its own properties.

```
myHonda = {color:"red",wheels:4,engine:{cylinders:4,size:2.2}}
```

You can also use object initializers to create arrays. See “Array Literals” on page 81.

**JavaScript 1.1 and earlier.** You cannot use object initializers. You can create objects only using their constructor functions or using a function supplied by some other object for that purpose. See “Using a Constructor Function” on page 142.

## Using a Constructor Function

Alternatively, you can create an object with these two steps:

1. Define the object type by writing a constructor function.
2. Create an instance of the object with `new`.

To define an object type, create a function for the object type that specifies its name, properties, and methods. For example, suppose you want to create an object type for cars. You want this type of object to be called `car`, and you want it to have properties for `make`, `model`, `year`, and `color`. To do this, you would write the following function:

```
function car(make, model, year) {  
    this.make = make  
    this.model = model  
    this.year = year  
}
```

Notice the use of `this` to assign values to the object’s properties based on the values passed to the function.

Now you can create an object called `mycar` as follows:

```
mycar = new car("Eagle", "Talon TSi", 1993)
```

This statement creates `mycar` and assigns it the specified values for its properties. Then the value of `mycar.make` is the string “Eagle”, `mycar.year` is the integer 1993, and so on.

You can create any number of `car` objects by calls to `new`. For example,

```
kenscar = new car("Nissan", "300ZX", 1992)  
vpgscar = new car("Mazda", "Miata", 1990)
```

An object can have a property that is itself another object. For example, suppose you define an object called `person` as follows:

```
function person(name, age, sex) {
    this.name = name
    this.age = age
    this.sex = sex
}
```

and then instantiate two new `person` objects as follows:

```
rand = new person("Rand McKinnon", 33, "M")
ken = new person("Ken Jones", 39, "M")
```

Then you can rewrite the definition of `car` to include an `owner` property that takes a `person` object, as follows:

```
function car(make, model, year, owner) {
    this.make = make
    this.model = model
    this.year = year
    this.owner = owner
}
```

To instantiate the new objects, you then use the following:

```
car1 = new car("Eagle", "Talon TSi", 1993, rand)
car2 = new car("Nissan", "300ZX", 1992, ken)
```

Notice that instead of passing a literal string or integer value when creating the new objects, the above statements pass the objects `rand` and `ken` as the arguments for the owners. Then if you want to find out the name of the owner of `car2`, you can access the following property:

```
car2.owner.name
```

Note that you can always add a property to a previously defined object. For example, the statement

```
car1.color = "black"
```

adds a property `color` to `car1`, and assigns it a value of “black.” However, this does not affect any other objects. To add the new property to all objects of the same type, you have to add the property to the definition of the `car` object type.

## Indexing Object Properties

In JavaScript 1.0, you can refer to an object's properties by their property name or by their ordinal index. In JavaScript 1.1 or later, however, if you initially define a property by its name, you must always refer to it by its name, and if you initially define a property by an index, you must always refer to it by its index.

This applies when you create an object and its properties with a constructor function, as in the above example of the `Car` object type, and when you define individual properties explicitly (for example, `myCar.color = "red"`). So if you define object properties initially with an index, such as `myCar[5] = "25 mpg"`, you can subsequently refer to the property as `myCar[5]`.

The exception to this rule is objects reflected from HTML, such as the `forms` array. You can always refer to objects in these arrays by either their ordinal number (based on where they appear in the document) or their name (if defined). For example, if the second `<FORM>` tag in a document has a `NAME` attribute of "myForm", you can refer to the form as `document.forms[1]` or `document.forms["myForm"]` or `document.myForm`.

## Defining Properties for an Object Type

You can add a property to a previously defined object type by using the `prototype` property. This defines a property that is shared by all objects of the specified type, rather than by just one instance of the object. The following code adds a `color` property to all objects of type `car`, and then assigns a value to the `color` property of the object `car1`.

```
Car.prototype.color=null
car1.color="black"
```

See the `prototype` property of the `Function` object in the *Server-Side JavaScript Reference* for more information.



## Defining Methods

A *method* is a function associated with an object. You define a method the same way you define a standard function. Then you use the following syntax to associate the function with an existing object:

```
object.methodname = function_name
```

where `object` is an existing object, `methodname` is the name you are assigning to the method, and `function_name` is the name of the function.

You can then call the method in the context of the object as follows:

```
object.methodname(params);
```

You can define methods for an object type by including a method definition in the object constructor function. For example, you could define a function that would format and display the properties of the previously-defined `car` objects; for example,

```
function displayCar() {
    var result = "A Beautiful " + this.year + " " + this.make
        + " " + this.model
    pretty_print(result)
}
```

where `pretty_print` is function to display a horizontal rule and a string. Notice the use of `this` to refer to the object to which the method belongs.

You can make this function a method of `car` by adding the statement

```
this.displayCar = displayCar;
```

to the object definition. So, the full definition of `car` would now look like

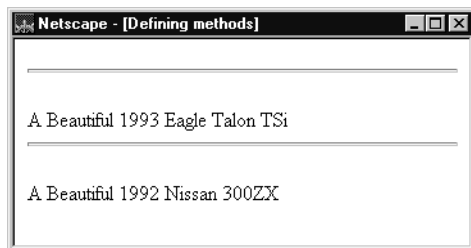
```
function car(make, model, year, owner) {
    this.make = make
    this.model = model
    this.year = year
    this.owner = owner
    this.displayCar = displayCar
}
```

Then you can call the `displayCar` method for each of the objects as follows:

```
car1.displayCar()
car2.displayCar()
```

This produces the output shown in the following figure.

Figure 9.1 Displaying method output



## Using this for Object References

JavaScript has a special keyword, `this`, that you can use within a method to refer to the current object. For example, suppose you have a function called `validate` that validates an object's `value` property, given the object and the high and low values:

```
function validate(obj, lowval, hival) {
    if ((obj.value < lowval) || (obj.value > hival))
        alert("Invalid Value!")
}
```

Then, you could call `validate` in each form element's `onChange` event handler, using `this` to pass it the form element, as in the following example:

```
<INPUT TYPE="text" NAME="age" SIZE=3
    onChange="validate(this, 18, 99)">
```

In general, `this` refers to the calling object in a method.

When combined with the `form` property, `this` can refer to the current object's parent form. In the following example, the form `myForm` contains a `Text` object and a button. When the user clicks the button, the value of the `Text` object is set to the form's name. The button's `onClick` event handler uses `this.form` to refer to the parent form, `myForm`.

```
<FORM NAME="myForm">
Form name:<INPUT TYPE="text" NAME="text1" VALUE="Beluga">
<P>
<INPUT NAME="button1" TYPE="button" VALUE="Show Form Name"
    onClick="this.form.text1.value=this.form.name">
</FORM>
```

## Deleting Objects

You can remove an object by using the `delete` operator. The following code shows how to remove an object.

```
myobj=new Number()
delete myobj    // removes the object and returns true
```

See “delete” on page 98 for more information.

**JavaScript 1.1.** You can remove an object by setting its object reference to null (if that is the last reference to the object). JavaScript finalizes the object immediately, as part of the assignment expression.

**JavaScript 1.0.** You cannot remove objects—they exist until you leave the page containing the object.

## Predefined Core Objects

This section describes the predefined objects in core JavaScript: `Array`, `Boolean`, `Date`, `Function`, `Math`, `Number`, `RegExp`, and `String`. The predefined server-side objects are described in Chapter 13, “Session Management Service.”

### Array Object

JavaScript does not have an explicit array data type. However, you can use the predefined `Array` object and its methods to work with arrays in your applications. The `Array` object has methods for manipulating arrays in various ways, such as joining, reversing, and sorting them. It has a property for determining the array length and other properties for use with regular expressions.

An *array* is an ordered set of values that you refer to with a name and an index. For example, you could have an array called `emp` that contains employees’ names indexed by their employee number. So `emp[1]` would be employee number one, `emp[2]` employee number two, and so on.

## Creating an Array

To create an `Array` object:

```
1. arrayObjectName = new Array(element0, element1, ..., elementN)
2. arrayObjectName = new Array(arrayLength)
```

`arrayObjectName` is either the name of a new object or a property of an existing object. When using `Array` properties and methods, `arrayObjectName` is either the name of an existing `Array` object or a property of an existing object.

`element0, element1, ..., elementN` is a list of values for the array's elements. When this form is specified, the array is initialized with the specified values as its elements, and the array's `length` property is set to the number of arguments.

`arrayLength` is the initial length of the array. The following code creates an array of five elements:

```
billingMethod = new Array(5)
```

`Array` literals are also `Array` objects; for example, the following literal is an `Array` object. See “[Array Literals](#)” on page 81 for details on array literals.

```
coffees = ["French Roast", "Columbian", "Kona"]
```

## Populating an Array

You can populate an array by assigning values to its elements. For example,

```
emp[1] = "Casey Jones"
emp[2] = "Phil Lesh"
emp[3] = "August West"
```

You can also populate an array when you create it:

```
myArray = new Array("Hello", myVar, 3.14159)
```

## Referring to Array Elements

You refer to an array's elements by using the element's ordinal number. For example, suppose you define the following array:

```
myArray = new Array("Wind", "Rain", "Fire")
```

You then refer to the first element of the array as `myArray[0]` and the second element of the array as `myArray[1]`.

The index of the elements begins with zero (0), but the length of array (for example, `myArray.length`) reflects the number of elements in the array.

## Array Methods

The `Array` object has the following methods:

- `concat` joins two arrays and returns a new array.
- `join` joins all elements of an array into a string.
- `pop` removes the last element from an array and returns that element.
- `push` adds one or more elements to the end of an array and returns that last element added.
- `reverse` transposes the elements of an array: the first array element becomes the last and the last becomes the first.
- `shift` removes the first element from an array and returns that element
- `slice` extracts a section of an array and returns a new array.
- `splice` adds and/or removes elements from an array.
- `sort` sorts the elements of an array.
- `unshift` adds one or more elements to the front of an array and returns the new length of the array.

For example, suppose you define the following array:

```
myArray = new Array("Wind", "Rain", "Fire")
```

`myArray.join()` returns “Wind,Rain,Fire”; `myArray.reverse` transposes the array so that `myArray[0]` is “Fire”, `myArray[1]` is “Rain”, and `myArray[2]` is “Wind”. `myArray.sort` sorts the array so that `myArray[0]` is “Fire”, `myArray[1]` is “Rain”, and `myArray[2]` is “Wind”.

## Two-Dimensional Arrays

The following code creates a two-dimensional array.

```
a = new Array(4)
for (i=0; i < 4; i++) {
    a[i] = new Array(4)
    for (j=0; j < 4; j++) {
        a[i][j] = "[" + i + ", " + j + "]"
    }
}
```

The following code displays the array:

```
for (i=0; i < 4; i++) {
    str = "Row " + i + ": "
    for (j=0; j < 4; j++) {
        str += a[i][j]
    }
    document.write(str, "<p>")
}
```

This example displays the following results:

```
Row 0:[0,0][0,1][0,2][0,3]
Row 1:[1,0][1,1][1,2][1,3]
Row 2:[2,0][2,1][2,2][2,3]
Row 3:[3,0][3,1][3,2][3,3]
```

In server-side JavaScript, you can display the same output by calling the `write` function instead of using `document.write`.

## Arrays and Regular Expressions

When an array is the result of a match between a regular expression and a string, the array returns properties and elements that provide information about the match. An array is the return value of `regexp.exec`, `string.match`, and `string.replace`. For information on using arrays with regular expressions, see Chapter 6, “Regular Expressions.”

## Boolean Object

The `Boolean` object is a wrapper around the primitive `Boolean` data type. Use the following syntax to create a `Boolean` object:

```
booleanObjectName = new Boolean(value)
```

## Date Object

JavaScript does not have a date data type. However, you can use the `Date` object and its methods to work with dates and times in your applications. The `Date` object has a large number of methods for setting, getting, and manipulating dates. It does not have any properties.

JavaScript handles dates similarly to Java. The two languages have many of the same date methods, and both languages store dates as the number of milliseconds since January 1, 1970, 00:00:00.

**Note** Dates prior to 1970 are not allowed.

To create a `Date` object:

```
dateObjectName = new Date([parameters])
```

where `dateObjectName` is the name of the `Date` object being created; it can be a new object or a property of an existing object.

The parameters in the preceding syntax can be any of the following:

- Nothing: creates today's date and time. For example, `today = new Date()`.
- A string representing a date in the following form: "Month day, year hours:minutes:seconds." For example, `Xmas95 = new Date("December 25, 1995 13:30:00")`. If you omit hours, minutes, or seconds, the value will be set to zero.
- A set of integer values for year, month, and day. For example, `Xmas95 = new Date(1995,11,25)`. A set of values for year, month, day, hour, minute, and seconds. For example, `Xmas95 = new Date(1995,11,25,9,30,0)`.

## Methods of the Date Object

The `Date` object methods for handling dates and times fall into these broad categories:

- "set" methods, for setting date and time values in `Date` objects.
- "get" methods, for getting date and time values from `Date` objects.
- "to" methods, for returning string values from `Date` objects.
- parse and UTC methods, for parsing `Date` strings.



With the “get” and “set” methods you can get and set seconds, minutes, hours, day of the month, day of the week, months, and years separately. There is a `getDay` method that returns the day of the week, but no corresponding `setDay` method, because the day of the week is set automatically. These methods use integers to represent these values as follows:

- Seconds and minutes: 0 to 59
- Hours: 0 to 23
- Day: 0 (Sunday) to 6 (Saturday)
- Date: 1 to 31 (day of the month)
- Months: 0 (January) to 11 (December)
- Year: years since 1900

For example, suppose you define the following date:

```
Xmas95 = new Date("December 25, 1995")
```

Then `Xmas95.getMonth()` returns 11, and `Xmas95.getFullYear()` returns 95.

The `getTime` and `setTime` methods are useful for comparing dates. The `getTime` method returns the number of milliseconds since January 1, 1970, 00:00:00 for a `Date` object.

For example, the following code displays the number of days left in the current year:

```
today = new Date()
endYear = new Date(1995,11,31,23,59,59) // Set day and month
endYear.setYear(today.getYear()) // Set year to this year
msPerDay = 24 * 60 * 60 * 1000 // Number of milliseconds per day
daysLeft = (endYear.getTime() - today.getTime()) / msPerDay
daysLeft = Math.round(daysLeft) //returns days left in the year
```

This example creates a `Date` object named `today` that contains today's date. It then creates a `Date` object named `endYear` and sets the year to the current year. Then, using the number of milliseconds per day, it computes the number of days between `today` and `endYear`, using `getTime` and rounding to a whole number of days.

The `parse` method is useful for assigning values from date strings to existing `Date` objects. For example, the following code uses `parse` and `setTime` to assign a date value to the `IPOdate` object:

```
IPOdate = new Date()
IPOdate.setTime(Date.parse("Aug 9, 1995"))
```

## Using the Date Object: an Example

In the following example, the function `JSClock()` returns the time in the format of a digital clock.

```
function JSClock() {
    var time = new Date()
    var hour = time.getHours()
    var minute = time.getMinutes()
    var second = time.getSeconds()
    var temp = "" + ((hour > 12) ? hour - 12 : hour)
    temp += ((minute < 10) ? ":0" : ":") + minute
    temp += ((second < 10) ? ":0" : ":") + second
    temp += (hour >= 12) ? " P.M." : " A.M."
    return temp
}
```

The `JSClock` function first creates a new `Date` object called `time`; since no arguments are given, `time` is created with the current date and time. Then calls to the `getHours`, `getMinutes`, and `getSeconds` methods assign the value of the current hour, minute and seconds to `hour`, `minute`, and `second`.

The next four statements build a string value based on the time. The first statement creates a variable `temp`, assigning it a value using a conditional expression; if `hour` is greater than 12, (`hour - 13`), otherwise simply `hour`.

The next statement appends a minute value to `temp`. If the value of `minute` is less than 10, the conditional expression adds a string with a preceding zero; otherwise it adds a string with a demarcating colon. Then a statement appends a seconds value to `temp` in the same way.

Finally, a conditional expression appends "PM" to `temp` if `hour` is 12 or greater; otherwise, it appends "AM" to `temp`.

# Function Object

The predefined `Function` object specifies a string of JavaScript code to be compiled as a function.

To create a `Function` object:

```
functionObjectName = new Function ([arg1, arg2, ... argn], functionBody)
```

`functionObjectName` is the name of a variable or a property of an existing object. It can also be an object followed by a lowercase event handler name, such as `window.onerror`.

`arg1, arg2, ... argn` are arguments to be used by the function as formal argument names. Each must be a string that corresponds to a valid JavaScript identifier; for example "x" or "theForm".

`functionBody` is a string specifying the JavaScript code to be compiled as the function body.

`Function` objects are evaluated each time they are used. This is less efficient than declaring a function and calling it within your code, because declared functions are compiled.

In addition to defining functions as described here, you can also use the `function` statement. See the *Server-Side JavaScript Reference* for more information.

The following code assigns a function to the variable `setBGColor`. This function sets the current document's background color.

```
var setBGColor = new Function("document.bgColor='antiquewhite'")
```

To call the `Function` object, you can specify the variable name as if it were a function. The following code executes the function specified by the `setBGColor` variable:

```
var colorChoice="antiquewhite"
if (colorChoice=="antiquewhite") {setBGColor()}
```

You can assign the function to an event handler in either of the following ways:

1. `document.form1.colorButton.onclick=setBGColor`
2. `<INPUT NAME="colorButton" TYPE="button" VALUE="Change background color" onClick="setBGColor()">`

Creating the variable `setBGColor` shown above is similar to declaring the following function:

```
function setBGColor() {  
    document.bgColor='antiquewhite'  
}
```

You can nest a function within a function. The nested (inner) function is private to its containing (outer) function:

- The inner function can be accessed only from statements in the outer function.
- The inner function can use the arguments and variables of the outer function. The outer function cannot use the arguments and variables of the inner function.

## Math Object

The predefined `Math` object has properties and methods for mathematical constants and functions. For example, the `Math` object's `PI` property has the value of pi (3.141...), which you would use in an application as

```
Math.PI
```

Similarly, standard mathematical functions are methods of `Math`. These include trigonometric, logarithmic, exponential, and other functions. For example, if you want to use the trigonometric function sine, you would write

```
Math.sin(1.56)
```

Note that all trigonometric methods of `Math` take arguments in radians.

The following table summarizes the `Math` object's methods.

Table 9.1 Methods of `Math`

Method	Description
<code>abs</code>	Absolute value
<code>sin</code> , <code>cos</code> , <code>tan</code>	Standard trigonometric functions; argument in radians
<code>acos</code> , <code>asin</code> , <code>atan</code>	Inverse trigonometric functions; return values in radians
<code>exp</code> , <code>log</code>	Exponential and natural logarithm, base <i>e</i>
<code>ceil</code>	Returns least integer greater than or equal to argument
<code>floor</code>	Returns greatest integer less than or equal to argument
<code>min</code> , <code>max</code>	Returns greater or lesser (respectively) of two arguments
<code>pow</code>	Exponential; first argument is base, second is exponent
<code>round</code>	Rounds argument to nearest integer
<code>sqrt</code>	Square root

Unlike many other objects, you never create a `Math` object of your own. You always use the predefined `Math` object.

It is often convenient to use the `with` statement when a section of code uses several math constants and methods, so you don't have to type “`Math`” repeatedly. For example,

```
with (Math) {
  a = PI * r*r
  y = r*sin(theta)
  x = r*cos(theta)
}
```

# Number Object

The `Number` object has properties for numerical constants, such as maximum value, not-a-number, and infinity. You cannot change the values of these properties and you use them as follows:

```
biggestNum = Number.MAX_VALUE
smallestNum = Number.MIN_VALUE
infiniteNum = Number.POSITIVE_INFINITY
negInfiniteNum = Number.NEGATIVE_INFINITY
notANum = Number.NaN
```

You always refer to a property of the predefined `Number` object as shown above, and not as a property of a `Number` object you create yourself.

The following table summarizes the `Number` object’s properties.

Table 9.2 Properties of Number

Method	Description
<code>MAX_VALUE</code>	The largest representable number
<code>MIN_VALUE</code>	The smallest representable number
<code>NaN</code>	Special “not a number” value
<code>NEGATIVE_INFINITY</code>	Special infinite value; returned on overflow
<code>POSITIVE_INFINITY</code>	Special negative infinite value; returned on overflow

# RegExp Object

The `RegExp` object lets you work with regular expressions. It is described in Chapter 6, “Regular Expressions.”

## String Object

The `String` object is a wrapper around the string primitive data type. Do not confuse a string literal with the `String` object. For example, the following code creates the string literal `s1` and also the `String` object `s2`:

```
s1 = "foo" //creates a string literal value
s2 = new String("foo") //creates a String object
```

You can call any of the methods of the `String` object on a string literal value—JavaScript automatically converts the string literal to a temporary `String` object, calls the method, then discards the temporary `String` object. You can also use the `String.length` property with a string literal.

You should use string literals unless you specifically need to use a `String` object, because `String` objects can have counterintuitive behavior. For example:

```
s1 = "2 + 2" //creates a string literal value
s2 = new String("2 + 2") //creates a String object
eval(s1) //returns the number 4
eval(s2) //returns the string "2 + 2"
```

A `String` object has one property, `length`, that indicates the number of characters in the string. For example, the following code assigns `x` the value 13, because “Hello, World!” has 13 characters:

```
myString = "Hello, World!"
x = mystring.length
```

A `String` object has two types of methods: those that return a variation on the string itself, such as `substring` and `toUpperCase`, and those that return an HTML-formatted version of the string, such as `bold` and `link`.

For example, using the previous example, both `mystring.toUpperCase()` and `"hello, world!".toUpperCase()` return the string “HELLO, WORLD!”.

The `substring` method takes two arguments and returns a subset of the string between the two arguments. Using the previous example, `mystring.substring(4, 9)` returns the string “o, Wo.” See the `substring` method of the `String` object in the *Server-Side JavaScript Reference* for more information.

The `String` object also has a number of methods for automatic HTML formatting, such as `bold` to create boldface text and `link` to create a hyperlink. For example, you could create a hyperlink to a hypothetical URL with the `link` method as follows:

```
mystring.link( "http://www.helloworld.com" )
```

The following table summarizes the methods of `String` objects.

Table 9.3 Methods of String

Method	Description
<code>anchor</code>	Creates HTML named anchor
<code>big</code> , <code>blink</code> , <code>bold</code> , <code>fixed</code> , <code>italics</code> , <code>small</code> , <code>strike</code> , <code>sub</code> , <code>sup</code>	Creates HTML formatted string
<code>charAt</code> , <code>charCodeAt</code>	Returns the character or character code at the specified position in string
<code>indexOf</code> , <code>lastIndexOf</code>	Returns the position of specified substring in the string or last position of specified substring, respectively
<code>link</code>	Creates HTML hyperlink
<code>concat</code>	Combines the text of two strings and returns a new string
<code>fromCharCode</code>	Constructs a string from the specified sequence of ISO-Latin-1 codeset values
<code>split</code>	Splits a <code>String</code> object into an array of strings by separating the string into substrings
<code>slice</code>	Extracts a section of an string and returns a new string.
<code>substring</code> , <code>substr</code>	Returns the specified subset of the string, either by specifying the start and end indexes or the start index and a length
<code>match</code> , <code>replace</code> , <code>search</code>	Used to work with regular expressions
<code>toLowerCase</code> , <code>toUpperCase</code>	Returns the string in all lowercase or all uppercase, respectively



# Details of the Object Model

JavaScript is an object-based language based on prototypes, rather than being class-based. Because of this different basis, it can be less apparent how JavaScript allows you to create hierarchies of objects and to have inheritance of properties and their values. This chapter attempts to clarify the situation.

This chapter assumes that you are already somewhat familiar with JavaScript and that you have used JavaScript functions to create simple objects.

This chapter contains the following sections:

- Class-Based vs. Prototype-Based Languages
- The Employee Example
- Creating the Hierarchy
- Object Properties
- More Flexible Constructors
- Property Inheritance Revisited

# Class-Based vs. Prototype-Based Languages

Class-based object-oriented languages, such as Java and C++, are founded on the concept of two distinct entities: classes and instances.

- A *class* defines all of the properties (considering methods and fields in Java, or members in C++, to be properties) that characterize a certain set of objects. A class is an abstract thing, rather than any particular member of the set of objects it describes. For example, the `Employee` class could represent the set of all employees.
- An *instance*, on the other hand, is the instantiation of a class; that is, one of its members. For example, `Victoria` could be an instance of the `Employee` class, representing a particular individual as an employee. An instance has exactly the properties of its parent class (no more, no less).

A prototype-based language, such as JavaScript, does not make this distinction: it simply has objects. A prototype-based language has the notion of a *prototypical object*, an object used as a template from which to get the initial properties for a new object. Any object can specify its own properties, either when you create it or at run time. In addition, any object can be associated as the *prototype* for another object, allowing the second object to share the first object's properties.

## Defining a Class

In class-based languages, you define a class in a separate *class definition*. In that definition you can specify special methods, called *constructors*, to create instances of the class. A constructor method can specify initial values for the instance's properties and perform other processing appropriate at creation time. You use the `new` operator in association with the constructor method to create class instances.

JavaScript follows a similar model, but does not have a class definition separate from the constructor. Instead, you define a constructor function to create objects with a particular initial set of properties and values. Any JavaScript function can be used as a constructor. You use the `new` operator with a constructor function to create a new object.

## Subclasses and Inheritance

In a class-based language, you create a hierarchy of classes through the class definitions. In a class definition, you can specify that the new class is a *subclass* of an already existing class. The subclass inherits all the properties of the superclass and additionally can add new properties or modify the inherited ones. For example, assume the `Employee` class includes only the `name` and `dept` properties, and `Manager` is a subclass of `Employee` that adds the `reports` property. In this case, an instance of the `Manager` class would have all three properties: `name`, `dept`, and `reports`.

JavaScript implements inheritance by allowing you to associate a prototypical object with any constructor function. So, you can create exactly the `Employee-Manager` example, but you use slightly different terminology. First you define the `Employee` constructor function, specifying the `name` and `dept` properties. Next, you define the `Manager` constructor function, specifying the `reports` property. Finally, you assign a new `Employee` object as the `prototype` for the `Manager` constructor function. Then, when you create a new `Manager`, it inherits the `name` and `dept` properties from the `Employee` object.

## Adding and Removing Properties

In class-based languages, you typically create a class at compile time and then you instantiate instances of the class either at compile time or at run time. You cannot change the number or the type of properties of a class after you define the class. In JavaScript, however, at run time you can add or remove properties from any object. If you add a property to an object that is used as the `prototype` for a set of objects, the objects for which it is the `prototype` also get the new property.

## Summary of Differences

The following table gives a short summary of some of these differences. The rest of this chapter describes the details of using JavaScript constructors and prototypes to create an object hierarchy and compares this to how you would do it in Java.

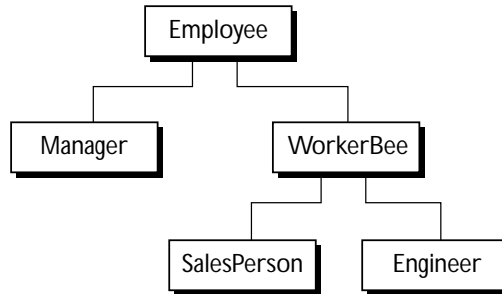
Table 10.1 Comparison of class-based (Java) and prototype-based (JavaScript) object systems

Class-based (Java)	Prototype-based (JavaScript)
Class and instance are distinct entities.	All objects are instances.
Define a class with a class definition; instantiate a class with constructor methods.	Define and create a set of objects with constructor functions.
Create a single object with the new operator.	Same.
Construct an object hierarchy by using class definitions to define subclasses of existing classes.	Construct an object hierarchy by assigning an object as the prototype associated with a constructor function.
Inherit properties by following the class chain.	Inherit properties by following the prototype chain.
Class definition specifies <i>all</i> properties of all instances of a class. Cannot add properties dynamically at run time.	Constructor function or prototype specifies an <i>initial set</i> of properties. Can add or remove properties dynamically to individual objects or to the entire set of objects.

# The Employee Example

The remainder of this chapter uses the employee hierarchy shown in the following figure.

Figure 10.1A simple object hierarchy



This example uses the following objects:

- `Employee` has the properties `name` (whose value defaults to the empty string) and `dept` (whose value defaults to “general”).
- `Manager` is based on `Employee`. It adds the `reports` property (whose value defaults to an empty array, intended to have an array of `Employee` objects as its value).
- `WorkerBee` is also based on `Employee`. It adds the `projects` property (whose value defaults to an empty array, intended to have an array of strings as its value).
- `SalesPerson` is based on `WorkerBee`. It adds the `quota` property (whose value defaults to 100). It also overrides the `dept` property with the value “sales”, indicating that all salespersons are in the same department.
- `Engineer` is based on `WorkerBee`. It adds the `machine` property (whose value defaults to the empty string) and also overrides the `dept` property with the value “engineering”.

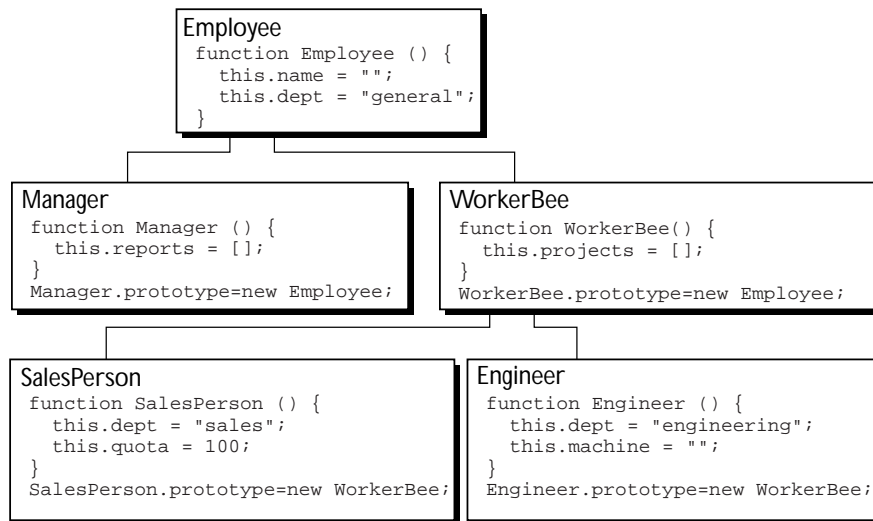
# Creating the Hierarchy

There are several ways to define appropriate constructor functions to implement the Employee hierarchy. How you choose to define them depends largely on what you want to be able to do in your application.

This section shows how to use very simple (and comparatively inflexible) definitions to demonstrate how to get the inheritance to work. In these definitions, you cannot specify any property values when you create an object. The newly-created object simply gets the default values, which you can change at a later time. Figure 10.2 illustrates the hierarchy with these simple definitions.

In a real application, you would probably define constructors that allow you to provide property values at object creation time (see “More Flexible Constructors” on page 173 for information). For now, these simple definitions demonstrate how the inheritance occurs.

Figure 10.2 The Employee object definitions



The following Java and JavaScript `Employee` definitions are similar. The only differences are that you need to specify the type for each property in Java but not in JavaScript, and you need to create an explicit constructor method for the Java class.

JavaScript	Java
<pre>function Employee () {     this.name = "";     this.dept = "general"; }</pre>	<pre>public class Employee {     public String name;     public String dept;     public Employee () {         this.name = "";         this.dept = "general";     } }</pre>

The `Manager` and `WorkerBee` definitions show the difference in how to specify the next object higher in the inheritance chain. In JavaScript, you add a prototypical instance as the value of the `prototype` property of the constructor function. You can do so at any time after you define the constructor. In Java, you specify the superclass within the class definition. You cannot change the superclass outside the class definition.

JavaScript	Java
<pre>function Manager () {     this.reports = []; } Manager.prototype = new Employee;  function WorkerBee () {     this.projects = []; } WorkerBee.prototype = new Employee;</pre>	<pre>public class Manager extends Employee {     public Employee[] reports;     public Manager () {         this.reports = new Employee[0];     } }  public class WorkerBee extends Employee {     public String[] projects;     public WorkerBee () {         this.projects = new String[0];     } }</pre>

The `Engineer` and `SalesPerson` definitions create objects that descend from `WorkerBee` and hence from `Employee`. An object of these types has properties of all the objects above it in the chain. In addition, these definitions override the inherited value of the `dept` property with new values specific to these objects.

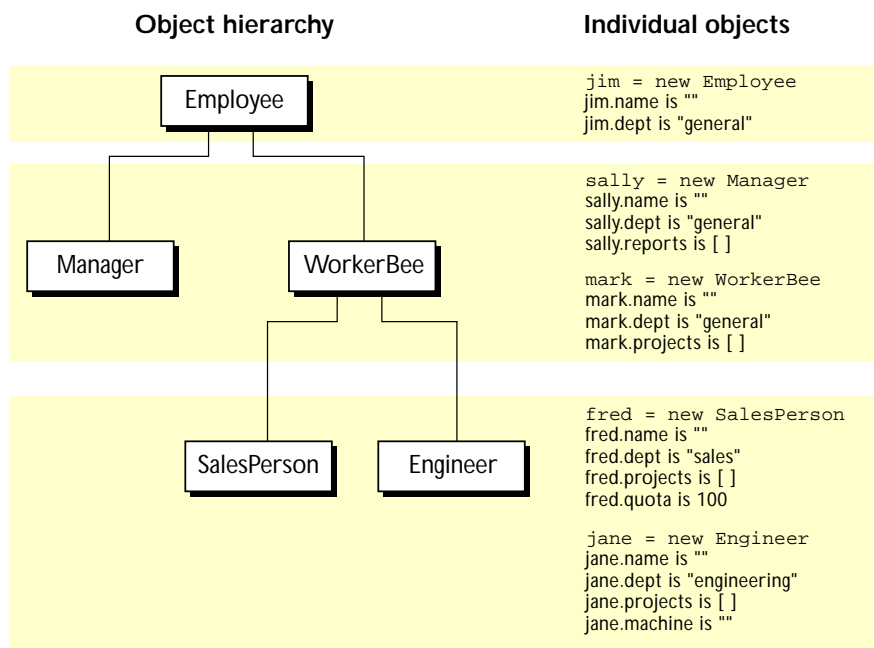
JavaScript	Java
<pre>function SalesPerson () {     this.dept = "sales";     this.quota = 100; } SalesPerson.prototype = new WorkerBee;  function Engineer () {     this.dept = "engineering";     this.machine = ""; } Engineer.prototype = new WorkerBee;</pre>	<pre>public class SalesPerson extends WorkerBee {     public double quota;     public SalesPerson () {         this.dept = "sales";         this.quota = 100.0;     } }  public class Engineer extends WorkerBee {     public String machine;     public Engineer () {         this.dept = "engineering";         this.machine = "";     } }</pre>

Using these definitions, you can create instances of these objects that get the default values for their properties. Figure 10.3 illustrates using these JavaScript definitions to create new objects and shows the property values for the new objects.

**Note** The term *instance* has a specific technical meaning in class-based languages. In these languages, an instance is an individual member of a class and is fundamentally different from a class. In JavaScript, “instance” does not have this technical meaning because JavaScript does not have this difference between classes and instances. However, in talking about JavaScript, “instance” can be used informally to mean an object created using a particular constructor function. So, in this example, you could informally say that `jane` is an instance of `Engineer`. Similarly, although the terms *parent*, *child*, *ancestor*, and *descendant* do not have formal meanings in JavaScript; you can use them informally to refer to objects higher or lower in the prototype chain.



Figure 10.3 Creating objects with simple definitions



## Object Properties

This section discusses how objects inherit properties from other objects in the prototype chain and what happens when you add a property at run time.

## Inheriting Properties

Suppose you create the `mark` object as a `WorkerBee` as shown in Figure 10.3 with the following statement:

```
mark = new WorkerBee;
```

When JavaScript sees the `new` operator, it creates a new generic object and passes this new object as the value of the `this` keyword to the `WorkerBee` constructor function. The constructor function explicitly sets the value of the `projects` property. It also sets the value of the internal `__proto__` property to

the value of `WorkerBee.prototype`. (That property name has two underscore characters at the front and two at the end.) The `__proto__` property determines the prototype chain used to return property values. Once these properties are set, JavaScript returns the new object and the assignment statement sets the variable `mark` to that object.

This process does not explicitly put values in the `mark` object (*local* values) for the properties `mark` inherits from the prototype chain. When you ask for the value of a property, JavaScript first checks to see if the value exists in that object. If it does, that value is returned. If the value is not there locally, JavaScript checks the prototype chain (using the `__proto__` property). If an object in the prototype chain has a value for the property, that value is returned. If no such property is found, JavaScript says the object does not have the property. In this way, the `mark` object has the following properties and values:

```
mark.name = "";
mark.dept = "general";
mark.projects = [];
```

The `mark` object inherits values for the `name` and `dept` properties from the prototypical object in `mark.__proto__`. It is assigned a local value for the `projects` property by the `WorkerBee` constructor. This gives you inheritance of properties and their values in JavaScript. Some subtleties of this process are discussed in “Property Inheritance Revisited” on page 178.

Because these constructors do not let you supply instance-specific values, this information is generic. The property values are the default ones shared by all new objects created from `WorkerBee`. You can, of course, change the values of any of these properties. So, you could give specific information for `mark` as follows:

```
mark.name = "Doe, Mark";
mark.dept = "admin";
mark.projects = ["navigator"];
```

## Adding Properties

In JavaScript, you can add properties to any object at run time. You are not constrained to use only the properties provided by the constructor function. To add a property that is specific to a single object, you assign a value to the object, as follows:

```
mark.bonus = 3000;
```

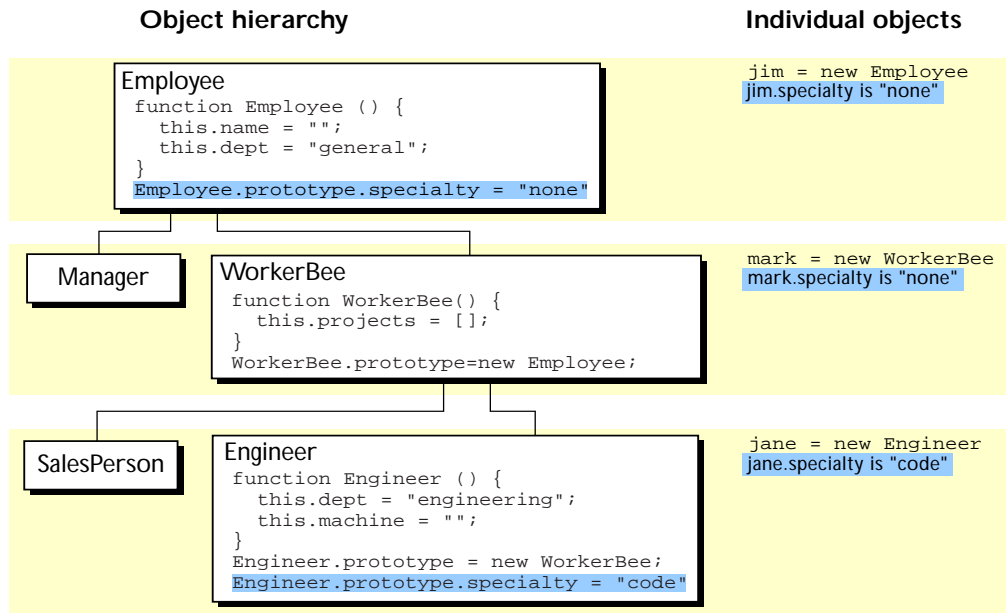
Now, the `mark` object has a `bonus` property, but no other `WorkerBee` has this property.

If you add a new property to an object that is being used as the prototype for a constructor function, you add that property to all objects that inherit properties from the prototype. For example, you can add a `specialty` property to all employees with the following statement:

```
Employee.prototype.specialty = "none";
```

As soon as JavaScript executes this statement, the `mark` object also has the `specialty` property with the value of `"none"`. The following figure shows the effect of adding this property to the `Employee` prototype and then overriding it for the `Engineer` prototype.

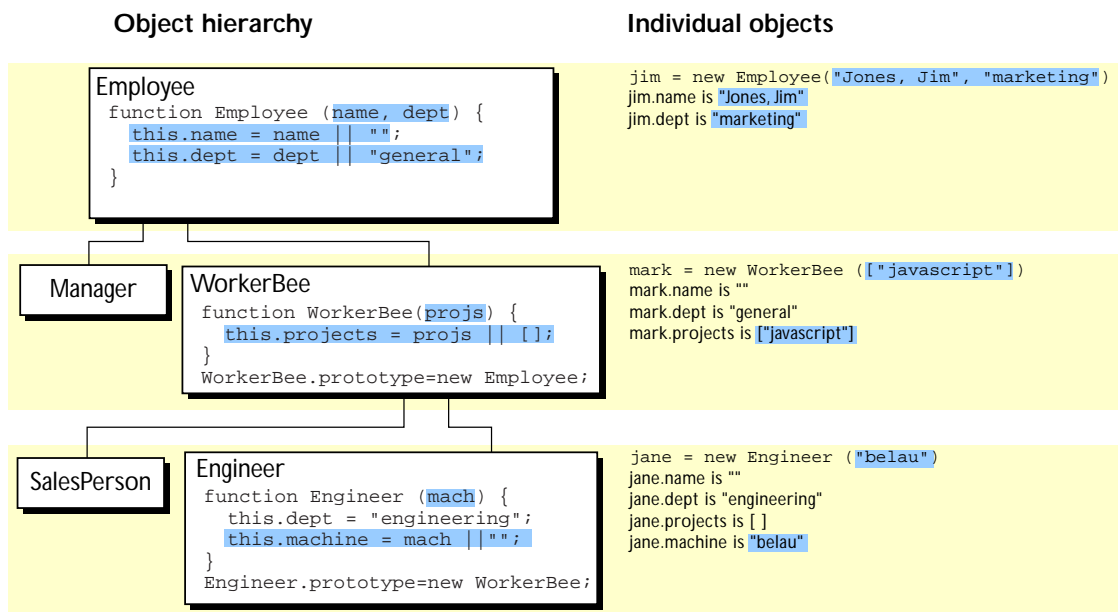
Figure 10.4 Adding properties



# More Flexible Constructors

The constructor functions shown so far do not let you specify property values when you create an instance. As with Java, you can provide arguments to constructors to initialize property values for instances. The following figure shows one way to do this.

Figure 10.5 Specifying properties in a constructor, take 1



The following table shows the Java and JavaScript definitions for these objects.

JavaScript	Java
<pre>function Employee (name, dept) {     this.name = name    "";     this.dept = dept    "general"; }</pre>	<pre>public class Employee {     public String name;     public String dept;     public Employee () {         this("", "general");     }     public Employee (name) {         this(name, "general");     }     public Employee (name, dept) {         this.name = name;         this.dept = dept;     } }</pre>
<pre>function WorkerBee (projs) {     this.projects = projs    []; } WorkerBee.prototype = new Employee;</pre>	<pre>public class WorkerBee extends Employee {     public String[] projects;     public WorkerBee () {         this(new String[0]);     }     public WorkerBee (String[] projs) {         this.projects = projs;     } }</pre>
<pre>function Engineer (mach) {     this.dept = "engineering";     this.machine = mach    ""; } Engineer.prototype = new WorkerBee;</pre>	<pre>public class Engineer extends WorkerBee {     public String machine;     public WorkerBee () {         this.dept = "engineering";         this.machine = "";     }     public WorkerBee (mach) {         this.dept = "engineering";         this.machine = mach;     } }</pre>

These JavaScript definitions use a special idiom for setting default values:

```
this.name = name || "";
```

The JavaScript logical OR operator (`| |`) evaluates its first argument. If that argument converts to true, the operator returns it. Otherwise, the operator returns the value of the second argument. Therefore, this line of code tests to

see if `name` has a useful value for the `name` property. If it does, it sets `this.name` to that value. Otherwise, it sets `this.name` to the empty string. This chapter uses this idiom for brevity; however, it can be puzzling at first glance.

With these definitions, when you create an instance of an object, you can specify values for the locally defined properties. As shown in Figure 10.5, you can use the following statement to create a new `Engineer`:

```
jane = new Engineer("belau");
```

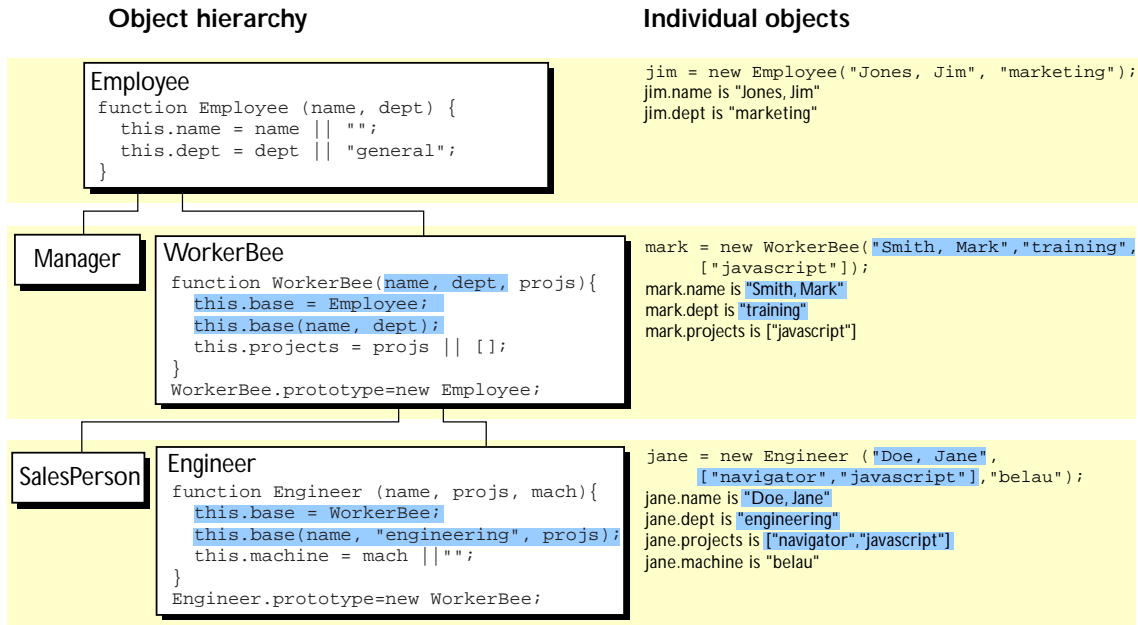
Jane's properties are now:

```
jane.name == "";  
jane.dept == "general";  
jane.projects == [];  
jane.machine == "belau"
```

Notice that with these definitions, you cannot specify an initial value for an inherited property such as `name`. If you want to specify an initial value for inherited properties in JavaScript, you need to add more code to the constructor function.

So far, the constructor function has created a generic object and then specified local properties and values for the new object. You can have the constructor add more properties by directly calling the constructor function for an object higher in the prototype chain. The following figure shows these new definitions.

Figure 10.6 Specifying properties in a constructor, take 2



Let's look at one of these definitions in detail. Here's the new definition for the Engineer constructor:

```
function Engineer (name, projs, mach) {
  this.base = WorkerBee;
  this.base(name, "engineering", projs);
  this.machine = mach || "";
}
```

Suppose you create a new Engineer object as follows:

```
jane = new Engineer("Doe, Jane", ["navigator", "javascript"], "belau");
```



JavaScript follows these steps:

1. The `new` operator creates a generic object and sets its `__proto__` property to `Engineer.prototype`.
2. The `new` operator passes the new object to the `Engineer` constructor as the value of the `this` keyword.
3. The constructor creates a new property called `base` for that object and assigns the value of the `WorkerBee` constructor to the `base` property. This makes the `WorkerBee` constructor a method of the `Engineer` object.

The name of the `base` property is not special. You can use any legal property name; `base` is simply evocative of its purpose.

4. The constructor calls the `base` method, passing as its arguments two of the arguments passed to the constructor (`"Doe, Jane"` and `["navigator", "javascript"]`) and also the string `"engineering"`. Explicitly using `"engineering"` in the constructor indicates that all `Engineer` objects have the same value for the inherited `dept` property, and this value overrides the value inherited from `Employee`.
5. Because `base` is a method of `Engineer`, within the call to `base`, JavaScript binds the `this` keyword to the object created in Step 1. Thus, the `WorkerBee` function in turn passes the `"Doe, Jane"` and `["navigator", "javascript"]` arguments to the `Employee` constructor function. Upon return from the `Employee` constructor function, the `WorkerBee` function uses the remaining argument to set the `projects` property.
6. Upon return from the `base` method, the `Engineer` constructor initializes the object's `machine` property to `"belau"`.
7. Upon return from the constructor, JavaScript assigns the new object to the `jane` variable.

You might think that, having called the `WorkerBee` constructor from inside the `Engineer` constructor, you have set up inheritance appropriately for `Engineer` objects. This is not the case. Calling the `WorkerBee` constructor ensures that an `Engineer` object starts out with the properties specified in all constructor

functions that are called. However, if you later add properties to the `Employee` or `WorkerBee` prototypes, those properties are not inherited by the `Engineer` object. For example, assume you have the following statements:

```
function Engineer (name, projs, mach) {
  this.base = WorkerBee;
  this.base(name, "engineering", projs);
  this.machine = mach || "";
}
jane = new Engineer("Doe, Jane", ["navigator", "javascript"], "belau");
Employee.prototype.specialty = "none";
```

The `jane` object does not inherit the `specialty` property. You still need to explicitly set up the prototype to ensure dynamic inheritance. Assume instead you have these statements:

```
function Engineer (name, projs, mach) {
  this.base = WorkerBee;
  this.base(name, "engineering", projs);
  this.machine = mach || "";
}
Engineer.prototype = new WorkerBee;
jane = new Engineer("Doe, Jane", ["navigator", "javascript"], "belau");
Employee.prototype.specialty = "none";
```

Now the value of the `jane` object's `specialty` property is "none".

## Property Inheritance Revisited

The preceding sections described how JavaScript constructors and prototypes provide hierarchies and inheritance. This section discusses some subtleties that were not necessarily apparent in the earlier discussions.

### Local versus Inherited Values

When you access an object property, JavaScript performs these steps, as described earlier in this chapter:

1. Check to see if the value exists locally. If it does, return that value.
2. If there is not a local value, check the prototype chain (using the `__proto__` property).

3. If an object in the prototype chain has a value for the specified property, return that value.
4. If no such property is found, the object does not have the property.

The outcome of these steps depends on how you define things along the way. The original example had these definitions:

```
function Employee () {
    this.name = "";
    this.dept = "general";
}

function WorkerBee () {
    this.projects = [];
}
WorkerBee.prototype = new Employee;
```

With these definitions, suppose you create *amy* as an instance of *WorkerBee* with the following statement:

```
amy = new WorkerBee;
```

The *amy* object has one local property, *projects*. The values for the *name* and *dept* properties are not local to *amy* and so are gotten from the *amy* object's *\_\_proto\_\_* property. Thus, *amy* has these property values:

```
amy.name == "";
amy.dept = "general";
amy.projects == [];
```

Now suppose you change the value of the *name* property in the prototype associated with *Employee*:

```
Employee.prototype.name = "Unknown"
```

At first glance, you might expect that new value to propagate down to all the instances of *Employee*. However, it does not.

When you create *any* instance of the *Employee* object, that instance gets a local value for the *name* property (the empty string). This means that when you set the *WorkerBee* prototype by creating a new *Employee* object, *WorkerBee.prototype* has a local value for the *name* property. Therefore, when JavaScript looks up the *name* property of the *amy* object (an instance of *WorkerBee*), JavaScript finds the local value for that property in *WorkerBee.prototype*. It therefore does not look farther up the chain to *Employee.prototype*.

If you want to change the value of an object property at run time and have the new value be inherited by all descendants of the object, you cannot define the property in the object's constructor function. Instead, you add it to the constructor's associated prototype. For example, assume you change the preceding code to the following:

```
function Employee () {
    this.dept = "general";
}
Employee.prototype.name = "";

function WorkerBee () {
    this.projects = [];
}
WorkerBee.prototype = new Employee;

amy = new WorkerBee;

Employee.prototype.name = "Unknown";
```

In this case, the name property of amy becomes "Unknown".

As these examples show, if you want to have default values for object properties and you want to be able to change the default values at run time, you should set the properties in the constructor's prototype, not in the constructor function itself.

## Determining Instance Relationships

You may want to know what objects are in the prototype chain for an object, so that you can tell from what objects this object inherits properties. In a class-based language, you might have an `instanceof` operator for this purpose. JavaScript does not provide `instanceof`, but you can write such a function yourself.

As discussed in "Inheriting Properties" on page 169, when you use the `new` operator with a constructor function to create a new object, JavaScript sets the `__proto__` property of the new object to the value of the `prototype` property of the constructor function. You can use this to test the prototype chain.

For example, suppose you have the same set of definitions already shown, with the prototypes set appropriately. Create a `__proto__` object as follows:

```
chris = new Engineer("Pigman, Chris", ["jsd"], "fiji");
```

With this object, the following statements are all true:

```
chris.__proto__ == Engineer.prototype;
chris.__proto__.__proto__ == WorkerBee.prototype;
chris.__proto__.__proto__.__proto__ == Employee.prototype;
chris.__proto__.__proto__.__proto__.__proto__ == Object.prototype;
chris.__proto__.__proto__.__proto__.__proto__.__proto__ == null;
```

Given this, you could write an `instanceOf` function as follows:

```
function instanceOf(object, constructor) {
  while (object != null) {
    if (object == constructor.prototype)
      return true;
    object = object.__proto__;
  }
  return false;
}
```

With this definition, the following expressions are all true:

```
instanceOf (chris, Engineer)
instanceOf (chris, WorkerBee)
instanceOf (chris, Employee)
instanceOf (chris, Object)
```

But the following expression is false:

```
instanceOf (chris, SalesPerson)
```

## Global Information in Constructors

When you create constructors, you need to be careful if you set global information in the constructor. For example, assume that you want a unique ID to be automatically assigned to each new employee. You could use the following definition for `Employee`:

```
var idCounter = 1;

function Employee (name, dept) {
  this.name = name || "";
  this.dept = dept || "general";
  this.id = idCounter++;
}
```

With this definition, when you create a new `Employee`, the constructor assigns it the next ID in sequence and then increments the global ID counter. So, if your next statement is the following, `victoria.id` is 1 and `harry.id` is 2:

```
victoria = new Employee("Pigbert, Victoria", "pubs")
harry = new Employee("Tschopik, Harry", "sales")
```

At first glance that seems fine. However, `idCounter` gets incremented every time an `Employee` object is created, for whatever purpose. If you create the entire `Employee` hierarchy shown in this chapter, the `Employee` constructor is called every time you set up a prototype. Suppose you have the following code:

```
var idCounter = 1;

function Employee (name, dept) {
  this.name = name || "";
  this.dept = dept || "general";
  this.id = idCounter++;
}

function Manager (name, dept, reports) {...}
Manager.prototype = new Employee;

function WorkerBee (name, dept, projs) {...}
WorkerBee.prototype = new Employee;

function Engineer (name, projs, mach) {...}
Engineer.prototype = new WorkerBee;

function SalesPerson (name, projs, quota) {...}
SalesPerson.prototype = new WorkerBee;

mac = new Engineer("Wood, Mac");
```

Further assume that the definitions omitted here have the `base` property and call the constructor above them in the prototype chain. In this case, by the time the `mac` object is created, `mac.id` is 5.

Depending on the application, it may or may not matter that the counter has been incremented these extra times. If you care about the exact value of this counter, one possible solution involves instead using the following constructor:

```
function Employee (name, dept) {
  this.name = name || "";
  this.dept = dept || "general";
  if (name)
    this.id = idCounter++;
}
```

When you create an instance of `Employee` to use as a prototype, you do not supply arguments to the constructor. Using this definition of the constructor, when you do not supply arguments, the constructor does not assign a value to the `id` and does not update the counter. Therefore, for an `Employee` to get an assigned `id`, you must specify a name for the employee. In this example, `mac.id` would be 1.

## No Multiple Inheritance

Some object-oriented languages allow multiple inheritance. That is, an object can inherit the properties and values from unrelated parent objects. JavaScript does not support multiple inheritance.

Inheritance of property values occurs at run time by JavaScript searching the prototype chain of an object to find a value. Because an object has a single associated prototype, JavaScript cannot dynamically inherit from more than one prototype chain.

In JavaScript, you can have a constructor function call more than one other constructor function within it. This gives the illusion of multiple inheritance. For example, consider the following statements:

```
function Hobbyist (hobby) {
    this.hobby = hobby || "scuba";
}

function Engineer (name, projs, mach, hobby) {
    this.base1 = WorkerBee;
    this.base1(name, "engineering", projs);
    this.base2 = Hobbyist;
    this.base2(hobby);
    this.machine = mach || "";
}
Engineer.prototype = new WorkerBee;

dennis = new Engineer("Doe, Dennis", ["collabra"], "hugo")
```

Further assume that the definition of `WorkerBee` is as used earlier in this chapter. In this case, the `dennis` object has these properties:

```
dennis.name == "Doe, Dennis"
dennis.dept == "engineering"
dennis.projects == ["collabra"]
dennis.machine == "hugo"
dennis.hobby == "scuba"
```

So `dennis` does get the `hobby` property from the `Hobbyist` constructor. However, assume you then add a property to the `Hobbyist` constructor's prototype:

```
Hobbyist.prototype.equipment = ["mask", "fins", "regulator", "bcd"]
```

The `dennis` object does not inherit this new property.



# 3

## *Server-Side JavaScript Features*

- Quick Start with the Sample Applications
- Basics of Server-Side JavaScript
- Session Management Service
- Other JavaScript Functionality



# Quick Start with the Sample Applications

This chapter describes the sample server-side JavaScript applications that ship with Netscape web servers. It introduces using server-side JavaScript by working with two of the simpler sample applications.

This chapter contains the following sections:

- Hello World
- Hangman

When you install a Netscape web server, several sample JavaScript applications are also installed. For an introduction to the full capabilities of JavaScript applications, run them and browse their source code. You can also modify these applications as you learn about JavaScript's capabilities. Both the source files and the application executables for these applications are installed in the `$NSHOME\js\samples` directory, where `$NSHOME` is the directory in which you installed the server.

The following table lists the sample applications.

Table 11.1 Sample JavaScript applications

---

Basic concepts	
world	“Hello World” application.
hangman	The word-guessing game.
cipher	Word game that has the player guess a cipher.
LiveWire Database Service <sup>a</sup>	
dbadmin	Simple interactive SQL access using LiveWire. If you have restricted access to the Application Manager, this sample is also protected with the server administrator’s user name and password.
videoapp	Complete video store application using a relational database of videos.
oldvideo	An alternative version of the video store application.
LiveConnect <sup>b</sup>	
bugbase	Simple bug entry sample using LiveConnect.
flexi	Accessing remote services running on an IIOP-enabled ORB through LiveConnect. This sample is not automatically added to the list of applications in the Application Manager; you must add it before you can use it.
bank	Another IIOP sample. This sample is not automatically added to the list of applications in the Application Manager; you must add it before you can use it.
Other sample applications	
sendmail	Demonstrates the ability to send email messages from your JavaScript application.

---

Table 11.1 Sample JavaScript applications (Continued)

<code>viewer</code>	Allows you to view files on the server, using JavaScript's <code>File</code> class. For security reasons this application is not automatically installed with the Netscape server. If you install it, be sure to restrict access. Otherwise, unauthorized persons may be able to read and write files on your server. For information on restricting access to an application, see the administrator's guide for your web server.
<code>jsacall</code>	Sample using external native libraries and providing access to CGI variables.

a. These sample applications work only if you have a supported database server installed on your network and have configured the client software correctly. For more information, see Chapter 17, "Configuring Your Database." These applications are discussed in Chapter 20, "Videoapp and Oldvideo Sample Applications." Before using `videoapp` or `oldvideo`, follow the instructions to set them up found in that chapter.

b. These applications are discussed in Chapter 22, "Accessing CORBA Services."

**Note** In addition to sample applications, the `$NSHOME\js\samples` directory also contains an application named `metadata`. This application is used by Visual JavaScript. While you are welcome to browse its source code, do not modify the executable.

For more advanced sample applications, you can visit the Netscape AppFoundry Online home page at [http://home.netscape.com/one\\_stop/intranet\\_apps/index.html](http://home.netscape.com/one_stop/intranet_apps/index.html).

The rest of this chapter walks you through two of the simpler samples, giving you a feel for working with JavaScript applications. For now, don't worry about any of the details. This discussion is intended only to give you a rough idea of the capabilities of JavaScript applications. Details are discussed in later chapters.

Chapter 20, "Videoapp and Oldvideo Sample Applications," discusses the `videoapp` application in detail. You should read that chapter when you're ready to start working with the LiveWire Database Service.

# Hello World

In this section, you run Hello World, the simplest sample application, and get an introduction to

- Reading JavaScript source files
- Embedding JavaScript in HTML
- Building and restarting an application

To get started with the sample applications, you need to access the JavaScript Application Manager. You can do so by loading the following URL in Navigator:

```
http://server.domain/appmgr
```

In this and other URLs throughout this manual, *server* represents the name of the server on which you run your application, such as `research1` or `www`, and *domain* represents your Internet domain name, such as `netscape.com` or `uiuc.edu`. If your server has Secure Sockets Layer (SSL) enabled, use `https` instead of `http` in the URL.

In the Application Manager, select `world` in the left frame and click the Run button. Alternatively, you can enter the application URL in the Navigator Location field:

```
http://server.domain/world
```

In response, the Application Manager displays the page shown in Figure 11.1.

Figure 11.1 Hello World



For more information on the Application Manager, see Chapter 3, "How to Develop Server Applications."

## What Hello World Does

This application illustrates two important capabilities: maintaining a distinct client state for multiple clients and maintaining a persistent application state. Specifically, it performs these functions:

- Displays the IP address of the client accessing it
- Displays the names entered previously and provides a simple form for the user to enter a name
- Displays the number of times the user has previously accessed the page and the total number of times the page has been accessed by anyone

The first time a user accesses this page, the values for both names are not defined. The number of times the user has previously accessed the page is 0; the total number of times it has been accessed is 0.

Type your name and click Enter. The page now shows the name you entered following the text “This time you are.” Both numbers of accesses have been incremented. This action illustrates simple form processing. Enter another name and click Enter. The page now shows the new name following the text “This time you are” and the previous name following the text “Last time you were.” Again, both numbers of accesses have been incremented.

If you access the application from another instance of the Navigator (or from another computer), the page displays the total number of accesses and the number of accesses by each instance of Navigator, not just by that particular instance.

## Looking at the Source Script

Now take a look at the JavaScript source script for this application. Using your favorite text editor, open the file `$NSHOME\js\samples\world\hello.html`, where `$NSHOME` is the directory in which you installed the Netscape server. The file begins with some typical HTML:

```
<html>
<head>
<title> Hello World </title>

</head>
<body>
<h1> Hello World </h1>
<p>Your IP address is <server>write(request.ip);</server>
```

The `SERVER` tags in the bottom line enclose JavaScript code that is executed on the server. In this case, the statement `write(request.ip)` displays the `ip` property of the `request` object (the IP address of the client accessing the page). The `write` function is very important in server-side JavaScript applications, because you use it to add the values of JavaScript expressions to the HTML page sent to the client.



The `request` object is part of the JavaScript Session Management Service. For a full description of it, see Chapter 13, “Session Management Service.” The `write` function is one of the functions JavaScript defines that is not associated with a specific object. For information on the `write` function, see “Constructing the HTML Page” on page 225.

Then come some statements you shouldn’t worry about quite yet. These are the next statements of interest:

```
<server> client.oldname = request.newname;</server>
```

This statement assigns the value of the `newname` property of the `request` object to the `oldname` property of the `client` object. The `client` object is also part of the JavaScript Session Management Service. For a full description of it, see Chapter 13, “Session Management Service.” For now, just realize that `client` can hold information about an application specific to a particular browser running that application.

The value of `request.newname` is set when a user enters a value in the form. Later in the file you’ll find these form statements:

```
<form method="post" action="hello.html">
<input type="text" name="newname" size="20">
```

The value of the form’s `ACTION` attribute is `hello.html` (the current filename). This means that when the user submits the form by clicking Enter or pressing the Enter key, Navigator reloads the current page. In general, `ACTION` can be any page in a JavaScript application.

The value of the `NAME` attribute of the text field is `newname`. When the page is submitted, this statement assigns whatever the user has entered in the text field to the `newname` property of the `request` object, referred to in JavaScript as `request.newname`. The values of form elements always correspond to properties of the `request` object. Properties of the `request` object last only for a single client request.

A few lines down, there is another `SERVER` tag, indicating that the following lines are server-side JavaScript statements. Here is the first group of statements:

```
if (client.number == null)
    client.number = 0
else
    client.number = 1 + parseInt (client.number, 10)
```

This conditional statement checks whether the `number` property of the `client` object has been initialized. If not, then the code initializes it to 0; otherwise, it increments `number` by 1 using the JavaScript `parseInt` function, which converts the string value to a number. Because the predefined `client` object converts all property values to strings, you must use `parseInt` or `parseFloat` to convert these values to numbers.

Because `number` is a property of the `client` object, it is distinct for each different client that accesses the application. This value indicates the number of times “you have been here.”

To track the total number of accesses, you use the `project` object, because it is shared by all clients accessing the application. Properties of the `project` object persist until the application is stopped. The next group of statements tracks accesses:

```
project.lock()
if (project.number == null)
    project.number = 0
else
    project.number = 1 + project.number
project.unlock()
```

The first statement uses the `lock` method of the `project` object. This gives the client temporary exclusive access to the `project` object. Another `if` statement checks whether `project.number` has been defined. If not, then the code initializes it to 0; otherwise, the code increments it by 1. Finally, the `unlock` method releases the `project` object so that other clients can access it.

The final statements in the file display the values of `client.number` and `project.number`.

```
<p>You have been here <server>write(client.number);</server> times.
<br>This page has been accessed <server>write(project.number);
</server> times.
```

## Modifying Hello World

In this section, you modify, recompile, and restart this sample application. To edit the source file, you must first determine where it is. In case you don't remember, the Application Manager shows the directory path of the application executable (the file that has the suffix `.web`). The source file, `hello.html`, should be in the same directory. Edit the file with your favorite text editor. The HTML file starts with these statements:

```
<html>
<head> <title> Hello World </title> </head>

<body>
<h1> Hello World </h1>
<p>Your IP address is <server>write(request.ip);</server>

<server>
write ("<P>Last time you were " + client.oldname + ".");
</server>
<p>This time you are <server>write(request.newname);</server>

<server>client.oldname = request.newname; </server>
```

Add a line that displays the type of browser the user has:

```
<p>You are using <server>write(request.agent)</server>
```

If you want, you can also personalize the heading of the page; for example, you could make the title “Fred’s Hello World.”

When you’ve finished modifying the file, run the JavaScript application compiler. At the command prompt, change to the directory containing the source code. Type this line at the command prompt to compile the application:

```
jsac -v -o hello.web hello.html
```

Alternatively, from that directory you can run the `build` script (for Unix) or `build.bat` script (for NT). In either case, the compiler starts and displays messages. The final message should be “Compiled and linked successfully.”

Publish the application’s files on your development server. To restart, access the Application Manager, select Hello World, then choose Restart. This loads the newly compiled version of the application into the server. You can then run the application by choosing Run. A window opens with Hello World. You see the changes you made to the application.

# Hangman

In this section, you run and modify the Hangman sample application and get an introduction to:

- Using a JavaScript-only source file
- Correcting compile-time errors
- Using the trace utility for runtime debugging

Hangman is a classic word game in which the players try to guess a secret word. The unknown letters in the word are displayed on the screen as asterisks; the asterisks are replaced as a player guesses the correct letters. When the guess is incorrect, one more part of the hanged man is drawn. The game also shows the incorrect letters you have guessed.

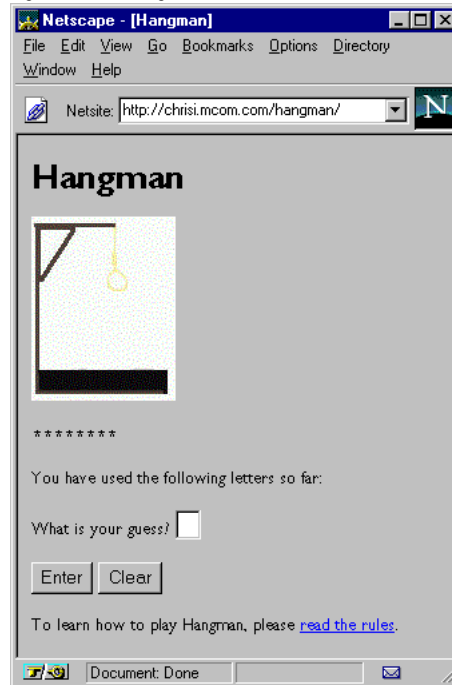
When the hanged man is completely drawn, the player loses the game. The player wins by guessing all the letters in the word before the man is hanged. In this simple version of the game, there are only three possible secret words. After a game, the player can choose to play again (and use the next secret word) or quit.

Run the Hangman application by selecting Hangman in the Application Manager and clicking Run. Alternatively, you can load the application URL in Navigator:

```
http://server.domain/hangman
```

In response, the Application Manager displays the page shown in the following figure.

Figure 11.2 Hangman



Play the game to get a feel for it.

## Looking at the Source Files

The following table shows the sources files for Hangman.

Table 11.2 Hangman source files

<code>hangman.html</code>	The main page for the application. This page is installed as the default page for Hangman in the Application Manager. It is displayed if the user enters just the <code>hangman</code> URL, with no specific page.
<code>hangman.js</code>	A file containing only server-side JavaScript functions used in <code>hangman.html</code> .
<code>youwon.html</code> <code>youlost.html</code> <code>thanks.html</code>	The pages displayed when a player wins, loses, and finishes playing the game, respectively.
<code>images</code> directory	Contains the Hangman images, <code>hang0.gif</code> , <code>hang1.gif</code> , and so on.
<code>rules.html</code>	Contains text explaining the game. This file is not compiled with the application; it is included as an example of an uncompiled application page that is part of the same site.

Most of the application logic is in `hangman.html`. The basic logic is simple:

1. For a new game, initialize the secret word and other variables.
2. If the player correctly guessed a letter, substitute it into the answer.
3. If the guess was wrong, increment the number of wrong (missed) guesses.
4. Check whether the user has won or lost.
5. Draw the current version of the hanged man, using a GIF image based on the number of wrong guesses.

The body of the HTML file `hangman.html` starts with some JavaScript code inside a `SERVER` tag. First comes code to initialize a new game:

```
if (client.gameno == null) {
    client.gameno = 1;
    client.newgame = "true";
}

if (client.newgame == "true") {
    if (client.gameno % 3 == 1)
        client.word = "LIVEWIRE";
    if (client.gameno % 3 == 2)
        client.word = "NETSCAPE";
    if (client.gameno % 3 == 0)
        client.word = "JAVASCRIPT";
    client.answer = InitAnswer(client.word);
    client.used = "";
    client.num_misses = 0;
}

client.newgame = "false";
```

This code makes extensive use of the `client` object to store information about this client playing the game. Because there is no state that needs to be saved between uses of this same application by different clients, the code doesn't use the `project` or `server` objects.

The first statement determines whether the player has played before, by checking if `client.gameno` exists; if not, the code initializes it to 1 and sets `client.newgame` to `true`. Then, some simple logic assigns the “secret word” to `client.word`; there are just three secret words that players cycle through as they play the game. The `client.gameno` property keeps track of how many times a particular player has played. The final part of initialization uses `InitAnswer`, a function defined in `hangman.js`, to initialize `client.answer` to a string of asterisks.

Then comes a block of statements to process the player's guess:

```
if (request.guess != null) {
    request.guess = request.guess.toUpperCase().substring(0,1);
    client.used = client.used + request.guess + " ";
    request.old_answer = client.answer;
    client.answer = Substitute (request.guess, client.word,
        client.answer);
    if (request.old_answer == client.answer)
        client.num_misses = parseInt(client.num_misses) + 1;
}
```

The `if` statement determines whether the player has made a guess (entered a letter in the form field). If so, the code calls `Substitute` (another function defined in `hangman.js`) to substitute the guessed letter into `client.answer`. This makes `client.answer` the answer so far (for example, "N\*T\*\*AP\*").

The second `if` statement checks whether `client.answer` has changed since the last guess; if not, then the code increments `client.num_misses` to keep track of the number of incorrect guesses. You must always use `parseInt` to work with integer property values of the predefined `client` object.

As shown in the following code, the final `if` statement in the JavaScript code checks whether the player has won or lost, and redirects the client accordingly. The `redirect` function opens the specified HTML file and passes control to it.

```
if (client.answer == client.word)
    redirect(addClient("youwon.html"));
else if (client.num_misses > 6)
    redirect(addClient("youlost.html"));
```

This is the end of the initial `SERVER` tag. HTML, augmented with more JavaScript expressions, begins. The hanged man is drawn by using a backquoted JavaScript expression inside an HTML `IMG` tag:

```
<IMG SRC=`"images\hang" + client.num_misses + ".gif"`>
```

The entire expression between the two backquotes (``) is a JavaScript string. It consists of the string literal `"images\hang"` concatenated with the value of `client.num_misses` (which represents an integer but is stored as a string), concatenated with the string literal `".gif"`. There are six GIF files containing the hanged man in different stages of completion: `image0.gif`, `image1.gif`, and so on. The backquoted JavaScript generates HTML of the form:

```
<IMG SRC="images\hang0.gif">
```

These lines follow:

```
<PRE><SERVER>write(client.answer)</SERVER></PRE>
You have used the following letters so far:
<SERVER>write(client.used)</SERVER>
```

They display the value of `client.answer` (the word containing all the correctly guessed letters) and all the guessed letters.

The remainder of the file consists of standard HTML. One important thing to notice is that the `ACTION` attribute of the `FORM` tag specifies `hangman.html` as the URL to which to submit the form. That means when you submit the form, the page is reloaded with the specified form values.

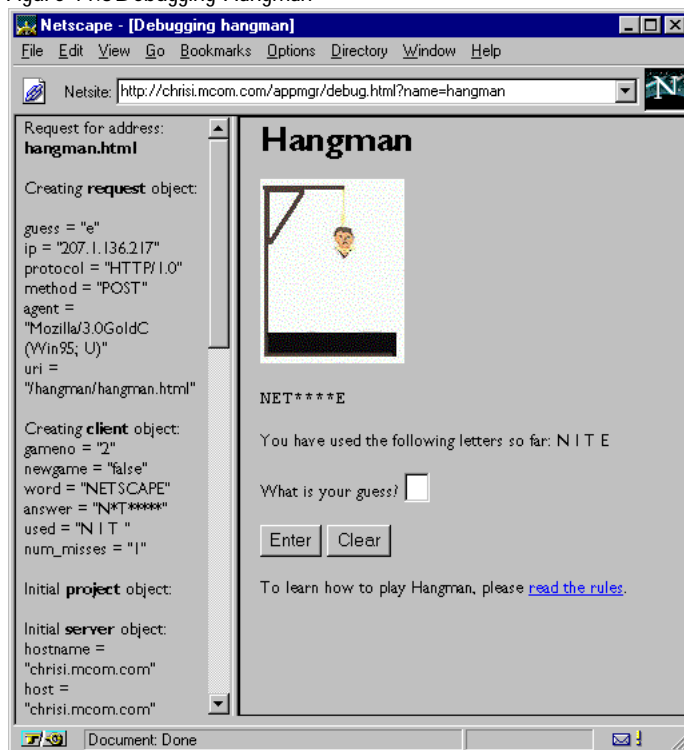


Examine `hangman.js`, an example of a server-side JavaScript-only source file. It defines two functions, `InitAnswer` and `Substitute`, used in the application. Notice that you do not use `SERVER` tags in a JavaScript-only file.

## Debugging Hangman

You can experiment more with JavaScript to get a feel for developing applications. One important task to master is debugging. In the Application Manager, select Hangman and choose Debug. The Application Manager opens a window with the application in one frame and debugging information in a narrow frame along the left side of the window, as shown in Figure 11.3.

Figure 11.3 Debugging Hangman



Notice that the URL is

`http://server.domain/appmgr/debug.html?name=hangman`

You can add a bookmark for this URL as a convenience while you work on Hangman. Then you don't have to go through the Application Manager.

Try adding a function to Hangman verifying that a player's guess is a letter (not a number or punctuation mark). You can use the function `InitAnswer` defined in `hangman.js` as a starting point. After compiling and restarting the application, use your bookmark to run the application in debug mode.

# Basics of Server-Side JavaScript

This chapter describes the basics of server-side JavaScript. It introduces server-side functionality and the differences between client-side and server-side JavaScript. The chapter describes how to embed server-side JavaScript in HTML files. It discusses what happens at runtime on the client and on the server, so that you can understand what to do when. The chapter describes how you use JavaScript to change the HTML page sent to the client and, finally, how to share information between the client and server processes.

This chapter contains the following sections:

- What to Do Where
- Overview of Runtime Processing
- Server-Side Language Overview
- Embedding JavaScript in HTML
- Runtime Processing on the Server
- Constructing the HTML Page
- Accessing CGI Variables
- Communicating Between Server and Client
- Garbage Collection

Server-side JavaScript contains the same core language as the client-side JavaScript with which you may already be familiar. The tasks you perform when running JavaScript on a server are quite different from those you perform when running JavaScript on a client. The different environments and tasks call for different objects.

# What to Do Where

The client (browser) environment provides the front end to an application. In this environment, for example, you display HTML pages in windows and maintain browser session histories of HTML pages displayed during a session. The objects in this environment, therefore, must be able to manipulate pages, windows, and histories.

By contrast, in the server environment you work with the resources on the server. For example, you can connect to relational databases, share information across users of an application, or manipulate the server's file system. The objects in this environment must be able to manipulate relational databases and server file systems.

In addition, an HTML page is not displayed on the server. It is retrieved from the server to be displayed on the client. The page retrieved can contain client-side JavaScript. If the requested page is part of a JavaScript application, the server may generate this page on the fly.

In developing a JavaScript application, keep in mind the differences between client and server platforms. They are compared in the following table.

Table 12.1 Client and server comparison

Servers	Clients
Servers are usually (though not always) high-performance workstations with fast processors and large storage capacities.	Clients are often (though not always) desktop systems with less processor power and storage capacity.
Servers can become overloaded when accessed by thousands of clients.	Clients are often single-user machines, so it can be advantageous to offload processing to the client.
	Preprocessing data on the client can also reduce bandwidth requirements, if the client application can aggregate data.

There are usually a variety of ways to partition an application between client and server. Some tasks can be performed only on the client or on the server; others can be performed on either. Although there is no definitive way to know what to do where, you can follow these general guidelines:

As a rule of thumb, use client processing (the `SCRIPT` tag) for these tasks:

- Validating user input; that is, checking that values entered in forms are valid
- Prompting a user for confirmation and displaying error or informational dialog boxes
- Performing aggregate calculations (such as sums or averages) or other processing on data retrieved from the server
- Conditionalizing HTML
- Performing other functions that do not require information from the server

Use server processing (the `SERVER` tag) for these tasks:

- Maintaining information through a series of client accesses
- Maintaining data shared among several clients or applications
- Accessing a database or files on the server
- Calling external libraries on the server
- Dynamically customizing Java applets; for example, visualizing data using a Java applet

JavaScript's Session Management Service provides objects to preserve information over time, but client-side JavaScript is more ephemeral. Client-side objects exist only as the user accesses a page. Also, servers can aggregate information from many clients and many applications and can store large amounts of data in databases. It is important to keep these characteristics in mind when partitioning functionality between client and server.

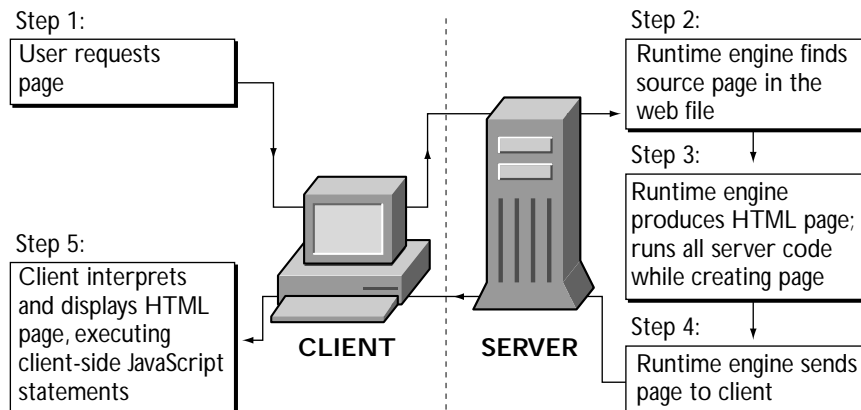
# Overview of Runtime Processing

Once you've installed and started a JavaScript application, users can access it. The basic procedure is as follows:

1. A user accesses the application URL with a web browser, such as Netscape Communicator. The web browser sends a client request to the server for a page in the application.
2. If the request is to a page under the application URL, the JavaScript runtime engine running on the server finds information in the web file corresponding to that URL. For details on what happens in this and the next two steps, see "Runtime Processing on the Server" on page 220.
3. The runtime engine constructs an HTML page to send to the client in response. It runs the bytecodes associated with `SERVER` tags from the original source code HTML, creating an HTML page based on those bytecodes and any other HTML found in the original. For information on how you can influence that page that is constructed, see "Constructing the HTML Page" on page 225.
4. The runtime engine sends the new HTML page (which may contain client-side JavaScript statements) to the client.
5. The JavaScript runtime engine inside the web browser interprets any client-side JavaScript statements, formats HTML output, and displays results to the user.

Figure 12.1 illustrates this process.

Figure 12.1 Processing a JavaScript page request



Of course, the user must have Netscape Navigator (or some other JavaScript-capable web browser), for the client to be able to interpret client-side JavaScript statements. Likewise, if you create a page containing server-side JavaScript, it must be installed on a Netscape server to operate properly.

For example, assume the client requests a page with this source:

```

<html>
<head> <title> Add New Customer </title> </head>

<body text="#FFFF00" bgcolor="#C0C0C0" background="blue_marble.gif">

<br>

<server>
if ( project.lock() ) {
    project.lastID = 1 + project.lastID;
    client.customerID = project.lastID;
    project.unlock();
}
</server>

<h1>Add a New Customer </h1>
<p>Note: <b>All</b> fields are required for the new customer
<form method="post" action="add.htm"></p>
<p>ID:
<br><server>write("<STRONG><FONT COLOR=\"#00FF00\">" +
    project.lastID + "</FONT></STRONG>");</server>

<!-- other html statements -->

</body>
</html>

```

When this page is accessed, the runtime engine on the server executes the code associated with the `SERVER` tags. (The code shown in bold.) If the new customer ID is 42, the server sends this HTML page to the client to be displayed:

```
<html>
<head> <title> Add New Customer </title> </head>

<body text="#FFFF00" bgcolor="#C0C0C0" background="blue_marble.gif">

<br>

<h1>Add a New Customer </h1>
<p>Note: <b>All</b> fields are required for the new customer
<form method="post" action="add.htm"></p>
<p>ID:
<br><STRONG><FONT COLOR="#00FF00">42</FONT></STRONG>

<!-- other html statements -->

</body>
</html>
```

## Server-Side Language Overview

Client-side and server-side JavaScript both implement the JavaScript 1.2 language. In addition, each adds objects and functions specific to working in the client or the server environment. For example, client-side JavaScript includes the `form` object to represent a form on an HTML page, whereas server-side JavaScript includes the `database` object for connecting to an external relational database.

The *Client-Side JavaScript Guide* discusses in detail the core JavaScript language and the additions specific to client-side JavaScript.

ECMA, the European standards organization for standardizing information and communication systems, derived its ECMA-262 standard from the JavaScript language. You can download the standard specification from ECMA's web site at <http://www.ecma.ch>.



# Core Language

For the most part, server-side JavaScript implements the JavaScript 1.2 language completely, as does client-side JavaScript. By default, however, server-side JavaScript differs from the JavaScript 1.2 specification in its treatment of comparison operators. Also, server-side JavaScript implements prototypes as defined in the specification, but the implications are somewhat different in the server environment than in the client environment. This section discusses these differences.

## Comparison Operators

The behavior of comparison operators changed between JavaScript 1.1 and JavaScript 1.2. JavaScript 1.1 provided automatic conversion of operands for comparison operators. In particular:

- If both operands are objects, JavaScript 1.1 compares object references.
- If either operand is null, JavaScript 1.1 converts the other operand to an object and compares references.
- If one operand is a string and the other is an object, JavaScript 1.1 converts the object to a string and compares string characters.
- Otherwise, JavaScript 1.1 converts both operands to numbers and compares numeric identity.

JavaScript 1.2 does not provide automatic conversion. In particular:

- JavaScript 1.2 never attempts to convert operands from one type to another.
- JavaScript 1.2 always compares the identity of operands of like type. If the operands are not of like type, they are not equal.

Server-side JavaScript can provide either behavior. By default, server-side JavaScript provides the automatic conversion of operands as was done in JavaScript 1.1. If you want to have server-side JavaScript have the JavaScript 1.2 behavior, you can specify the `-a 1.2` option to `jsac`, the JavaScript compiler. The compiler is described in “Compiling an Application” on page 59.

## Prototypes

As described in the *Server-Side JavaScript Reference*, you can use the `prototype` property of many classes to add new properties to a class and to all of its instances. As described in “Classes and Objects” on page 213, server-side JavaScript adds several classes and predefined objects. For the new classes that have the `prototype` property, it works for server-side JavaScript exactly as for client-side JavaScript.

You can use the `prototype` property to add new properties to the `Blob`, `Connection`, `Cursor`, `DbPool`, `File`, `Lock`, `Resultset`, `SendMail`, and `Stproc` classes. In addition, you can use the `prototype` property of the `DbBuiltin` class to add properties to the predefined database object. Note that you cannot create an instance of the `DbBuiltin` class; instead, you use the database object provided by the JavaScript runtime engine.

You cannot use `prototype` with the `client`, `project`, `request`, and `server` objects.

Also, as for client-side JavaScript, you can use the `prototype` property for any class that you define for your application.

Remember that all JavaScript applications on a server run in the same environment. This is why you can share information between clients and applications. One consequence of this, however, is that if you use the `prototype` property to add a new property to any of the server-side classes added by JavaScript, the new property is available to all applications running on the server, not just the application in which the property was added. This provides you with an easy mechanism for adding functionality to all JavaScript applications on your server.

By contrast, if you add a property to a class you define in your application, that property is available only to the application in which it was created.

## Usage

You need to be aware of how the JavaScript application compiler recognizes client-side and server-side JavaScript in an HTML file.

Client-side JavaScript statements can occur in several situations:

- By including them as statements and functions within a `SCRIPT` tag
- By specifying a file as JavaScript source to the `SCRIPT` tag
- By specifying a JavaScript expression as the value of an HTML attribute
- By including statements as event handlers within certain other HTML tags

For detailed information, see the *Client-Side JavaScript Guide*.

Server-side JavaScript statements can occur in these situations:

- By including them as statements and functions within a `SERVER` tag
- By specifying a file as JavaScript source to the JavaScript application compiler
- By specifying a JavaScript expression as the value or name of an HTML attribute

Notice that you cannot specify a server-side JavaScript statement as an event handler. For more information, see “Embedding JavaScript in HTML” on page 216.

## Environment

The LiveConnect feature of the core JavaScript language works differently on the server than it does on the client. For more information, see Chapter 21, “LiveConnect Overview.”

JavaScript provides additional functionality without the use of objects. You access this functionality through functions not associated with any object (global functions). The core JavaScript language provides the global functions described in the following table.

Table 12.2 Core JavaScript global functions

Function	Description
<code>escape</code>	Returns the hexadecimal encoding of an argument in the ISO Latin-1 character set; used to create strings to add to a URL.
<code>unescape</code>	Returns the ASCII string for the specified value; used in parsing a string added to a URL.
<code>isNaN</code>	Evaluates an argument to determine if it is not a number.
<code>parseFloat</code>	Parses a string argument and returns a floating-point number.
<code>parseInt</code>	Parses a string argument and returns an integer.

Server-side JavaScript adds the global functions described in the following table.

Table 12.3 JavaScript server-side global functions

Function	Description
<code>write</code>	Adds statements to the client-side HTML page being generated. (See “Generating HTML” on page 226.)
<code>flush</code>	Flushes the output buffer. (See “Flushing the Output Buffer” on page 226.)
<code>redirect</code>	Redirects the client to the specified URL. (See “Changing to a New Client Request” on page 227.)
<code>getOptionValue</code>	Gets values of individual options in an HTML <code>SELECT</code> form element. (See “Using Select Lists” on page 234.)
<code>getOptionValueCount</code>	Gets the number of options in an HTML <code>SELECT</code> form element. (See “Using Select Lists” on page 234.)
<code>debug</code>	Displays values of expressions in the trace window or frame. (See “Using the debug Function” on page 70.)
<code>addClient</code>	Appends client information to URLs. (See “Manually Appending client Properties to URLs” on page 277.)
<code>registerCFunction</code>	Registers a native function for use in server-side JavaScript. (See “Registering Native Functions” on page 300.)

Table 12.3 JavaScript server-side global functions (Continued)

Function	Description
<code>callC</code>	Calls a native function. (See “Using Native Functions in JavaScript” on page 300.)
<code>deleteResponseHeader</code>	Removes information from the s sent to the client. (See “Request and Response Manipulation” on page 302.)
<code>addResponseHeader</code>	Adds new information to the response header sent to the client. (See “Request and Response Manipulation” on page 302.)
<code>ssjs_getClientID</code>	Returns the identifier for the <code>client</code> object used by some of JavaScript’s client-maintenance techniques. (See “Uniquely Referring to the client Object” on page 255.)
<code>ssjs_generateClientID</code>	Returns an identifier you can use to uniquely specify the <code>client</code> object. (See “Uniquely Referring to the client Object” on page 255.)
<code>ssjs_getCGIVariable</code>	Returns the value of the specified CGI environment variable. (See “Accessing CGI Variables” on page 228.)

## Classes and Objects

To support the different tasks you perform on each side, JavaScript has classes and predefined objects that work on the client but not on the server and other classes and predefined objects that work on the server but not on the client.

**Important** These names of these objects are reserved for JavaScript. Do not create your own objects using any of these names.

The core JavaScript language provides the classes described in the following table. For details of all of these objects, see the *Server-Side JavaScript Reference*.

Table 12.4 Core JavaScript classes

Class	Description
Array	Represents an array.
Boolean	Represents a Boolean value.
Date	Represents a date.
Function	Specifies a string of JavaScript code to be compiled as a function.
Math	Provides basic math constants and functions; for example, its <code>PI</code> property contains the value of pi.
Number	Represents primitive numeric values.
Object	Contains the base functionality shared by all JavaScript objects.
Packages	Represents a Java package in JavaScript. Used with LiveConnect.
String	Represents a JavaScript string.

Server-side JavaScript includes the core classes, but not classes added by client-side JavaScript. Server-side JavaScript has its own set of additional classes to support needed functionality, as described in the following table.

Table 12.5 Server-side JavaScript classes

Class	Description
Connection	Represents a single database connection from a pool of connections. (See “Individual Database Connections” on page 323.)
Cursor	Represents a database cursor. (See “Manipulating Query Results with Cursors” on page 338.)
DbPool	Represents a pool of database connections. (See “Database Connection Pools” on page 314.)
Stproc	Represents a database stored procedure. (See “Calling Stored Procedures” on page 354.)
ResultSet	Represents the information returned by a database stored procedure. (See “Calling Stored Procedures” on page 354.)
File	Provides access to the server’s file system. (See “File System Service” on page 290.)

Table 12.5 Server-side JavaScript classes (Continued)

Class	Description
<code>Lock</code>	Provides functionality for safely sharing data among requests, clients, and applications. (See “Sharing Objects Safely with Locking” on page 279.)
<code>SendMail</code>	Provides functionality for sending electronic mail from your JavaScript application. (See “Mail Service” on page 287.)

In addition, server-side JavaScript has the predefined objects described in the following table. These objects are all available for each HTTP request. You cannot create additional instances of any of these objects.

Table 12.6 Server-side JavaScript singleton objects

Object	Description
<code>database</code>	Represents a database connection. (See “Approaches to Connecting” on page 311.)
<code>client</code>	Encapsulates information about a client/application pair, allowing that information to last longer than a single HTTP request. (See “The client Object” on page 252.)
<code>project</code>	Encapsulates information about an application that lasts until the application is stopped on the server. (See “The project Object” on page 260.)
<code>request</code>	Encapsulates information about a single HTTP request. (See “The request Object” on page 249.)
<code>server</code>	Encapsulates global information about the server that lasts until the server is stopped. (See “The server Object” on page 261.)

# Embedding JavaScript in HTML

There are two ways to embed server-side JavaScript statements in an HTML page:

- With the `SERVER` tag

Use this tag to enclose a single JavaScript statement or several statements. You precede the JavaScript statements with `<SERVER>` and follow them with `</SERVER>`.

You can intermix the `SERVER` tag with complete HTML statements. Never put the `SERVER` tag between the open bracket (`<`) and close bracket (`>`) of a single HTML tag. (See “The `SERVER` tag” on page 217.) Also, do not use the `<SCRIPT>` tag between `<SERVER>` and `</SERVER>`.

- With a backquote (```), also known as a tick

Use this character to enclose a JavaScript expressions inside an HTML tag, typically to generate an HTML attribute or attribute value based on JavaScript values. This technique is useful inside tags such as anchors, images, or form element tags, for example, to provide the value of an anchor's `HREF` attribute.

Do not use backquotes to enclose JavaScript expressions outside HTML tags. (See “Backquotes” on page 218.)

When you embed server-side JavaScript in an HTML page, the JavaScript runtime engine on the server executes the statements it encounters while processing the page. Most statements perform some action on the server, such as opening a database connection or locking a shared object. However, when you use the `write` function in a `SERVER` tag or enclose statements in backquotes, the runtime engine dynamically generates new HTML to modify the page it sends to the client.



## The SERVER tag

The `SERVER` tag is the most common way to embed server-side JavaScript in an HTML page. You can use the `SERVER` tag in any situation; typically, however, you use backquotes instead if you're generating attributes names or values for the HTML page.

Most statements between the `<SERVER>` and `</SERVER>` tags do not appear on the HTML page sent to the client. Instead, the statements are executed on the server. However, the output from any calls to the `write` function do appear in the resulting HTML.

The following excerpt from the Hello World sample application illustrates these uses:

```
<P>This time you are
<SERVER>
write(request.newname);
client.oldname = request.newname;
</SERVER>
<h3>Enter your name</h3>
```

When given this code snippet, the runtime engine on the server generates HTML based on the value of `request.newname` in the `write` statement. In the second statement, it simply performs a JavaScript operation, assigning the value of `request.newname` to `client.oldname`. It does not generate any HTML. So, if `request.newname` is "Mr. Ed," the runtime engine generates the following HTML for the previous snippet:

```
<P>This time you are
Mr. Ed
<h3>Enter your name</h3>
```

## Backquotes

Use backquotes (``) to enclose server-side JavaScript expressions as substitutes for HTML attribute names or attribute values. JavaScript embedded in HTML with backquotes automatically generates HTML; you do not need to use `write`.

In general, HTML tags are of the form

```
<TAG ATTRIB="value" [...ATTRIB="value"]>
```

where *ATTRIB* is an attribute and "value" is its value. The bracketed expression indicates that any number of attribute/value pairs is possible.

When you enclose a JavaScript expression in backquotes to be used as an attribute value, the JavaScript runtime engine automatically adds quotation marks for you around the entire value. You do not provide quotation marks yourself for this purpose, although you may need them to delimit string literals in the expression, as in the example that follows. The runtime engine does not do this for attribute names, because attribute names are not supposed to be enclosed in quotation marks.

For example, consider the following line from the Hangman sample application:

```
<IMG SRC=`"images\hang" + client.num_misses + ".gif"`>
```

This line dynamically generates the name of the image to use based on the value of `client.num_misses`. The backquotes enclose a JavaScript expression that concatenates the string "images\hang" with the integer value of `client.num_misses` and the string ".gif", producing a string such as "images\hang0.gif". The result is HTML such as

```
<IMG SRC="images\hang0.gif">
```

The order of the quotation marks is critical. The backquote comes first, indicating that the following value is a JavaScript expression, consisting of a string ("images\hang"), concatenated with an integer (`client.num_misses`), concatenated with another string (".gif"). JavaScript converts the entire expression to a string and adds the necessary quotation marks around the attribute value.

You need to be careful about using double quotation marks inside backquotes, because the value they enclose is interpreted as a literal value. For this reason, do not surround JavaScript expressions you want evaluated with quotation marks. For example, if the value of `client.val` is `NetHead`, then this statement:

```
<A NAME='client.val'>
```

generates this HTML:

```
<A NAME="NetHead">
```

But this statement:

```
<A NAME=`"client.val"`>
```

generates this HTML:

```
<A NAME="client.val">
```

As another example, two of the `ANCHOR` tag's attributes are `HREF` and `NAME`. `HREF` makes the tag a hyperlink, and `NAME` makes it a named anchor. The following statements use the `choice` variable to set the `attrib` and `val` properties of the `client` object and then create either a hyperlink or a target, depending on those values:

```
<SERVER>
if (choice == "link") {
    client.attrib = "HREF";
    client.val = "http://www.netscape.com";
}
if (choice == "target") {
    client.attrib = "NAME";
    client.val = "NetHead";
}
</SERVER>

<A 'client.attrib'='client.val'>Netscape Communications</A>
```

If the value of `choice` is `"link"`, the result is

```
<A HREF="http://home.netscape.com">Netscape Communications</A>
```

If the value of `choice` is `"target"`, the result is

```
<A NAME="NetHead">Netscape Communications</A>
```

## When to Use Each Technique

In most cases, it is clear when to use the `SERVER` tag and when to use backquotes. However, sometimes you can achieve the same result either way. In general, it is best to use backquotes to embed JavaScript values inside HTML tags, and to use the `SERVER` tag elsewhere.

For example, in Hangman, instead of writing

```
<IMG SRC=`"images\hang" + client.num_misses + ".gif"`>
```

you could write

```
<SERVER>
write("<IMG SRC=\"images\hang\"");
write(client.num_misses);
write(".gif\">");
</SERVER>
```

Notice the backslash that lets you use a quotation mark inside a literal string. Although the resulting HTML is the same, in this case backquotes are preferable because the source is easier to read and edit.

## Runtime Processing on the Server

“Overview of Runtime Processing” on page 206 gives an overview of what happens at runtime when a single user accesses a page in a JavaScript application. This section provides more details about steps 2 through 4 of this process, so you can better see what happens at each stage. This description provides a context for understanding what you can do on the client and on the server.

One of the most important things to remember when thinking about JavaScript applications is the asynchronous nature of processing on the Web. JavaScript applications are designed to be used by many people at the same time. The JavaScript runtime engine on the server handles requests from many different users as they come in and processes them in the order received.

Unlike a traditional application that is run by a single user on a single machine, your application must support the interleaving of multiple simultaneous users. In fact, since each frame in an HTML document with multiple frames generates its own request, what might seem to be a single request to the end user can appear as several requests to the runtime engine.

HTTP (Hypertext Transfer Protocol) is the protocol by which an HTML page is sent to a client. This protocol is *stateless*, that is, information is not preserved between requests. In general, any information needed to process an HTTP request needs to be sent with the request. This poses problems for many applications. How do you share information between different users of an application or even between different requests by the same user? JavaScript's Session Management Service was designed to help with this problem. This service is discussed in detail in Chapter 13, "Session Management Service." For now simply remember that the runtime engine automatically maintains the `client`, `server`, `project`, and `request` objects for you.

When the Netscape server receives a client request for an application page, it first performs authorization. This step is part of the basic server administration functions. If the request fails server authorization, then no subsequent steps are performed. If the request receives server authorization, then the JavaScript runtime engine continues. The runtime engine performs these steps, described in the following sections:

1. Constructs a new request object and constructs or restores the client object.
2. Finds the page for the request and starts constructing an HTML page to send to the client.
3. For each piece of the source HTML page, adds to the buffer or executes code.
4. Saves the client object properties.
5. Sends HTML to the client.
6. Destroys the request object and saves or destroys the client object.

## Step 1. Construct request object and construct or restore client object

It initializes the built-in properties of the `request` object, such as the request's IP address and any form input elements associated with the request. If the URL for the request specifies other properties, those are initialized for the `request` object, as described in “Encoding Information in a URL” on page 235.

If the `client` object already exists, the runtime engine retrieves it based on the specified client-maintenance technique. (See “Techniques for Maintaining the client Object” on page 263.) If no `client` object exists, the runtime engine constructs a new object with no properties.

You cannot count on the order in which these objects are constructed.

## Step 2. Find source page and start constructing HTML page

When you compiled your JavaScript application, the source included HTML pages containing server-side JavaScript statements. The main goal of the runtime engine is to construct, from one of those source pages, an HTML page containing only HTML and client-side JavaScript statements. As it creates this HTML page, the runtime engine stores the partially created page in a special area of memory called a buffer until it is time to send the buffered contents to the client.

## Step 3. Add to output buffer or execute code

This step is performed for each piece of code on the source page. The details of the effects of various server-side statements are covered in other sections of this manual. For more information, see “Constructing the HTML Page” on page 225.

For a given request, the runtime engine keeps performing this step until one of these things happens:

- The buffer contains 64KB of HTML.

In this situation, the runtime engine performs steps 4 and 5 and then returns to step 3 with a newly emptied buffer and continues processing the same request. (Step 4 is only executed once, even if steps 3 and 5 are repeated.)

- The server executes the `flush` function.  
In this situation, the runtime engine performs steps 4 and 5 and then returns to step 3 with a newly emptied buffer and continues processing the same request. (Step 4 is only executed once, even if steps 3 and 5 are repeated.)
- The server executes the `redirect` function.  
In this situation, the runtime engine finishes this request by performing steps 4 through 6. It ignores anything occurring after the `redirect` function in the source file and starts a new request for the page specified in the call to `redirect`.
- It reaches the end of the page.  
In this situation, the runtime engine finishes this request by performing steps 4 through 6.

## Step 4. Save client object properties

The runtime engine saves the `client` object's properties immediately before the *first* time it sends part of the HTML page to the client. It only saves these properties once. The runtime engine can repeat steps 3 and 5, but it cannot repeat this step.

The runtime engine saves the properties at this point to support some of the maintenance techniques for the `client` object. For example, the *client URL encoding* scheme sends the `client` properties in the header of the HTML file. Because the header is sent as the first part of the file, the `client` properties must be sent then.

As a consequence, you should be careful of where in your source file you set `client` properties. You should always change `client` properties in the file before any call to `redirect` or `flush` and before generating 64KB of HTML output.

If you change property values for the `client` object in the code after HTML has been sent to the client, those changes remain in effect for the rest of that client request, but they are then discarded. Hence, the next client request does not get those values for the properties; it gets the values that were in effect when content was first sent to the client. For example, assume your code contains these statements:

```
<HTML>
<P>The current customer is
<SERVER>
client.customerName = "Mr. Ed";
write(client.customerName);
client.customerName = "Mr. Bill";
</SERVER>

<P>The current customer really is
<SERVER>
write(client.customerName);
</SERVER>
</HTML>
```

This series of statements results in this HTML being sent to the client:

```
<P>The current customer is Mr. Ed
<P>The current customer really is Mr. Bill
```

And when the next client request occurs, the value of `client.customerName` is “Mr. Bill”. This very similar set of statements results in the same HTML:

```
<HTML>
<P>The current customer is
<SERVER>
client.customerName = "Mr. Ed";
write(client.customerName);
flush();
client.customerName = "Mr. Bill";
</SERVER>
<P>The current customer really is
<SERVER>
write(client.customerName);
</SERVER>
</HTML>
```

However, when the next client request occurs, the value of `client.customerName` is “Mr. Ed”; it is *not* “Mr. Bill”.

For more information, see “Techniques for Maintaining the client Object” on page 263.



## Step 5. Send HTML to client

The server sends the page content to the client. For pages with no server-side JavaScript statements, the server simply transfers HTML to the client. For other pages, the runtime engine performs the application logic to construct HTML and then sends the generated page to the client.

## Step 6. Destroy request object and save or destroy client object

The runtime engine destroys the `request` object constructed for this client request. It saves the values of the `client` object and then destroys the physical JavaScript object. It does not destroy either the `project` or the `server` object.

# Constructing the HTML Page

When you compile your JavaScript application, the source includes HTML pages that contain server-side JavaScript statements and perhaps also HTML pages that do not contain server-side JavaScript statements. When a user accesses a page in an application that does not contain server-side JavaScript statements, the server sends the page back as it would any other HTML page. When a user accesses a page that does contain server-side JavaScript statements, the runtime engine on the server constructs an HTML page to send in response, using one of the source pages of your application.

The runtime engine scans the source page. As it encounters HTML statements or client-side JavaScript statements, it appends them to the page being created. As it encounters server-side JavaScript statements, it executes them. Although most server-side JavaScript statements perform processing on the server, some affect the page being constructed. The following sections discuss three functions—`write`, `flush`, and `redirect`—that affect the HTML page served.

## Generating HTML

As discussed earlier in this chapter, the `write` function generates HTML based on the value of JavaScript expression given as its argument. For example, consider this statement

```
write("<P>Customer Name is:" + project.custname + ".");
```

In response to this statement, JavaScript generates HTML including a paragraph tag and some text, concatenated with the value of the `custname` property of the `project` object. For example, if this property is “Fred’s software company”, the client would receive the following HTML:

```
<P>Customer Name is: Fred’s software company.
```

As far as the client is concerned, this is normal HTML on the page. However, it is actually generated dynamically by the JavaScript runtime engine.

## Flushing the Output Buffer

To improve performance, JavaScript buffers the HTML page it constructs. The `flush` function immediately sends data from the internal buffer to the client. If you do not explicitly call the `flush` function, JavaScript sends data to the client after each 64KB of content in the constructed HTML page.

Don’t confuse the `flush` function with the `flush` method of the `File` class. (For information on using the `File` class to perform file input and output, see “File System Service” on page 290.)

You can use `flush` to control the timing of data transfer to the client. For example, you might choose to flush the buffer before an operation that creates a delay, such as a database query. Also, if a database query retrieves a large number of rows, flushing the buffer every few rows prevents long delays in displaying data.

**Note** If you use the client cookie technique to maintain the properties of the `client` object, you must make all changes to the `client` object before flushing the buffer. For more information, see “Techniques for Maintaining the client Object” on page 263.

The following code fragment shows how `flush` is used. Assume that your application needs to perform some action on every customer in your customer database. If you have a lot of customers, this could result in a lengthy delay. So that the user doesn't have to wait in front of an unchanging screen, your application could send output to the client before starting the processing and then again after processing each row. To do so, you could use code similar to the following:

```
flush();
conn.beginTransaction();
cursor = conn.cursor ("SELECT * FROM CUSTOMER", true);
while ( cursor.next() ) {
    // ... process the row ...
    flush();
}
conn.commitTransaction();
cursor.close();
```

## Changing to a New Client Request

The `redirect` function terminates the current client request and starts another for the specified URL. For example, assume you have this statement:

```
redirect("http://www.royalairways.com/apps/page2.html");
```

When the runtime engine executes this statement, it terminates the current request. The runtime engine does not continue to process the original page. Therefore any HTML or JavaScript statements that follow the call to `redirect` on the original page are lost. The client immediately loads the indicated page, discarding any previous content.

The parameter to `redirect` can be any server-side JavaScript statement that evaluates to a URL. In this way, you can dynamically generate the URL used in `redirect`. For example, if a page defines a variable `choice`, you can redirect the client to a page based on the value of `choice`, as follows:

```
redirect ("http://www.royalairways.com/apps/page"
        + choice + ".html");
```

If you want to be certain that the current `client` properties are available in the new request, and you're using one of the URL-based maintenance techniques for the `client` object, you should encode the properties in the URL you pass to `redirect`. For information on doing so, see “Manually Appending client Properties to URLs” on page 277.

In general, properties of the `request` object and top-level JavaScript variables last only for a single client request. When you redirect to a new page, you may want to maintain some of this information for multiple requests. You can do so by appending the property names and values to the URL, as described in “Encoding Information in a URL” on page 235.

## Accessing CGI Variables

Like most web servers, Netscape servers set values for a particular set of environment variables, called CGI variables, when setting up the context for running a CGI script. Writers of CGI scripts expect to be able to use these variables in their scripts.

By contrast, Netscape web servers do not set up a separate environment for server-side JavaScript applications. Nevertheless, some of the information typically set in CGI variables can also be useful in JavaScript applications. The runtime engine provides several mechanisms for accessing this information:

- By accessing properties of the predefined `request` object
- By using the `ssjs_getCGIVariable` function to access some CGI variables and other environment variables
- By using the `httpHeader` method of `request` to access properties of the client request header

The following table lists properties of the `request` object that correspond to CGI variables. For more information on these properties and on the `request` object in general, see “The request Object” on page 249.

Table 12.7 CGI variables accessible as properties of the `request` object

CGI variable	Property	Description
<code>AUTH_TYPE</code>	<code>auth_type</code>	The authorization type, if the request is protected by any type of authorization. Netscape web servers support HTTP basic access authorization. Example value: <code>basic</code>
<code>REMOTE_USER</code>	<code>auth_user</code>	The name of the local HTTP user of the web browser, if HTTP access authorization has been activated for this URL. Note that this is not a way to determine the user name of any person accessing your program. Example value: <code>ksmith</code>
<code>REQUEST_METHOD</code>	<code>method</code>	The HTTP method associated with the request. An application can use this to determine the proper response to a request. Example value: <code>GET</code>
<code>SERVER_PROTOCOL</code>	<code>protocol</code>	The HTTP protocol level supported by the client's software. Example value: <code>HTTP/1.0</code>
<code>QUERY_STRING</code>	<code>query</code>	Information from the requesting HTML page; if <code>"?"</code> is present, the information in the URL that comes after the <code>"?"</code> . Example value: <code>x=42</code>

The server-side function `ssjs_getCGIVariable` lets you access the environment variables set in the server process, including the CGI variables listed in the following table.

Table 12.8 CGI variables accessible through `ssjs_getCGIVariable`

Variable	Description
<code>AUTH_TYPE</code>	The authorization type, if the request is protected by any type of authorization. Netscape web servers support HTTP basic access authorization. Example value: <code>basic</code>
<code>HTTPS</code>	If security is active on the server, the value of this variable is <code>ON</code> ; otherwise, it is <code>OFF</code> . Example value: <code>ON</code>
<code>HTTPS_KEYSIZE</code>	The number of bits in the session key used to encrypt the session, if security is on. Example value: <code>128</code>

Table 12.8 CGI variables accessible through `ssjs_getCGIVariable` (Continued)

Variable	Description
<code>HTTPS_SECRETKEYSIZE</code>	The number of bits used to generate the server's private key. Example value: 128
<code>PATH_INFO</code>	Path information, as sent by the browser. Example value: <code>/cgivars/cgivars.html</code>
<code>PATH_TRANSLATED</code>	The actual system-specific pathname of the path contained in <code>PATH_INFO</code> . Example value: <code>/usr/ns-home/myhttpd/js/samples/cgivars/cgivars.html</code>
<code>QUERY_STRING</code>	Information from the requesting HTML page; if "?" is present, the information in the URL that comes after the "?". Example value: <code>x=42</code>
<code>REMOTE_ADDR</code>	The IP address of the host that submitted the request. Example value: <code>198.93.95.47</code>
<code>REMOTE_HOST</code>	If DNS is turned on for the server, the name of the host that submitted the request; otherwise, its IP address. Example value: <code>www.netscape.com</code>
<code>REMOTE_USER</code>	The name of the local HTTP user of the web browser, if HTTP access authorization has been activated for this URL. Note that this is not a way to determine the user name of any person accessing your program. Example value: <code>ksmith</code>
<code>REQUEST_METHOD</code>	The HTTP method associated with the request. An application can use this to determine the proper response to a request. Example value: <code>GET</code>
<code>SCRIPT_NAME</code>	The pathname to this page, as it appears in the URL. Example value: <code>cgivars.html</code>
<code>SERVER_NAME</code>	The hostname or IP address on which the JavaScript application is running, as it appears in the URL. Example value: <code>piccolo.mcom.com</code>
<code>SERVER_PORT</code>	The TCP port on which the server is running. Example value: <code>2020</code>
<code>SERVER_PROTOCOL</code>	The HTTP protocol level supported by the client's software. Example value: <code>HTTP/1.0</code>
<code>SERVER_URL</code>	The URL that the user typed to access this server. Example value: <code>https://piccolo:2020</code>

The syntax of `ssjs_getCGIVariable` is shown here:

```
value = ssjs_getCGIVariable("name");
```

This statement sets the variable `value` to the value of the `name` CGI variable. If you supply an argument that isn't one of the CGI variables listed in Table 12.8, the runtime engine looks for an environment variable by that name in the server environment. If found, the runtime engine returns the value; otherwise, it returns null. For example, the following code assigns the value of the standard `CLASSPATH` environment variable to the JavaScript variable `classpath`:

```
classpath = ssjs_getCGIVariable("CLASSPATH");
```

The `httpHeader` method of `request` returns the header of the current client request. For a CGI script, Netscape web servers set CGI variables for some of the information in the header. For JavaScript applications, you get that information directly from the header. Table 12.9 shows information available as CGI variables in the CGI environment, but as header properties in server-side JavaScript. In header properties, the underlines in the CGI-variable name (`_`) are replaced with dashes (`-`); for example, the CGI variable `CONTENT_LENGTH` corresponds to the header property `content-length`.

Table 12.9 CGI variables accessible through the client header

CGI variable	Description
<code>CONTENT_LENGTH</code>	The number of bytes being sent by the client.
<code>CONTENT_TYPE</code>	The type of data being sent by the client, if a form is submitted with the <code>POST</code> method.
<code>HTTP_ACCEPT</code>	Enumerates the types of data the client can accept.
<code>HTTP_USER_AGENT</code>	Identifies the browser software being used to access your program.
<code>HTTP_IF_MODIFIED_SINCE</code>	A date, set according to GMT standard time, allowing the client to request a response be sent only if the data has been modified since the given date.

For more information on manipulating the client header, see “Request and Response Manipulation” on page 302.

The following table shows the CGI variables that are not supported by server-side JavaScript, because they are not applicable when running JavaScript applications.

Table 12.10 CGI variables not supported by server-side JavaScript

Variable	Description
<code>GATEWAY_INTERFACE</code>	The version of CGI running on the server. Not applicable to JavaScript applications.
<code>SERVER_SOFTWARE</code>	The type of server you are running. Not available to JavaScript applications.

## Communicating Between Server and Client

Frequently your JavaScript application needs to communicate information either from the server to the client or from the client to the server. For example, when a user first accesses the `videoapp` application, the application dynamically generates the list of movie categories from the current database contents. That information, generated on the server, needs to be communicated back to the client. Conversely, when the user picks a category from that list, the user's choice must be communicated back to the server so that it can generate the set of movies.

### Sending Values from Client to Server

Here are several ways to send information from the client to the server:

- The runtime engine automatically creates properties of the `request` object for each value in an HTML form. (See “Accessing Form Values” on page 233.)
- If you're using a URL-based maintenance technique for properties of the `client` object, you can modify the URL sent to the server to include property values for the `client` and `request` objects. (See “Encoding Information in a URL” on page 235.)
- You can use cookies to set property values for the `client` and `request` objects. (See “Using Cookies” on page 239.)



- On the client, you can modify the header of the client request. You can then use the `httpHeader` method of the `request` object to manipulate the header and possibly the body of the request. (See “Request and Response Manipulation” on page 302.)
- You can use LiveConnect with CORBA services to communicate between client and server. (See Chapter 22, “Accessing CORBA Services.”)

## Accessing Form Values

Forms are the bread and butter of a JavaScript application. You use form elements such as text fields and radio buttons as the primary mechanism for transferring data from the client to the server. When the user clicks a Submit button, the browser submits the values entered in the form to the server for processing.

The `ACTION` attribute of the `FORM` tag determines the application to which the values are submitted. To send information to the application on the server, use an application URL as the value of the `ACTION` attribute.

If the document containing the form is a compiled part of the same application, you can simply supply the name of the page instead of a complete URL. For example, here is the `FORM` tag from the Hangman sample application:

```
<FORM METHOD="post" ACTION="hangman.html">
```

Forms sent to server-side JavaScript applications can use either `get` or `post` as the value of the `METHOD` attribute.

**Note** Server-side JavaScript applications do not automatically support file upload. That is, if the action specified is a page in a JavaScript application and you submit an `INPUT` element of `TYPE="file"`, your application must manually handle the file, as described in “Request and Response Manipulation” on page 302.

Each input element in an HTML form corresponds to a property of the `request` object. The property name is specified by the `NAME` attribute of the form element. For example, the following HTML creates a `request` property called `guess` that accepts a single character in a text field. You refer to this property in server-side JavaScript as `request.guess`.

```
<FORM METHOD="post" ACTION="hangman.html">
<P>
What is your guess?
<INPUT TYPE="text" NAME="guess" SIZE="1">
```

A `SELECT` form element that allows multiple selections requires special treatment, because it is a single property that can have multiple values. You can use the `getOptionValue` function to retrieve the values of selected options in a multiple select list. For more information, see “Using Select Lists” on page 234.

For more information on the `request` object, see “The request Object” on page 249.

If you want to process data on the client first, you have to create a client-side JavaScript function to perform processing on the form-element values and then assign the output of the client function to a form element. You can hide the element, so that it is not displayed to the user, if you want to perform client preprocessing.

For example, suppose you have a client-side JavaScript function named `calc` that performs calculations based on the user’s input. You want to pass the result of this function to your application for further processing. You first need to define a hidden form element for the result, as follows:

```
<INPUT TYPE="hidden" NAME="result" SIZE=5>
```

Then you need to create an `onClick` event handler for the Submit button that assigns the output of the function to the hidden element:

```
<INPUT TYPE="submit" VALUE="Submit"
  onClick="this.form.result.value=calc(this.form)">
```

The value of `result` is submitted along with any other form-element values. This value can be referenced as `request.result` in the application.

## Using Select Lists

The HTML `SELECT` tag, used with the `MULTIPLE` attribute, allows you to associate multiple values with a single form element. If your application requires select lists that allow multiple selected options, you use the `getOptionValue` function to get the values in JavaScript. The syntax of `getOptionValue` is

```
itemValue = getOptionValue(name, index)
```

Here, `name` is the string specified as the `NAME` attribute of the `SELECT` tag, and `index` is the zero-based ordinal index of the selected option. The `getOptionValue` function returns the value of the selected item, as specified by the associated `OPTION` tag.

The function `getOptionValueCount` returns the number of options (specified by `OPTION` tags) in the select list. It requires only one argument, the string containing the name of the `SELECT` tag.

For example, suppose you have the following element in a form:

```
<SELECT NAME="what-to-wear" MULTIPLE SIZE=8>
  <OPTION SELECTED>Jeans
  <OPTION>Wool Sweater
  <OPTION SELECTED>Sweatshirt
  <OPTION SELECTED>Socks
  <OPTION>Leather Jacket
  <OPTION>Boots
  <OPTION>Running Shoes
  <OPTION>Cape
</SELECT>
```

You could process the input from this select list as follows:

```
<SERVER>
var i = 0;
var howmany = getOptionValueCount("what-to-wear");
while ( i < howmany ) {
  var optionValue =
    getOptionValue("what-to-wear", i);
  write ("<br>Item #" + i + ": " + optionValue + "\n");
  i++;
}
</SERVER>
```

If the user kept the default selections, this script would return:

```
Item #0: Jeans
Item #1: Sweatshirt
Item #2: Socks
```

## Encoding Information in a URL

You can manually encode properties of the `request` object into a URL that accesses a page of your application. In creating the URL, you use the following syntax:

```
URL?varName1=value1[&varName2=value2...]
```

Here, `URL` is the base URL, each `varNameN` is a property name, and each `valueN` is the corresponding property value (with special characters escaped). In this scheme, the base URL is followed by a question mark (?) which is in turn followed by pairs of property names and their values. Separate each pair

with an ampersand (&). When the runtime engine on the server receives the resultant URL as a client request, it creates a `request` property named `varNameN` for each listed variable.

For example, the following HTML defines a hyperlink to a page that instantiates the `request` properties `i` and `j` to 1 and 2, respectively. JavaScript statements in `refpage.html` can then refer to these variables as `request.i` and `request.j`.

```
<A HREF="refpage.html?i=1&j=2">Click Here</A>
```

Instead of using a static URL string, as in the preceding example, you can use server-side or client-side JavaScript statements to dynamically generate the URL that encodes the property values. For example, your application could include a page such as the following:

```
<HTML>
<HEAD>
<SCRIPT>
function compute () {
    // ...replace with an appropriate computation
    // that returns a search string ...
    return "?num=25";
}
</SCRIPT>
</HEAD>

<BODY>
<a HREF="refpage.htm" onClick="this.search=compute()">
Click here to submit a value.</a></p>

</BODY>
</HTML>
```

In this case, when the user clicks the link, the runtime engine on the client runs the `onClick` event handler. This event handler sets the search portion of the URL in the link to whatever string is returned by the `compute` function. When the runtime engine on the server gets this request, it creates a `num` property for the `request` object and sets the value to 25.

As a second example, you might want to add `request` properties to a URL created in a server-side script. This is most likely to be useful if you'll be redirecting the client request to a new page. To add `request` properties in a server-side script, you could instead use this statement:

```
<A HREF=' "refpage.html?i=" + escape(i) + "&j=" + escape(j) ' >
Click Here</A>
```

If you create a URL in a server-side JavaScript statement, the `client` object's properties are not automatically added. If you're using a URL-based maintenance technique for the `client` object, use the `addClient` function to generate the final URL. In this example, the statement would be:

```
<A HREF='addClient("refpage.html?i=" + escape(i)
+ "&j=" + escape(j))'>Click Here</A>
```

For information on using `addClient`, see “Manually Appending client Properties to URLs” on page 277.

The core JavaScript `escape` function allows you to encode names or values appended to a URL that may include special characters. In general, if an application needs to generate its own property names and values in a URL request, you should use `escape`, to ensure that all values are interpreted properly. For more information, see the *Server-Side JavaScript Reference*.

Remember that a URL does not change when a user reloads it, although the page's contents may change. Any properties sent in the original URL are restored to their values in the URL as it was first sent, regardless of any changes that may have been made during processing. For example, if the user clicks the Reload button to reload the URL in the previous example, `i` and `j` are again set to 1 and 2, respectively.

## Sending Values from Server to Client

A JavaScript application communicates with the client through HTML and client-side JavaScript. If you simply want to display information to the user, there is no subtlety: you create the HTML to format the information as you want it displayed.

However, you may want to send values to client scripts directly. You can do this in a variety of ways, including these three:

- You can set default form values and values for hidden form elements. (See “Default Form Values and Hidden Form Elements” on page 238.)
- You can directly substitute information in client-side `SCRIPT` statements or event handlers. (See “Direct Substitution” on page 239.)
- You can use cookies to send `client` property values or other values to the client. (See “Using Cookies” on page 239.)

- You can modify the header of the response sent to the client, using the `deleteResponseHeader` and `addResponseHeader` functions. (See “Request and Response Manipulation” on page 302.)
- You can use LiveConnect with CORBA services to communicate between client and server. (See Chapter 22, “Accessing CORBA Services.”)

## Default Form Values and Hidden Form Elements

To display an HTML form with default values set in the form elements, use the `INPUT` tag to create the desired form element, substituting a server-side JavaScript expression for the `VALUE` attribute. For example, you can use the following statement to display a text element and set the default value based on the value of `client.custname`:

```
<INPUT TYPE="text" NAME="customerName" SIZE="30"
      VALUE='client.custname'>
```

The initial value of this text field is set to the value of the variable `client.custname`. So, if the value of `client.custname` is Victoria, this statement is sent to the client:

```
<INPUT TYPE="text" NAME="customerName" SIZE="30" VALUE="Victoria">
```

You can use a similar technique with hidden form elements if you do not want to display the value to the user, as in this example:

```
<INPUT TYPE="hidden" NAME="custID" SIZE=5 VALUE='client.custID'>
```

In both cases, you can use these values in client-side JavaScript in property values of objects available on the client. If these two elements are in a form named `entryForm`, then these values become the JavaScript properties `document.entryForm.customerName` and `document.entryForm.custID`, respectively. You can then perform client processing on these values in client-side scripts. For more information, see the *Client-Side JavaScript Guide*.

## Direct Substitution

You can also use server-side JavaScript to generate client-side scripts. These values can be used in subsequent statements on the client. As a simple example, you could initialize a client-side variable named `budget` based on the value of `client.amount` as follows:

```
<p>The budget is:
<SCRIPT>
<SERVER>
write("var budget = " + client.amount);
</SERVER>
document.write(budget);
</SCRIPT>
```

If the value of `client.amount` is 50, this would generate the following JavaScript:

```
<p>The budget is:
<SCRIPT>
var budget = 50
document.write(budget);
</SCRIPT>
```

When run on the client, this appears as follows:

```
The budget is: 50
```

## Using Cookies

Cookies are a mechanism you can use on the client to maintain information between requests. This information resides in a file called `cookie.txt` (the cookie file) stored on the client machine. The Netscape cookie protocol is described in detail in the *Client-Side JavaScript Guide*.

You can use cookies to send information in both directions, from the client to the server and from the server to the client. Cookies you send from the client become properties of either the `client` object or of the `request` object. Although you can send any string value to the client from the server as a cookie, the simplest method involves sending `client` object properties.

## Properties of the client Object as Cookies

If an application uses the client cookie technique to maintain the `client` object, the runtime engine on the server stores the names and values of properties of the `client` object as cookies on the client. For information on using cookies to maintain the `client` object, see “Techniques for Maintaining the client Object” on page 263.

For a `client` property called `propName`, the runtime engine automatically creates a cookie named `NETSCAPE_LIVEWIRE.propName`, assuming the application uses the client cookie maintenance technique. The runtime engine encodes property values as required by the Netscape cookie protocol.

To access these cookies in a client-side JavaScript script, you can extract the information using the `document.cookie` property and a function such as the `getSSCookie` function shown here:

```
function getSSCookie(name) {
    var search = "NETSCAPE_LIVEWIRE." + name + "=";
    var retstr = "";
    var offset = 0;
    var end = 0;
    if (document.cookie.length > 0) {
        offset = document.cookie.indexOf(search);
        if (offset != -1) {
            offset += search.length;
            end = document.cookie.indexOf(";", offset);
            if (end == -1)
                end = document.cookie.length;
            retstr = unescape(document.cookie.substring(offset, end));
        }
    }
    return(retstr)
}
```

The `getSSCookie` function is not a predefined JavaScript function. If you need similar functionality, you must define it for your application.

To send information to the server to become a property of the `client` object, add a cookie whose name is of the form `NETSCAPE_LIVEWIRE.propName`. Assuming your application uses the client cookie maintenance technique, the runtime engine on the server creates a `client` property named `propName` for this cookie.



To do so, you can use a function such as the following:

```
function setSSCookie (name, value, expire) {
    document.cookie =
        "NETSCAPE_LIVEWIRE." + name + "="
        + escape(value)
        + ((expire == null) ? "" : ("; expires=" + expire.toGMTString()));
}
```

Here, too, the `setSSCookie` function is not a predefined JavaScript function. If you need similar functionality, you must define it for your application.

You can call these functions in client-side JavaScript to get and set property values for the `client` object, as in the following example:

```
var value = getSSCookie ("answer");
if (value == "") {
    var expires = new Date();
    expires.setDate(expires.getDate() + 7);
    setSSCookie ("answer", "42", Expires);
}
else
    document.write ("The answer is ", value);
```

This group of statements checks whether there is a `client` property called `answer`. If not, the code creates it and sets its value to 42; if so, it displays its value.

## Other Cookies

When a request is sent to the server for a page in a JavaScript application, the header of the request includes all cookies currently set for the application. You can use the `request.httpHeader` method to access these cookies from server-side JavaScript and assign them to server-side variables. Conversely, you can use the `addResponseHeader` function to add new cookies to the response sent back to the client. This functionality is described in “Request and Response Manipulation” on page 302.

On the client, you can use a function such as the following to access a particular cookie:

```
function GetCookie (name) {
    var arg = name + "=";
    var alen = arg.length;
    var clen = document.cookie.length;
    var i = 0;
    while (i < clen) {
```

```

    var j = i + alen;
    if (document.cookie.substring(i, j) == arg) {
        var end = document.cookie.indexOf(";", j);
        if (end == -1)
            end = document.cookie.length;
        return unescape(document.cookie.substring(j, end));
    }
    i = document.cookie.indexOf(" ", i) + 1;
    if (i == 0) break;
}
return null;
}

```

And you can use a function such as the following to set a cookie on the client:

```

function setCookie (name, value, expires, path, domain, secure) {
    document.cookie =
        name + "="
        + escape(value)
        + ((expires) ? "; expires=" + expires.toGMTString() : "")
        + ((path) ? "; path=" + path : "")
        + ((domain) ? "; domain=" + domain : "")
        + ((secure) ? "; secure" : "");
}

```

If the path you specify for a cookie is in your JavaScript application, then that cookie will be sent in any request sent to the application.

You can use this technique for passing cookie information between the client and the server regardless of the `client` object maintenance technique you use.

## Garbage Collection

Server-side JavaScript contains a garbage collector that automatically frees memory allocated to objects no longer in use. Most users do not need to understand the details of the garbage collector. This section gives an overview of the garbage collector and information on when it is invoked.

**Important** This section provides advanced users with a peek into the internal workings of server-side JavaScript. Netscape does not guarantee that these algorithms will remain the same in future releases.

The JavaScript object space consists of arenas. That is, the JavaScript runtime engine allocates a set of arenas from which it allocates objects. When the runtime engine receives a request for a new object, it first looks on the free list.

If the free list has available space, the engine allocates that space. Otherwise, the runtime engine allocates space from the arena currently in use. If all arenas are in use, the runtime engine allocates a new arena. When all the objects from an arena are garbage, the garbage collector frees the arena.

A JavaScript string is typically allocated as a GC object. The string has a reference to the bytes of the string which are also allocated in the process heap. When a string object is garbage collected, the string's bytes are freed.

The JavaScript garbage collector is based on mark and sweep. It does not relocate objects. The garbage collector maintains a root set of objects at all times. This root set includes the JavaScript stack, the global object for the JavaScript context, and any JavaScript objects which have been explicitly added to the root set. During the mark phase, the garbage collector marks all objects that are reachable from the root set. At the end of this phase, all unmarked objects are garbage. All garbage objects are collected into a free list.

A garbage collection is considered necessary if the number of bytes currently in use is 1.5 times the number of bytes that were in use at the end of the last garbage collection. The runtime engine checks for this condition at the following points and starts the garbage collector if it needs to:

- At the end of every request.
- During a long JavaScript computation after a predetermined number of JavaScript bytecode operations have executed, and only when a branch operation is executed. If you have code with no branch operations, garbage collection won't occur simply because a predetermined number of operations have executed. (A branch operation can be an `if` statement, `while` statement, function call, and so on.)
- When an attempt is made to allocate a new JavaScript object but JavaScript has no available memory and no additional memory can be obtained from the operation system.
- When the `lw_ForceGarbageCollection` function is called.



# Session Management Service

This chapter describes the Session Management Service objects available in server-side JavaScript for sharing data among multiple client requests to an application, among multiple users of a single application, or even among multiple applications on a server.

This chapter contains the following sections:

- Overview of the Predefined Objects
- The request Object
- The client Object
- The project Object
- The server Object
- Techniques for Maintaining the client Object
- Sharing Objects Safely with Locking

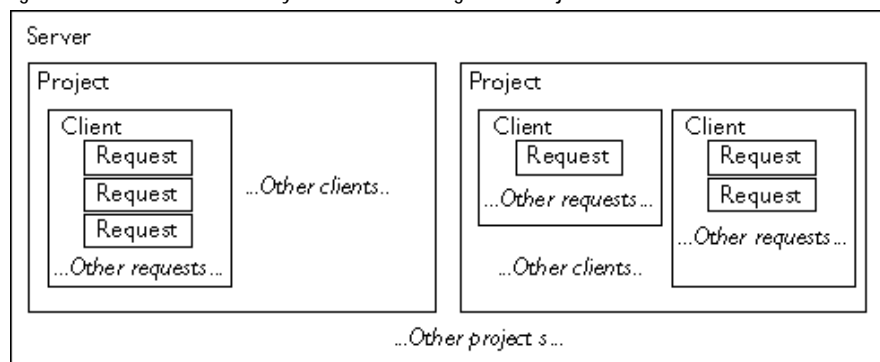
The Session Management Service is a set of capabilities that control the construction and destruction of various predefined objects during any server session. These capabilities are provided in the predefined objects `request`, `client`, `project`, and `server`.

In addition, you can construct instances of `Lock` to control access during the sharing of information. `Lock` instances provide you with fine-grained control over information sharing by getting exclusive access to specified objects.

# Overview of the Predefined Objects

The predefined `request`, `client`, `project`, and `server` objects contain data that persists for different periods and is available to different clients and applications. There is one `server` object shared by all running applications on the server. There is a separate `project` object for each running application. There is one `client` object for each browser (client) accessing a particular application. Finally, there is a separate `request` object for each client request from a particular client to a particular application. Figure 13.1 illustrates the relative availability of the different objects.

Figure 13.1 Relative availability of session-management objects



The JavaScript runtime engine on the server constructs session-management objects at different times. These objects are useful for storing a variety of data. You can define application-specific properties for any of these objects.

- `request` object

Contains data available only to the current client request. Nothing else shares this object. The `request` object has predefined properties you can access.

Treat the `request` object almost as a read-only object. The runtime engine stores the current value of all form elements as properties of the `request` object. You can use it to store information specific to a single request, but it is more efficient to use JavaScript variables for this purpose.

The runtime engine constructs a `request` object each time the server responds to a client request from the web browser. It destroys the object at the end of the client request. The runtime engine does not save `request` data at all.

For more details, see “The request Object” on page 249.

- `client` object

Contains data available only to an individual client/application pair. If a single client is connected to two different applications at the same time, the JavaScript runtime engine constructs a separate `client` object for each client/application pair. All requests from a single client to the same application share the same `client` object. The `client` object has no predefined properties.

In general, use the `client` object for data that should be shared across multiple requests from the same client (user) but that should not be shared across multiple clients of the application. For example, you can store a user’s customer ID as a property of the `client` object.

The runtime engine physically constructs the `client` object for each client request, but properties persist across the lifetime of the client’s connection to the application. Therefore, although the physical `client` object exists only for a single client request, conceptually you can think of it as being constructed when the client is first connected to the application, and not destroyed until the client stops accessing the application. There are several approaches to maintaining the properties of the `client` object across multiple requests. For more information, see “Techniques for Maintaining the client Object” on page 263.

The runtime engine destroys the `client` object when the client has finished using the application. In practice, it is tricky for the JavaScript runtime engine to determine when the `client` object and its properties should be destroyed. For information on how it makes this determination, see “The Lifetime of the client Object” on page 275. Also, see “The client Object” on page 252.

- `project` object

Contains data that is available to all clients accessing any part of the application. All clients accessing the same application share the same `project` object. The `project` object has no predefined properties.

In general, use the `project` object to share data among multiple clients accessing the same application. For example, you can store the next available customer ID as a property of the `project` object. When you use the `project` object to share data, you need to be careful about simultaneous access to that data; see “Sharing Objects Safely with Locking” on page 279. Because of limitations on the `client` object’s properties, you sometimes use the `project` object to store data for a single client.

The runtime engine constructs the `project` object when the application is started by the Application Manager or when the server is started. It destroys the object when the application or the server is stopped.

For more details, see “The project Object” on page 260.

- `server` object

Contains data available to all clients and all applications for the entire server. All applications and all client/application pairs share the same `server` object. The `server` object has predefined properties you can access.

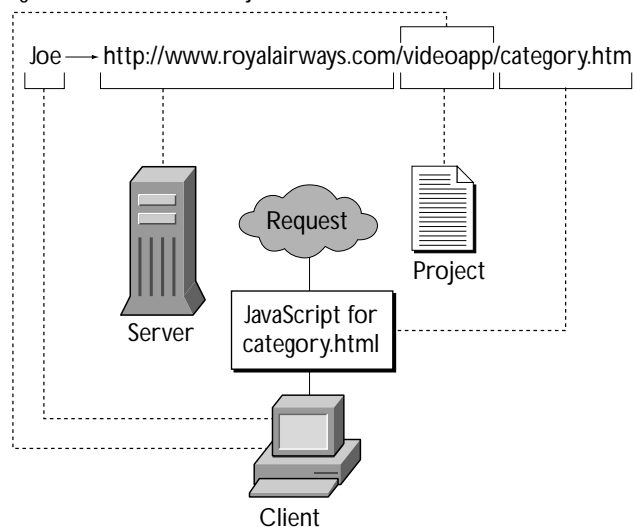
Use the `server` object to share data among multiple applications on the server. For example, you might use the `server` object to track usage of all of the applications on your server. When you use the `server` object to share data, you need to be careful about simultaneous access to that data; see “Sharing Objects Safely with Locking” on page 279.

The runtime engine constructs the `server` object when the server is started and destroys the object when the server is stopped.

For more details, see “The server Object” on page 261.

It may help to think about how these objects correspond to a URL for a page in your application. Consider Figure 13.2.

Figure 13.2 Predefined objects in a URL





In this illustration, Joe requests the URL `http://www.royalairways.com/videoapp/category.html`, corresponding to a page in the `videoapp` sample application. When the runtime engine receives the request, it uses the already-existing `server` object corresponding to `www.royalairways.com` and the already-existing `project` object corresponding to the `videoapp` application. The engine creates a `client` object corresponding to the combination of Joe and the `videoapp` application. If Joe has already accessed other pages of this application, this new `client` object uses any stored properties. Finally, the engine creates a new `request` object for the specific request for the `category.html` page.

## The request Object

The `request` object contains data specific to the current client request. It has the shortest lifetime of any of the objects. JavaScript constructs a new `request` object for each client request it receives; for example, it creates an object when

- A user manually requests a URL by typing it in or choosing a bookmark.
- A user clicks a hyperlink or otherwise requests a document that refers to another page.
- Client-side JavaScript sets the property `document.location` or navigates to the page using the `history` method.
- Server-side JavaScript calls the `redirect` function.

The JavaScript runtime engine on the server destroys the `request` object when it finishes responding to the request (typically by providing the requested page). Therefore, the typical lifetime of a `request` object can be less than one second.

**Note** You cannot use the `request` object on your application's initial page. This page is run when the application is started on the server. At this time, there is no client request, and so there is no available `request` object. For more information on initial pages, see “Installing a New Application” on page 61.

For summary information on the `request` object, see “Overview of the Predefined Objects” on page 246.

# Properties

The following table lists the predefined properties of the `request` object. Several of these predefined properties correspond to CGI environment variables. You can also access these and other CGI environment variables using the `ssjs_getCGIVariable` function described in “Accessing CGI Variables” on page 228.

Table 13.1 Properties of the `request` object

Property	Description	Example value
<code>agent</code>	Name and version of the client software. Use this information to conditionally employ advanced features of certain browsers.	<code>Mozilla/1.1N (Windows; I; 32bit)</code>
<code>auth_type</code>	The authorization type, if this request is protected by any type of authorization. Netscape web servers support HTTP basic access authorization. Corresponds to the CGI <code>AUTH_TYPE</code> environment variable.	<code>basic</code>
<code>auth_user</code>	The name of the local HTTP user of the web browser, if HTTP access authorization has been activated for this URL. Note that this is not a way to determine the user name of any person accessing your program. Corresponds to the CGI <code>REMOTE_USER</code> environment variable.	<code>vpg</code>
<code>ip</code>	The IP address of the client. May be useful to authorize or record access.	<code>198.95.251.30</code>
<code>method</code>	The HTTP method associated with the request. An application can use this to determine the proper response to a request. Corresponds to the CGI <code>REQUEST_METHOD</code> environment variable.	<code>GET<sup>a</sup></code>
<code>protocol</code>	The HTTP protocol level supported by the client's software. Corresponds to the CGI <code>SERVER_PROTOCOL</code> environment variable.	<code>HTTP/1.0</code>
<code>query</code>	Information from the requesting HTML page; this is information in the URL that comes after the “?”. Corresponds to the CGI <code>QUERY_STRING</code> environment variable.	<code>button1=on&amp;button2=off</code>

Table 13.1 Properties of the `request` object (Continued)

Property	Description	Example value
<code>imageX</code>	The horizontal position of the cursor when the user clicked over an image map. Described in “Working with Image Maps” on page 252.	45
<code>imageY</code>	The vertical position of the cursor when the user clicked over an image map. Described in “Working with Image Maps” on page 252.	132
<code>uri</code>	The request’s partial URL, with the protocol, host name, and the optional port number stripped out.	<code>videoapp/add.html</code>

a. For HTTP 1.0, `method` is one of `GET`, `POST`, or `HEAD`.

When you declare top-level variables in server-side JavaScript, they have the same lifetime as `request` properties. For example, this declaration persists during the current request only:

```
var number = 42;
```

In addition to the predefined properties, you can, in your client code, have information that will become properties of the `request` object. You do so by using form elements and by encoding properties into the request URL, as described “Sending Values from Client to Server” on page 232.

Although you can also create additional properties for `request` directly in server-side JavaScript statements, performance may be better if you instead use JavaScript variables. The properties of the `request` object you create can be of any legal JavaScript type, including references to other JavaScript objects.

Remember that the lifetime of the `request` object and hence of its properties is the duration of the request. If you store a reference to another object in the `request` object, the referenced object is destroyed at the end of the request along with the `request` object, unless the referenced object has other live references to it, directly or indirectly from the `server` or `project` object.

## Working with Image Maps

The `ISMAP` attribute of the `IMG` tag indicates a server-based image map. If the user clicks on an image map, the horizontal and vertical positions of the cursor are sent to the server. The `imageX` and `imageY` properties return these horizontal and vertical positions. Consider this HTML:

```
<A HREF="mapchoice.html">
<IMG SRC="images\map.gif" HEIGHT=599 WIDTH=424 BORDER=0
      ISMAP ALT="SANTA CRUZ COUNTY">
</A>
```

The page `mapchoice.html` has properties `request.imageX` and `request.imageY` based on the cursor position at the time the user clicked.

## The client Object

Many browser clients can access a JavaScript application simultaneously. The `client` object provides a method for dealing with each browser client individually. It also provides a technique for tracking each browser client's progress through an application across multiple requests.

The JavaScript runtime engine on the server constructs a `client` object for every client/application pair. A browser client connected to one application has a different `client` object from the same browser client connected to a different application. The runtime engine constructs a new `client` object each time a user accesses an application; there can be hundreds or thousands of `client` objects active at the same time.

**Note** You cannot use the `client` object on your application's initial page. This page is run when the application is started on the server. At this time, there is no client request, and so there is no available `client` object. For more information on initial pages, see "Installing a New Application" on page 61.

The runtime engine constructs and destroys the `client` object for each client request. However, while processing a request, the runtime engine saves the names and values of the `client` object's properties. In this way, the runtime engine can construct a new `client` object from the saved data when the same user returns to the application with a subsequent request. Thus, conceptually you can think of the `client` object as remaining for the duration of a client's session with the application.

JavaScript does not save `client` objects that have no property values. Therefore, if an application does not need `client` objects and does not assign any `client` object property values, it incurs no additional overhead.

You have several options for how and where the runtime engine saves `client` object properties. These options are discussed in “Techniques for Maintaining the client Object” on page 263.

For summary information on the `client` object, see “Overview of the Predefined Objects” on page 246.

## Properties

There are no predefined property values in the `client` object, because it is intended to contain data specific to the application. JavaScript statements can assign application-specific properties and values to the `client` object. A good example of a `client` object property is a customer ID number. When the user first accesses the application, the application might assign a customer ID, as in the following example:

```
client.custID = getNextCustID();
```

This example uses the application-defined `getNextCustID` function to compute a customer ID. The runtime engine then assigns this ID to the `client` object's `custID` property.

Once the customer ID has been established, it would be inconvenient to require the user to reenter the ID on each page of the application. However, without the `client` object, there would be no way to associate the correct customer ID with subsequent requests from a client.

Because of the techniques used to maintain `client` properties across multiple client requests, there is one major restriction on `client` property values. The JavaScript runtime engine on the server converts the values of all of the `client` object's properties to strings.

Do not assign an object as the value of a `client` property. If you do so, the runtime engine converts that object to a string; once this happens, you won't be able to work with it as an object anymore. If a client property value represents

another data type, such as a number, you must convert the value from a string before using it. For example, you can create an integer `client` property as follows:

```
client.totalNumber = 17;
```

You could then use `parseInt` to increment the value of `totalNumber` as follows:

```
client.totalNumber = parseInt(client.totalNumber) + 1;
```

Similarly, you can create a Boolean `client` property as follows:

```
client.bool = true;
```

Then you can check it as follows:

```
if (client.bool == "true")
    write("It's true!");
else
    write("It's false!");
```

Notice that the conditional expression compares `client.bool` to the string `"true"`. You can use other techniques to handle Boolean expressions. For example, to negate a Boolean property, you can use code like this:

```
client.bool = (client.bool == "true") ? false : true;
```

Although you can work with `client` properties directly, you incur some overhead doing so. If you repeatedly use the value of a `client` property, consider using top-level JavaScript variables. Before using the `client` property, assign it to a variable. When you have finished working with that variable, assign the resulting value back to the appropriate `client` property. This technique can result in a substantial performance improvement.

As noted previously, you cannot store references to other objects in the `client` object. You can, however, store object references in either the `project` or the `server` object. If you want a property associated with the client to have object values, create an array indexed by client ID and store a reference to the array in the `project` or `server` object. You can use that array to store object values associated with the client. Consider the following code:

```
if client.id == null
    client.id = ssjs_generateClientID();
project.clientDates[client.id] = new Date();
```

This code uses the `ssjs_generateClientID` function, described next, to create a unique ID for this `client` object. It uses that ID as an index into the `clientDates` array on the `project` object and stores a new `Date` object in that array associated with the current `client` object.

## Uniquely Referring to the client Object

For some applications, you may want to store information specific to a client/application pair in the `project` or `server` objects. Two common cases are storing a database connection between client requests (described in Chapter 15, “Connecting to a Database”) or storing a custom object that has the same lifetime as the predefined `client` object and that contains object values (described in “Creating a Custom client Object” on page 256).

In these situations, you need a way to refer uniquely to the client/application pair. JavaScript provides two functions for this purpose, `ssjs_getClientID` and `ssjs_generateClientID`. Neither function takes any arguments; both return a unique string you can use to identify the pair.

Each time you call `ssjs_generateClientID`, the runtime engine returns a new identifier. For this reason, if you use this function and want the identifier to last longer than a single client request, you need to store the identifier, possibly as a property of the `client` object. For an example of using this function, see “Sharing an Array of Connection Pools” on page 321.

If you use this function and store the ID in the `client` object, you may need to be careful that an intruder cannot get access to that ID and hence to sensitive information.

An alternative approach is to use the `ssjs_getClientID` function. If you use one of the server-side maintenance techniques for the `client` object, the JavaScript runtime engine generates and uses a identifier to access the information for a particular client/application pair. (For information on maintaining the `client` object, see “Techniques for Maintaining the client Object” on page 263.)

When you use these maintenance techniques, `ssjs_getClientID` returns the identifier used by the runtime engine. Every time you call this function from a particular client/application pair, you get the same identifier. Therefore, you do not need to store the identifier returned by `ssjs_getClientID`. However, if

you use any of the other maintenance techniques, this function returns “undefined”; if you use those techniques you must instead use the `ssjs_generateClientID` function.

If you need an identifier and you’re using a server-side maintenance technique, you probably should use the `ssjs_getClientID` function. If you use this function, you do not need to store and track the identifier yourself; the runtime engine does it for you. However, if you use a client-side maintenance technique, you cannot use the `ssjs_getClientID` function; you must use the `ssjs_generateClientID` function.

## Creating a Custom client Object

As discussed in earlier sections, properties of the predefined `client` object can have only string values. This restriction can be problematic for some applications. For instance, your application may require an object that persists for the same lifetime as the predefined `client` object, but that can have objects or other data types as property values. In this case, you can create your own object and store it as a property of the `client` object.

This section provides an example of creating such an object. You can include the code in this section as a JavaScript file in your application. Then, at the beginning of pages that need to use this object, include the following statement:

```
var customClient = getCustomClient()
```

(Of course, you can use a different variable name.) If this is the first page that requests the object, `getCustomClient` creates a new object. On other pages, it returns the existing object.

This code stores an array of all the custom `client` objects defined for an application as the value of the `customClients` property of the predefined `project` object. It stores the index in this array as a string value of the `customClientID` property of the predefined `client` object. In addition, the code uses a lock stored in the `customClientLock` property of `project` to ensure safe access to that array. For information on locking, see “Sharing Objects Safely with Locking” on page 279.

The `timeout` variable in the `getCustomClient` function hard-codes the expiration period for this object. If you want a different expiration time, specify a different value for that variable. Whatever expiration time you use, you



should call the predefined `client` object's `expiration` method to set its expiration to the same time as specified for your custom object. For information on this method, see “The Lifetime of the client Object” on page 275.

To remove all expired custom client objects for the application, call the following function:

```
expireCustomClients()
```

That's all there is to it! If you use this code, the predefined `client` and `project` objects have these additional properties that you should not change:

- `client.customClientID`
- `project.customClients`
- `project.customClientLock`

You can customize the custom class by changing its `onInit` and `onDestroy` methods. As shown here, those methods are just stubs. You can add code to modify what happens when the object is created or destroyed.

Here's the code:

```
// This function creates a new custom client object or retrieves
// an existing one.
function getCustomClient()
{
    // =====> Change the hardcoded hold period <=====
    // Note: Be sure to set the client state maintenance
    // expiration to the same value you use below by calling
    // client.expiration. That allows the index held in the predefined
    // client object to expire about the same time as the state held in
    // the project object.
    var timeout = 600;
    var customClient = null;
    var deathRow = null;
    var newObjectWasCreated = false;

    var customClientLock = getCustomClientLock();
    customClientLock.lock();
    var customClientID = client.customClientID;
    if ( customClientID == null ) {
        customClient = new CustomClient(timeout);
        newObjectWasCreated = true;
    }
    else {
        var customClients = getCustomClients();
        customClient = customClients[customClientID];
        if ( customClient == null ) {
            customClient = new CustomClient(timeout);
        }
    }
}
```

```

        newObjectWasCreated = true;
    }
    else {
        var now = (new Date()).getTime();
        if ( customClient.expiration <= now ) {
            delete customClients[customClientID];
            deathRow = customClient;

            customClient = new CustomClient(timeout);
            newObjectWasCreated = true;
        }
        else {
            customClient.expiration = (new Date()).getTime() +
                timeout*1000;
        }
    }
}
if ( newObjectWasCreated )
    customClient.onInit();
customClientLock.unlock();

if ( deathRow != null )
    deathRow.onDestroy();
return customClient;
}

// Function to remove old custom client objects.
function expireCustomClients()
{
    var customClients = getCustomClients();
    var now = (new Date()).getTime();
    for ( var i in customClients ) {
        var clientObj = customClients[i];
        if ( clientObj.expiration <= now ) {
            var customClientLock = getCustomClientLock();
            customClientLock.lock();
            if ( clientObj.expiration <= now ) {
                delete customClients[i];
            }
            else {
                clientObj = null;
            }
            customClientLock.unlock()
            if ( clientObj != null )
                clientObj.onDestroy();
        } } }

// Don't call this function directly.
// It's used by getCustomClient and expireCustomClients.
function getCustomClientLock()
{
    if ( project.customClientLock == null ) {

```

```

        project.lock()
        if ( project.customClientLock == null )
            project.customClientLock = new Lock()
        project.unlock()
    }
    return project.customClientLock
}

// Don't call this function directly.
// It's used by getCustomClient and expireCustomClients.
function getCustomClients()
{
    if ( project.customClients == null ) {
        project.lock()
        if ( project.customClients == null )
            project.customClients = new Object()
        project.unlock()
    }
    return project.customClients
}

// The constructor for the CustomClient class. Don't call this directly.
// Instead use the getCustomClient function.
function CustomClient(seconds)
{
    var customClients = getCustomClients();
    var customClientID = ssjs_generateClientID();

    this.onInit = CustomClientMethod_onInit;
    this.onDestroy = CustomClientMethod_onDestroy;
    this.expiration = (new Date()).getTime() + seconds*1000;

    client.customClientID = customClientID;

    customClients[customClientID] = this;
}

// If you want to customize, do so by redefining the next 2 functions.
function CustomClientMethod_onInit()
{
    // =====> Add your object initialization code <=====
    // This method is called while in a lock, so keep it quick!
}

function CustomClientMethod_onDestroy()
{
    // =====> Add your object cleanup code <=====
    // This method is not called from within a lock.
}

```

# The project Object

The `project` object contains global data for an application and provides a method for sharing information among the clients accessing the application. JavaScript constructs a new `project` object when an application is started using the Application Manager. Each client accessing the application shares the same `project` object. For summary information on the `project` object, see “Overview of the Predefined Objects” on page 246.

In this release the JavaScript runtime engine does not, as in previous releases, create or destroy the `project` object for each request. When you stop an application, that application's `project` object is destroyed. A new `project` object is created for it when the application is started again. A typical `project` object lifetime is days or weeks.

JavaScript constructs a set of `project` objects for each Netscape HTTP process running on the server. JavaScript constructs a `project` object for each application running on each distinct server. For example, if one server is running on port 80 and another is running on port 142 on the same machine, JavaScript constructs a distinct set of `project` objects for each process.

## Properties

There are no predefined properties for the `project` object, because it is intended to contain application-specific data accessible by multiple clients. You can create properties of any legal JavaScript type, including references to other JavaScript objects. If you store a reference to another object in the `project` object, the runtime engine does not destroy the referenced object at the end of the client request during which it is created. The object is available during subsequent requests.

A good example of a `project` object property is the next available customer ID. An application could use this property to track sequentially assigned customer IDs. Any client that accesses the application without a customer ID would be assigned an ID, and the value would be incremented for each initial access.

Remember that the `project` object exists only as long as the application is running on the server. When the application is stopped, the `project` object is destroyed, along with all of its property values. For this reason, if you have

application data that needs to be stored permanently, you should store it either in a database (see Part 4, “LiveWire Database Service”) or in a file on the server (see “File System Service” on page 290).

## Sharing the project Object

There is one `project` object for each application. Thus, code executing in any request for a given application can access the same `project` object. Because the server is multithreaded, there can be multiple requests active at any given time, either from the same client or from several clients.

To maintain data integrity, you must make sure that you have exclusive access to a property of the `project` object when you change the property’s value. There is no implicit locking as in previous releases; you must request exclusive access. The simplest way to do this is to use the `project` object’s `lock` and `unlock` methods. For details, see “Sharing Objects Safely with Locking” on page 279.

## The server Object

The `server` object contains global data for the entire server and provides a method for sharing information between several applications running on a server. The `server` object is also automatically initialized with information about the server. For summary information on the `server` object, see “Overview of the Predefined Objects” on page 246.

The JavaScript runtime engine constructs a new `server` object when the server is started and destroys the `server` object when the server is stopped. Every application that runs on the server shares the same `server` object.

JavaScript constructs a `server` object for each Netscape HTTPD process (server) running on a machine. For example, there might be a server process running for port 80 and another for port 8080. These are entirely distinct server processes, and JavaScript constructs a `server` object for each.

## Properties

The following table describes the properties of the `server` object.

Table 13.2 Properties of the `server` object

Property	Description	Example
<code>hostname</code>	Full host name of the server, including the port number	<code>www.netscape.com:85</code>
<code>host</code>	Server name, subdomain, and domain name	<code>www.netscape.com</code>
<code>protocol</code>	Communications protocol being used	<code>http:</code>
<code>port</code>	Server port number being used; default is 80 for HTTP	<code>85</code>
<code>jsVersion</code>	Server version and platform	<code>3.0 WindowsNT</code>

For example, you can use the `jsVersion` property to conditionalize features based on the server platform (or version) on which the application is running, as demonstrated here:

```
if (server.jsVersion == "3.0 WindowsNT")
    write ("Application is running on a Windows NT server.");
```

In addition to these automatically initialized properties, you can create properties to store data to be shared by multiple applications. Properties may be of any legal JavaScript type, including references to other JavaScript objects. If you store a reference to another object in the `server` object, the runtime engine does not destroy the referenced object at the end of the request during which it is created. The object is available during subsequent requests.

Like the `project` object, the `server` object has a limited lifetime. When the web server is stopped, the `server` object is destroyed, along with all of its property values. For this reason, if you have application data that needs to be stored permanently, you should store it either in a database (see Part 4, “LiveWire Database Service”) or in a file on the server (see “File System Service” on page 290).

## Sharing the server Object

There is one `server` object for the entire server. Thus, code executing in any request, in any application, can access the same `server` object. Because the server is multithreaded, there can be multiple requests active at any given time. To maintain data integrity, you must make sure that you have exclusive access to the `server` object when you make changes to it.

Also, you must make sure that you have exclusive access to a property of the `server` object when you change the property's value. There is no implicit locking as in previous releases; you must request exclusive access. The simplest way to do this is to use the `server` object's `lock` and `unlock` methods. For details, see “Sharing Objects Safely with Locking” on page 279.

## Techniques for Maintaining the client Object

The `client` object is associated with both a particular application and a particular client. As discussed in “The client Object” on page 252, the runtime engine creates a new `client` object each time a new request comes from the client to the server. However, the intent is to preserve `client` object properties from one request to the next. In order to do so, the runtime engine needs to store `client` properties between requests.

There are two basic approaches for maintaining the properties of the `client` object; you can maintain them either on the client or on the server. The two client-side techniques either store the property names and their values as cookies on the client or store the names and values directly in URLs on the generated HTML page. The three server-side techniques all store the property names and their values in a data structure in server memory, but they differ in the scheme used to index that data structure.

You select the technique to use when you use the JavaScript Application Manager to install or modify the application, as explained in “Installing a New Application” on page 61. This allows you (or the site manager) to change the maintenance technique without recompiling the application. However, the behavior of your application may change depending on the client-maintenance technique in effect, as described in the following sections. Be sure to make clear to your site manager if your application depends on using a particular technique. Otherwise, the manager can change this setting and break your application.

Because some of these techniques involve storing information either in a data structure in server memory or in the cookie file on the client, the JavaScript runtime engine additionally needs to decide when to get rid of those properties. “The Lifetime of the client Object” on page 275 discusses how the runtime engine makes this decision and describes methods you can use to modify its behavior.

## Comparing Client-Maintenance Techniques

Each maintenance technique has its own set of advantages and disadvantages and what is a disadvantage in one situation may be an advantage in another. You should select the technique most appropriate for your application. The individual techniques are described in more detail in subsequent sections; this section gives some general comparisons.

The following table provides a general comparison of the client-side and server-side techniques.

Table 13.3 Comparison of server-side and client-side maintenance techniques

Server-Side	Client-Side
1. No limit on number of properties stored or the space they use.	Limits on properties.
2. Consumes extra server memory between client requests.	Does not consume extra server memory between client requests.
These differences are related. The lack of a limit on the number and size of properties can be either a disadvantage or an advantage. In general, you want to limit the quantity of data for a consumer application available on the Internet so that the memory of your server is not swamped. In this case, you could use a client technique. However, if you have an Intranet application for which you want to store a lot of data, doing so on the server may be acceptable, as the number of expected clients is limited.	
3. Properties are stored in server memory and so are lost when server or application is restarted.	Properties are not stored in server memory and so aren't lost when server is restarted.
If the properties are user preferences, you may want them to remain between server restarts; if they are particular to a single session, you may want them to disappear.	



Table 13.3 Comparison of server-side and client-side maintenance techniques (Continued)

Server-Side	Client-Side
4. Either no increase or a modest increase in network traffic.	Larger increases in network traffic.
Client-side techniques transmit every property name and corresponding value to the client one or more times. This causes a significant increase in network traffic. Because the server-side techniques all store the property names and values on the server, at most they send a generated name to the client to use in identifying the appropriate entry in the server data structure.	

Figure 13.3 and Figure 13.4 show what information is stored for each technique, where it is stored, and what information goes across the network. Figure 13.3 shows this information for the client-side techniques.

Figure 13.3 Client-side techniques

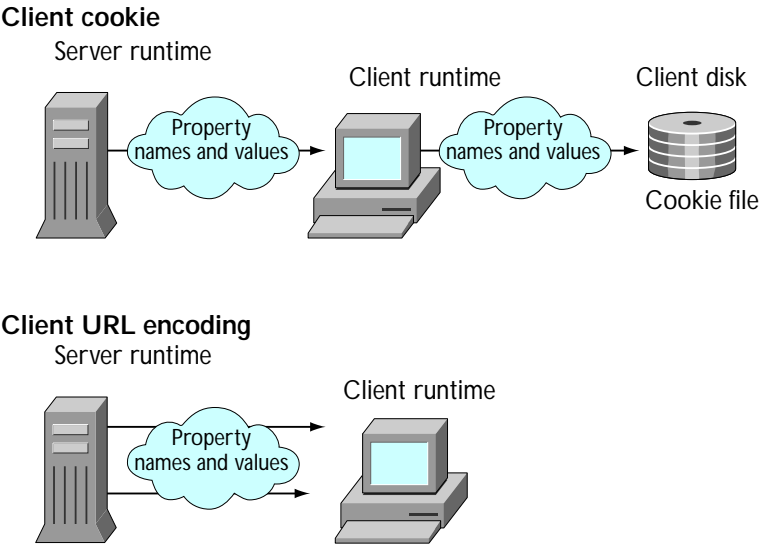
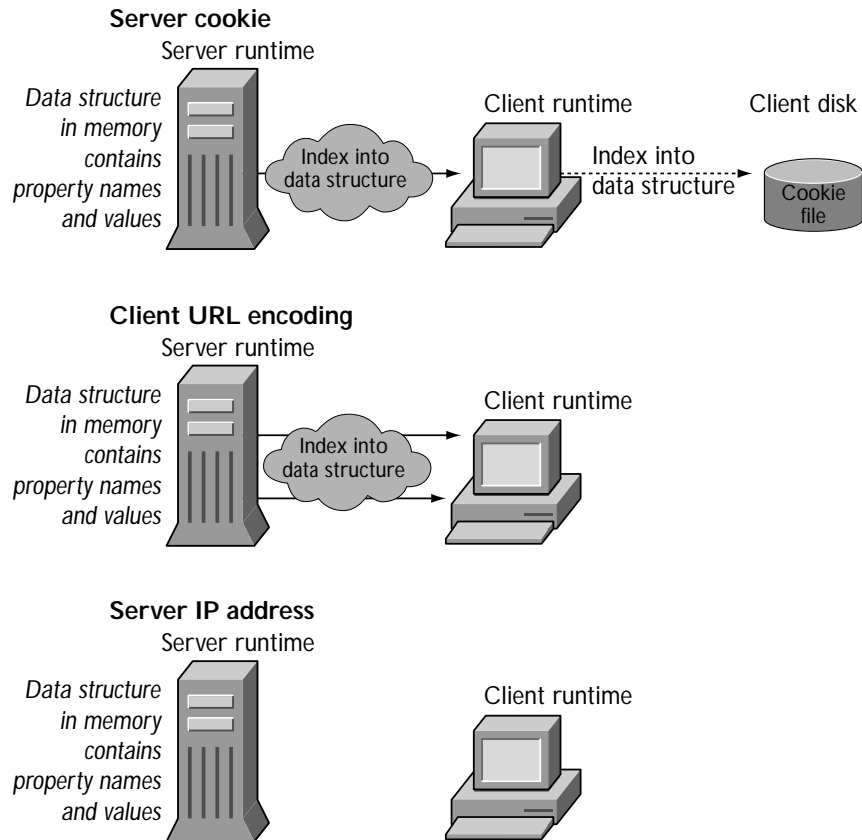


Figure 13.4 shows this information for the server-side techniques.

Figure 13.4 Server-side techniques



There are some other general considerations. For both techniques that use cookies, the browser must support the Netscape cookie protocol. In both cases, when you close your browser on the client machine, information is stored in the client machine's cookie file. The other techniques do not have this restriction.

The server cookie technique creates a single cookie to identify the appropriate `client` object. By contrast, the client cookie technique creates a separate cookie for each property of the `client` object. The client cookie technique is therefore more likely to be affected by the limit of 20 cookies per application.

With the client cookie technique, the `client` object properties are sent to the client when the first piece of the HTML page is sent. If you change `client` property values later in the execution of that page, those changes are not sent to the client and so are lost. This restriction does not apply to any other maintenance technique.

For both techniques that use URL encoding, if your application constructs a URL at runtime or uses the `redirect` function, it must either manually append any `client` properties that need to be saved or use `addClient` to have the runtime engine append the properties. Although appending properties is not required for other techniques, you might want to do it anyway, so that changing the maintenance technique does not break your application.

In addition, for the URL encoding techniques, as soon as the browser accesses any page outside the application, or even submits a form to the application using the `GET` method, all `client` properties are lost. Properties are not lost this way for the other techniques. Your choice of technique should be partially guided by whether or not you want `client` properties to be persist in these situations.

Your choice of maintenance technique rests with the requirements of your application. The client cookie technique does not use extra server memory (as do the server-side techniques) and sends information only once per page (in contrast to the client URL encoding technique). These facts may make the client cookie technique appropriate for high-volume Internet applications. However, there are circumstances under which another technique is more suitable. For example, server IP address is the fastest technique, causing no increase in network traffic. You may use it for a speed-critical application running on your intranet.

## Client-Side Techniques

There are two client-side maintenance techniques:

- Client cookie
- Client URL encoding

For a comparison of all of the maintenance techniques, see “Comparing Client-Maintenance Techniques” on page 264.

When an application uses client-side maintenance techniques, the runtime engine encodes properties of the `client` object into its response to a client request, either in the header of the response (for client cookie) or in URLs in the body of the response (for client URL encoding).

Because the actual property names and values are sent between the client and the server, restarting the server does not cause the client information to be lost. However, sending this information causes an increase of network traffic.

### Using Client Cookie

In the client cookie technique, the JavaScript runtime engine on the server uses the Netscape cookie protocol to transfer the properties of the `client` object and their values to the client. It creates one cookie per `client` property. The properties are sent to the client once, in the response header of the generated HTML page. The Netscape cookie protocol is described in the *Client-Side JavaScript Guide*.

To avoid conflicts with other cookies you might create for your application, the runtime engine creates a cookie name by adding `NETSCAPE_LIVEWIRE.` to the front of the name of a `client` property. For example, if `client` has a property called `custID`, the runtime engine creates a cookie named `NETSCAPE_LIVEWIRE.custID`. When it sends the cookie information to the client, the runtime engine performs any needed encoding of special characters in a property value, as described in the *Client-Side JavaScript Guide*.

Sometimes your application needs to communicate information between its JavaScript statements running on the client and those running on the server. Because this maintenance technique sends `client` object properties as cookies to the client, you can use it as a way to facilitate this communication. For more information, see “Communicating Between Server and Client” on page 232.

With this technique, the runtime engine stores `client` properties the first time it flushes the internal buffer containing the generated HTML page. For this reason, to prevent losing any information, you should assign all `client` property values early in the scripts on each page. In particular, you should ensure that `client` properties are set *before* (1) the runtime engine generates 64KB of content for the HTML page (it automatically flushes the output buffer at this point), (2) you call the `flush` function to clear the output buffer, or (3) you call the `redirect` function to change client requests. For more information, see “Flushing the Output Buffer” on page 226 and “Runtime Processing on the Server” on page 220.

By default, when you use the client cookie technique, the runtime engine does not explicitly set the expiration of the cookies. In this case, the cookies expire when the user exits the browser. (This is the default behavior for all cookies.) As described in “The Lifetime of the client Object” on page 275, you can use the `client` object’s `expiration` method to change this expiration period. If you use `client.expiration`, the runtime engine sets the cookie expiration appropriately in the cookie file.

When using the client cookie technique, `client.destroy` eliminates all `client` property values but does not affect what is stored in the cookie file on the client machine. To remove the cookies from the cookie file or browser memory, do not use `client.destroy`; instead, use `client.expiration` with an argument of 0 seconds.

In general, Netscape cookies have the following limitations. These limitations apply when you use cookies to store `client` properties:

- 4KB for each cookie (including both the cookie’s name and its value). If a single cookie is longer than 4KB, the cookie entry is truncated to 4KB. This may result in an invalid `client` property value.
- 20 cookies for each application. If you create more than 20 cookies for an application, the oldest (first created) cookies are eliminated. Because the client cookie technique creates a separate cookie for each `client` property, the `client` object can store at most 20 properties. If you want to use other cookies in your application as well, the total number of cookies is still limited to 20.
- 300 total cookies in the cookie file. If you create more than 300 cookies, the oldest (first created) cookies are eliminated.

## Using Client URL Encoding

In the client URL encoding technique, the runtime engine on the server transmits the properties of the `client` object and their values to the client by appending them to each URL in the generated HTML page. Consequently, the properties and their values are sent as many times as there are links on the generated HTML page, resulting in the largest increase in network traffic of all of the maintenance techniques.

The size of a URL string is limited to 4KB. Therefore, when you use client URL encoding, the total size of all the property names and their values is somewhat less than 4KB. Any information beyond the 4KB limit is truncated.

If you generate URLs dynamically or use the `redirect` function, you can add `client` properties or other properties to the URL. For this reason, whenever you call `redirect` or generate your own URL, the compiler does not automatically append the `client` properties for you. If you want `client` properties appended, use the `addClient` function. For more information, see “Manually Appending client Properties to URLs” on page 277.

In the client URL encoding technique, property values are added to URLs as those URLs are processed. You need to be careful if you expect your URLs to have the same properties and values. For example, consider the following code:

```
<SERVER>
...
client.numwrites = 2;
write (addClient(
    "<A HREF='page2.htm'>Some link</A>"));
client.numwrites = 3;
write (addClient(
    "<A HREF='page3.htm'>Another link</A>"));
...
</SERVER>
```

When the runtime engine processes the first `write` statement, it uses 2 as the value of the `numwrites` property, but when it processes the second `write` statement, it uses 3 as the value.

Also, if you use the `client.destroy` method in the middle of a page, only those links that come before the method call have property values appended to their URLs. Those that come after the method call do not have any values appended. Therefore, `client` property values are propagated to some pages but not to others. This may be undesirable.

If your page has a link to a URL outside of your application, you may not want the client state appended. In this situation, do not use a static string as the `HREF` value. Instead, compute the value. This prevents the runtime engine from automatically appending the client state to the URL. For example, assume you have this link:

```
<A HREF="mailto:me@royalairways.com">
```

In this case, the runtime engine appends the `client` object properties. To instead have it not do so, use this very similar link:

```
<A HREF=`"mailto:me@royalairways.com"`>
```

In this technique, the `client` object does not expire, because it exists solely in the URL string residing on the client. Therefore, the `client.expiration` method does nothing.

In client URL encoding, you lose all `client` properties when you submit a form using the `GET` method and when you access another application,. Once again, you may or may not want to lose these properties, depending on your application's needs.

In contrast to the client cookie technique, client URL encoding does not require the web browser support the Netscape cookie protocol, nor does it require writing information on the client machine.

## Server-Side Techniques

There are three server-side maintenance techniques:

- IP addresses
- Server cookie
- Server URL encoding

For a comparison of all of the maintenance techniques, see “Comparing Client-Maintenance Techniques” on page 264.

In all of these techniques, the runtime engine on the server stores the properties of the `client` object and their values in a data structure in server memory. A single data structure, preserved between client requests, is used for all applications running on the server. These techniques differ only in the index

used to access the information in that data structure, ensuring that each client/application pair gets the appropriate properties and values for the `client` object.

None of these techniques writes information to the server disk. Only the server cookie technique can cause information to be written to the client machine's disk, when the browser is exited.

Because these techniques store `client` object information in server memory between client requests, there is little or no network traffic increase. The property names and values are never sent to the client. Additionally, there are no restrictions on the number of properties a `client` object can have nor on the size of the individual properties.

The trade-off, of course, is that these techniques consume server memory between client requests. For applications that are accessed by a large number of clients, this memory consumption could become significant. Of course, this can be considered an advantage as well, in that you can store as much information as you need.

## Using IP Address

The IP address technique indexes the data structure based on the application and the client's IP address. This simple technique is also the fastest, because it doesn't require sending any information to the client at all. Since the index is based on both the application and the IP address, this technique does still create a separate index for every application/client pair running on the server.

This technique works well when all clients have fixed IP addresses. It does not work reliably if the client is not guaranteed to have a fixed IP address, for example, if the client uses the Dynamic Host Configuration Protocol (DHCP) or an Internet service provider that dynamically allocates IP addresses. This technique also does not work for clients that use a proxy server, because all users of the proxy report the same IP address. For this reason, this technique is probably useful only for intranet applications.



## Using Server Cookie

The server cookie technique uses a long unique name, generated by the runtime engine, to index the data structure on the server. The runtime engine uses the Netscape cookie protocol to store the generated name as a cookie on the client. It does not store the property names and values as cookies. For this reason, this technique creates a single cookie, whereas the client cookie technique creates a separate cookie for each property of the `client` object.

The generated name is sent to the client once, in the header of the HTML page. You can access this generated name with the `ssjs_getClientID` function, described in “Uniquely Referring to the client Object” on page 255. This technique uses the same cookie file as the client cookie technique; these techniques differ in what information is stored in the cookie file. The Netscape cookie protocol is described in the *Client-Side JavaScript Guide*.

Also, because only the generated name is sent to the client, and not the actual property names and values, it does not matter where in your page you make changes to the `client` object properties. This contrasts with the client cookie technique.

By default, the runtime engine sets the expiration of the server data structure to ten minutes and does not set the expiration of the cookie sent to the client. As described in “The Lifetime of the client Object” on page 275, you can use the `client` object’s `expiration` method to change this expiration period and to set the cookie’s expiration.

When using server cookie, `client.destroy` eliminates all `client` property values.

In general, Netscape cookies have the limitations listed in “Using Client Cookie” on page 268. When you use server cookies, however, these limits are unlikely to be reached because only a single cookie (containing the index) is created.

This technique is fast and has no built-in restrictions on the number and size of properties and their values. You are limited more by how much space you’re willing to use on your server for saving this information.

## Using Server URL Encoding

The server URL encoding technique uses a long unique name, generated by the runtime engine, to index the data structure on the server. In this case, rather than making that generated name be a cookie on the client, the server appends the name to each URL in the generated HTML page. Consequently, the name is sent as many times as there are links on the generated HTML page. (Property names and values are not appended to URLs, just the generated name.) Once again, you can access this generated name with the `ssjs_getClientID` function, described in “Uniquely Referring to the client Object” on page 255.

If you generate URLs dynamically or use the `redirect` function, you can add properties to the URL. For this reason, whenever you call `redirect` or generate your own URL, the compiler does not automatically append the index for you. If you want to retain the index for the `client` properties, use the `addClient` function. For more information, see “Manually Appending client Properties to URLs” on page 277.

If your page has a link to a URL outside of your application, you may not want the client index appended. In this situation, do not use a static string for the `HREF` value. Instead, compute the value. This prevents the runtime engine from automatically appending the client index to the URL. For example, assume you have this link:

```
<A HREF="mailto:me@royalairways.com">
```

In this case, the runtime engine appends the `client` index. To instead have it not do so, use this very similar link:

```
<A HREF=`"mailto:me@royalairways.com"`>
```

In server URL encoding, you lose the `client` identifier (and hence its properties and values) when you submit a form using the `GET` method. You may or may not want to lose these properties, depending on your application's needs.

# The Lifetime of the client Object

Once a client accesses an application, there is no guarantee that it will request further processing or will continue to a logical end point. For this reason, the `client` object has a built-in expiration mechanism. This mechanism allows JavaScript to occasionally “clean up” old `client` objects that are no longer necessary. Each time the server receives a request for a page in an application, JavaScript resets the lifetime of the `client` object.

## Causing client Object Properties to Expire

The default behavior of the expiration mechanism varies, depending on the `client` object maintenance technique you use, as shown in the following table.

Table 13.4 Default expiration of `client` properties based on the maintenance technique

For this maintenance technique...	The properties of the client object...
client cookie	Expire when the browser is exited.
client URL encoding	Never expire.
server cookie	Are removed from the data structure on the server after 10 minutes. The cookie on the client expires when the browser is exited. The <code>client</code> object properties are no longer accessible as soon the data structure is removed or the browser exited.
server URL encoding	Are removed from the data structure on the server after 10 minutes.
server IP address	Are removed from the data structure on the server after 10 minutes.

An application can control the length of time JavaScript waits before cleaning up `client` object properties. To change the length of this period, use the `expiration` method, as in the following example:

```
client.expiration(30);
```

In response to this call, the runtime engine causes `client` object properties to expire after 30 seconds. For server-side maintenance techniques, this call causes the server to remove the object properties from its data structures after 30 seconds. For the two cookie techniques, the call sets the expiration of the cookie to 30 seconds.

If the `client` object expires while there is an active client request using that object, the runtime engine waits until the end of the request before destroying the `client` object.

You must call `expiration` on each application page whose expiration behavior you want to specify. Any page that does not specify an expiration uses the default behavior.

## Destroying the client Object

An application can explicitly destroy a `client` object with the `destroy` method, as follows:

```
client.destroy();
```

When an application calls `destroy`, JavaScript removes all properties from the `client` object.

If you use the client cookie technique to maintain the `client` object, `destroy` eliminates all `client` property values but has no effect on what is stored in the client cookie file. To also eliminate property values from the cookie file, do not use `destroy`; instead, use `expiration` with an argument of 0 seconds.

When you use client URL encoding to maintain the `client` object, the `destroy` method removes all `client` properties. Links on the page before the call to `destroy` retain the `client` properties in their URLs, but links after the call have no properties. Because it is unlikely that you will want only some of the URLs from the page to contain `client` properties, you probably should call `destroy` either at the top or bottom of the page when using client URL maintenance. For more information, see “Using Client URL Encoding” on page 270.

## Manually Appending client Properties to URLs

When using URL encoding either on the client or on the server to maintain the `client` object, in general the runtime engine should store the appropriate information (`client` property names and values or the server data structure's index) in all URLs sent to the client, whether those URLs were presented as static HTML or were generated by server-side JavaScript statements.

The runtime engine automatically appends the appropriate information to HTML hyperlinks that do not occur inside the `SERVER` tag. So, for example, assume your HTML page contains the following statements:

```
<HTML>
For more information, contact
<A HREF="http://royalairways.com/contact_info.html">
Royal Airways</a>
...
</HTML>
```

If the application uses URL encoding for the `client` object, the runtime engine automatically appends the `client` information to the end of the URL. You do not have to do anything special to support this behavior.

However, your application may use the `write` function to dynamically generate an HTML statement containing a URL. You can also use the `redirect` function to start a new request. Whenever you use server-side JavaScript statements to add a URL to the HTML page being generated, the runtime engine assumes that you have specified the complete URL as you want it sent. It does not automatically append client information, even when using URL encoding to maintain the `client` object. If you want client information appended, you must do so yourself.

You use the `addClient` function to manually add the appropriate `client` information. This function takes a URL and returns a new URL with the information appended. For example, suppose the appropriate contact URL varies based on the value of the `client.contact` property. Instead of the HTML above, you might have the following:

```
<HTML>
For more information, contact
<server>
if (client.contact == "VIP") {
    write ("<A HREF='http://royalairways.com/vip_contact_info.html'>");
```

```

        write ("Royal Airways VIP Contact</a>");
    }
    else {
        write ("<A HREF='http://royalairways.com/contact_info.html'>");
        write ("Royal Airways</a>");
    }
</server>
...
</HTML>

```

In this case, the runtime engine does not append `client` properties to the URLs. If you use one of the URL-encoding `client` maintenance techniques, this may be a problem. If you want the `client` properties sent with this URL, instead use this code:

```

<HTML>
For more information, contact
<server>
if (client.contact == "VIP") {
    write (addClient(
        "<A HREF='http://royalairways.com/vip_contact_info.html'>"));
    write ("Royal Airways VIP Contact</a>");
}
else {
    write (addClient(
        "<A HREF='http://royalairways.com/contact_info.html'>"));
    write ("Royal Airways</a>");
}
</server>
...
</HTML>

```

Similarly, any time you use the `redirect` function to change the client request, you should use `addClient` to append the information, as in this example:

```

redirect(addClient("mypage.html"));

```

Conversely, if your page has a link to a URL outside of your application, you may *not* want client information appended. In this situation, do not use a static string for the `HREF` value. Instead, compute the value. This prevents the runtime engine from automatically appending the client index or properties to the URL. For example, assume you have this link:

```

<A HREF="mailto:me@royalairways.com">

```

In this case, the runtime engine appends client information. To instead have it not do so, use this very similar link:

```

<A HREF=`"mailto:me@royalairways.com"`>

```

Even though an application is initially installed to use a technique that does not use URL encoding to maintain `client`, it may be modified later to use a URL encoding technique. Therefore, if your application generates dynamic URLs or uses `redirect`, you may always want to use `addClient`.

## Sharing Objects Safely with Locking

The execution environment for a 3.x version of a Netscape server is multithreaded; this is, it processes more than one request at the same time. Because these requests could require JavaScript execution, more than one thread of JavaScript execution can be active at the same time.

If multiple threads simultaneously attempt to change a property of the same JavaScript object, they could leave the object in an inconsistent state. A section of code in which you want one and only one thread executing at any time is called a **critical section**.

One `server` object is shared by all clients and all applications running on the server. One `project` object is shared by all clients accessing the same application on the server. In addition, your application may create other objects it shares among client requests, or it may even share objects with other applications. To maintain data integrity within any of these shared objects, you must get exclusive access to the object before changing any of its properties.

**Important** There is no implicit locking for the `project` and `server` objects as there was in previous releases.

To better understand what can happen, consider the following example. Assume you create a shared object `project.orders` to keep track of customer orders. You update `project.orders.count` every time there is a new order, using the following code:

```
var x = project.orders.count;
x = x + 1;
project.orders.count = x;
```

Assume that `project.orders.count` is initially set to 1 and two new orders come in, in two separate threads. The following events occur:

1. The first thread stores `project.orders.count` into `x`.
2. Before it can continue, the second thread runs and stores the same value in its copy of `x`.
3. At this point, both threads have a value of 1 in `x`.
4. The second thread completes its execution and sets `project.orders.count` to 2.
5. The first thread continues, unaware that the value of `project.orders.count` has changed, and also sets it to 2.

So, the end value of `project.orders.count` is 2 rather than the correct value, 3.

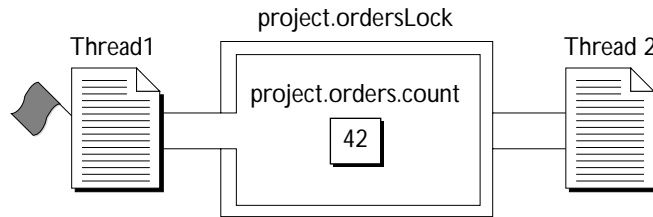
To prevent problems of this kind, you need to obtain exclusive access to the properties of shared objects when writing to them. You can construct your own instances of `Lock` for this purpose that work with any shared object. In addition, the `server` and `project` objects have `lock` and `unlock` methods you can use to restrict access to those objects.

## Using Instances of Lock

Think of a lock as a named flag that you must hold before you gain access to a critical section. If you ask for the named flag and somebody else is already holding it, you wait in line until that person releases the flag. While waiting, you won't change anything you shouldn't. Once you get the flag, anybody else who's waiting for it won't change anything either. If an error occurs or a timeout period elapses before you get the flag, you can either get back in line to wait some more or do something else, such as letting your user know the application is too busy to perform that operation right now. You should not decide to break into the line (by changing shared information)! Figure 13.5 illustrates this process.



Figure 13.5 Thread 2 waits while thread 1 has the lock



In programming terms, a lock is represented by an instance of the `Lock` class. You can use an instance of `Lock` to gain exclusive access to any shared object, providing all code that accesses the shared object honors the lock. Typically, you create your `Lock` instances on the initial page of your application (for reasons that explained later).

In your other pages, before a critical section for the shared object (for example, sections that retrieve and change a property value), you call the `Lock` instance's `lock` method. If that method returns `true`, you have the lock and can proceed. At the end of the critical section, you call the `Lock` instance's `unlock` method.

When a client request in a single execution thread calls the `lock` method, any other request that calls `lock` for the same `Lock` instance waits until the original thread calls the `unlock` method, until some timeout period elapses, or until an error occurs. This is true whether the second request is in a different thread for the same client or in a thread for a different client.

If all threads call the `lock` method before trying to change the shared object, only one thread can enter the critical section at one time.

**Important** The use of locks is completely under the developer's control and requires cooperation. The runtime engine does not force you to call `lock`, nor does it force you to respect a lock obtained by someone else. If you don't ask, you can change anything you want. For this reason, it's very important to get into the habit of always calling `lock` and `unlock` when entering any critical section of code and to check the return value of `lock` to ensure you have the lock. You can think of it in terms of holding a flag: if you don't ask for the flag, you won't be told to wait in line. If you don't wait in line, you might change something you shouldn't.

You can create as many locks as you need. The same lock may be used to control access to multiple objects, or each object (or even object property) can have its own lock.

A lock is just a JavaScript object itself; you can store a reference to it in any other JavaScript object. Thus, for example, it is common practice to construct a `Lock` instance and store it in the `project` object.

**Note** Because using a lock blocks other users from accessing the named flag, potentially delaying execution of their tasks, it is good practice to use locks for as short a period as possible.

The following code illustrates how to keep track of customer orders in the shared `project.orders` object discussed earlier and to update `project.orders.count` every time there is a new order. In the application's initial page, you include this code:

```
// Construct a new Lock and save in project
project.ordersLock = new Lock();
if (! project.ordersLock.isValid()) {
    // Unable to create a Lock. Redirect to error page
    redirect ("sysfailure.htm");
}
```

This code creates the `Lock` instance and verifies (in the call to `isValid`) that nothing went wrong creating it. Only in very rare cases is your `Lock` instance improperly constructed. This happens only if the runtime engine runs out of system resources while creating the object.

You typically create your `Lock` instances on the initial page so that you don't have to get a lock before you create the `Lock` instances. The initial page is run exactly once during the running of the application, when the application is started on the server. For this reason, you're guaranteed that only one instance of each lock is created.

If, however, your application creates a lock on another of its pages, multiple requests could be invoking that page at the same time. One request could check for the existence of the lock and find it not there. While that request creates the lock, another request might create a second lock. In the meantime, the first request calls the `lock` method of its object. Then the second request calls the `lock` method of *its* object. Both requests now think they have safe access to the critical section and proceed to corrupt each other's work.

Once it has a valid lock, your application can continue. On a page that requires access to a critical section, you can use this code:

```
// Begin critical section -- obtain lock
if ( project.ordersLock.lock() ) {
    var x = project.orders.count;
```

```

    x = x + 1;
    project.orders.count = x;

    // End critical section -- release lock
    project.ordersLock.unlock();
}
else
    redirect("combacklater.htm");

```

This code requests the lock. If it gets the lock (that is, if the `lock` method returns `true`), then it enters the critical section, makes the changes, and finally releases the lock. If the `lock` method returns `false`, then this code did not get the lock. In this case, it redirects the application to a page that indicates the application is currently unable to satisfy the request.

## Special Locks for project and server Objects

The `project` and `server` objects each have `lock` and `unlock` methods. You can use these methods to obtain exclusive access to properties of those objects.

There is nothing special about these methods. You still need cooperation from other sections of code. You can think of these methods as already having one flag named “project” and another named “server.” If another section of code does not call `project.lock`, it can change any of the `project` object’s properties.

Unlike the `lock` method of the `Lock` class, however, you cannot specify a timeout period for the `lock` method of the `project` and `server` objects. That is, when you call `project.lock`, the system waits indefinitely for the lock to be free. If you want to wait for only a specified amount of time, instead use an instance of the `Lock` class.

The following example uses `lock` and `unlock` to get exclusive access to the `project` object while modifying the customer ID property:

```

project.lock()
project.next_id = 1 + project.next_id;
client.id = project.next_id;
project.unlock();

```

## Avoiding Deadlock

You use locks to protect a critical section of your code. In practice, this means one request waits while another executes in the critical section. You must be careful in using locks to protect critical sections. If one request is waiting for a lock that is held by a second request, and that second request is waiting for a lock held by the first request, neither request can ever continue. This situation is called **deadlock**.

Consider the earlier example of processing customer orders. Assume that the application allows two interactions. In one, a user enters a new customer; in the other, the user enters a new order. As part of entering a new customer, the application also creates a new customer order. This interaction is done in one page of the application that could have code similar to the following:

```
// Create a new customer.
if ( project.customersLock.lock() ) {

    var id = project.customers.ID;
    id = id + 1;
    project.customers.ID = id;

    // Start a new order for this new customer.
    if ( project.ordersLock.lock() ) {

        var c = project.orders.count;
        c = c + 1;
        project.orders.count = c;
        project.ordersLock.unlock();
    }

    project.customersLock.unlock();
}
```

In the second type of interaction, a user enters a new customer order. As part of entering the order, if the customer is not already a registered customer, the application creates a new customer. This interaction is done in a different page of the application that could have code similar to the following:

```
// Start a new order.
if ( project.ordersLock.lock() ) {

    var c = project.orders.count;
    c = c + 1;
    project.orders.count = c;

    if (...code to establish unknown customer...) {

        // Create a new customer.
        // This internal lock is going to cause trouble!
```

```

        if ( project.customersLock.lock() ) {
            var id = project.customers.ID;
            id = id + 1;
            project.customers.ID = id;
            project.customersLock.unlock();
        }
    }
    project.ordersLock.unlock();
}

```

Notice that each of these code fragments tries to get a second lock while already holding a lock. That can cause trouble. Assume that one thread starts to create a new customer; it obtains the `customersLock` lock. At the same time, another thread starts to create a new order; it obtains the `ordersLock` lock. Now, the first thread requests the `ordersLock` lock. Since the second thread has this lock, the first thread must wait. However, assume the second thread now asks for the `customersLock` lock. The first thread holds that lock, so the second thread must wait. The threads are now waiting for each other. Because neither specified a timeout period, they will both wait indefinitely.

In this case, it is easy to avoid the problem. Since the values of the customer ID and the order number do not depend on each other, there is no real reason to nest the locks. You could avoid potential deadlock by rewriting both code fragments. Rewrite the first fragment as follows:

```

// Create a new customer.
if ( project.customersLock.lock() ) {
    var id = project.customers.ID;
    id = id + 1;
    project.customers.ID = id;
    project.customersLock.unlock();
}

// Start a new order for this new customer.
if ( project.ordersLock.lock() ) {
    var c = project.orders.count;
    c = c + 1;
    project.orders.count = c;
    project.ordersLock.unlock();
}

```

The second fragment looks like this:

```

// Start a new order.
if ( project.ordersLock.lock() ) {

```

```

    var c = project.orders.count;
    c = c + 1;
    project.orders.count = c;

    project.ordersLock.unlock();
  }

  if (...code to establish unknown customer...) {
    // Create a new customer.
    if ( project.customersLock.lock() ) {
      var id = project.customers.ID;
      id = id + 1;
      project.customers.ID = id;

      project.customersLock.unlock();
    }
  }
}

```

Although this situation is clearly contrived, deadlock is a very real problem and can happen in many ways. It does not even require that you have more than one lock or even more than one request. Consider code in which two functions each ask for the same lock:

```

function fn1 () {
  if ( project.lock() ) {
    // ... do some stuff ...
    project.unlock();
  }
}

function fn2 () {
  if ( project.lock() ) {
    // ... do some other stuff ...
    project.unlock();
  }
}

```

By itself, that is not a problem. Later, you change the code slightly, so that `fn1` calls `fn2` while holding the lock, as shown here:

```

function fn1 () {
  if ( project.lock() ) {
    // ... do some stuff ...
    fn2();
    project.unlock();
  }
}

```

Now you have deadlock. This is particularly ironic, in that a single request waits forever for itself to release a flag!

# Other JavaScript Functionality

This chapter describes additional server-side JavaScript functionality you can use to send email messages from you application, access the server file system, include external libraries in your application, or directly manipulate client requests and client responses.

This chapter contains the following sections:

- Mail Service
- File System Service
- Working with External Libraries
- Request and Response Manipulation

## Mail Service

Your application may need to send an email message. You use an instance of the `SendMail` class for this purpose. The only methods of `SendMail` are `send`, to send the message, and `errorCode` and `errorMessage`, to interpret an error.

For example, the following script sends mail to `vpg` with the specified subject and body for the message:

```
<server>
SMName = new SendMail();
SMName.To = "vpg@royalairways.com";
SMName.From = "thisapp@netscape.com";
```

```
SMName.Subject = "Here's the information you wanted";
SMName.Body = "sharm, maldives, phuket, coral sea, taveuni, maui,
              cocos island, marathon cay, san salvador";
SMName.send();
</server>
```

The following table describes the properties of the `SendMail` class. The `To` and `From` properties are required; all other properties are optional.

Table 14.1 Properties of the `SendMail` class

To	A comma-delimited list of primary recipients of the message.
From	The user name of the person sending the message.
Cc	A comma-delimited list of additional recipients of the message.
Bcc	A comma-delimited list of recipients of the message whose names should not be visible in the message.
Smtptserver	The mail (SMTP) server name. This property defaults to the value specified through the setting in the administration server.
Subject	The subject of the message.
Body	The text of the message.

In addition to these properties, you can add any other properties you wish. All properties of the `SendMail` class are included in the header of the message when it is actually sent. For example, the following code sends a message to bill from vpg, setting vpg's organization field to Royal Airways. Replies to the message go to vpgboss.

```
mailObj["Reply-to"] = "vpgboss";
mailObj.Organization = "Royal Airways";
mailObj.From = "vpg";
mailObj.To = "bill";
mailObj.send();
```

For more information on predefined header fields, refer to [RFC 822](#), the standard for the format of internet text messages.

The `SendMail` class allows you to send either simple text-only mail messages or complex MIME-compliant mail. You can also add attachments to your message. To send a MIME message, add a `Content-type` property to the `SendMail` object and set its value to the MIME type of the message.



For example, the following code segment sends a GIF image:

```
<server>
SMName = new SendMail();
SMName.To = "vpg@royalairways.com";
SMName.From = "thisapp@netscape.com";
SMName.Subject = "Here's the image file you wanted";
SMName["Content-type"] = "image/gif";
SMName["Content-Transfer-Encoding"] = "base64";

// In this next statement, image2.gif must be base 64 encoded.
// If you use uuencode to encode the GIF file, delete the header
// (for example, "begin 644 image2.gif") and the trailer ("end").
fileObj = new File("/usr/somebody/image2.gif");

openFlag = fileObj.open("r");
if ( openFlag ) {
    len = fileObj.getLength();
    SMName.Body = fileObj.read(len);
    SMName.send();
}
</server>
```

Some MIME types may need more information. For example, if the content type is multipart/mixed, you must also specify a boundary separator for one or more different sets of data in the body. For example, the following code sends a multipart message containing two parts, both of which are plain text:

```
<server>
SMName = new SendMail();
SMName.To = "vpg@royalairways.com";
SMName.From = "thisapp@netscape.com";
SMName.Subject = "Here's the information you wanted";
SMName["Content-type"]
    = "multipart/mixed; boundary=\"simple boundary\"";
fileObj = new File("/usr/vpg/multi.txt");
openFlag = fileObj.open("r");
if ( openFlag ) {
    len = fileObj.getLength();
    SMName.Body = fileObj.read(len);
    SMName.send();
}
</server>
```

Here the file `multi.txt` contains the following multipart message:

```
This is the place for preamble.
It is to be ignored.
It is a handy place for an explanatory note to non-MIME compliant
readers.
--simple boundary
```

```
This is the first part of the body.  
This does NOT end with a line break.  
  
--simple boundary  
Content-Type: text/plain; charset=us-ascii  
  
This is the second part of the body.  
It DOES end with a line break  
  
--simple boundary--  
This is the epilogue. It is also to be ignored.
```

You can nest multipart messages. That is, if you have a message whose content type is multipart, you can include another multipart message in its body. In such cases, be careful to ensure that each nested multipart entity uses a different boundary delimiter.

For details on MIME types, refer to RFC 1341<sup>1</sup>, the MIME standard. For more information on sending mail messages with JavaScript, see the description of this class in the *Server-Side JavaScript Reference*.

## File System Service

JavaScript provides a `File` class that enables applications to write to the server's file system. This is useful for generating persistent HTML files and for storing information without using a database server. One of the main advantages of storing information in a file instead of in JavaScript objects is that the information is preserved even if the server goes down.

## Security Considerations

Exercise caution when using the `File` class. A JavaScript application can read or write files anywhere the operating system allows, potentially including sensitive system files. You should be sure your application does not allow an intruder to read password files or other sensitive information or to write files at will. Take care that the filenames you pass to its methods cannot be modified by an intruder.

---

1. <http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1341.txt>

For example, do not use `client` or `request` properties as filenames, because the values may be accessible to an intruder through cookies or URLs. In such cases, the intruder can modify cookie or URL values to gain access to sensitive files.

For similar security reasons, Navigator does not provide automatic access to the file system of client machines. If needed, the user can save information directly to the client file system by making appropriate menu choices in Navigator.

## Creating a File Object

To create an instance of the `File` class, use the standard JavaScript syntax for object creation:

```
fileObjectName = new File("path");
```

Here, `fileObjectName` is the name by which you refer to the file, and `path` is the complete file path. The path should be in the format of the server's file system, not a URL path.

You can display the name of a file by using the `write` function, with the `File` object as its argument. For example, the following statement displays the filename:

```
x = new File("\path\file.txt");  
write(x);
```

## Opening and Closing a File

Once you have created a `File` object, you use the `open` method to open the file so that you can read from it or write to it. The `open` method has the following syntax:

```
result = fileObjectName.open("mode");
```

This method returns `true` if the operation is a success and `false` otherwise. If the file is already open, the operation fails and the original file remains open.

The parameter `mode` is a string that specifies the mode in which to open the file. The following table describes how the file is opened for each mode.

Table 14.2 File-access modes

Mode	Description
<code>r</code>	Opens the file, if it exists, as a text file for reading and returns <code>true</code> . If the file does not exist, returns <code>false</code> .
<code>w</code>	Opens the file as a text file for writing. Creates a new (initially empty) text file whether or not the file exists.
<code>a</code>	Opens the file as a text file for appending (writing at the end of the file). If the file does not already exist, creates it.
<code>r+</code>	Opens the file as a text file for reading and writing. Reading and writing commence at the beginning of the file. If the file exists, returns <code>true</code> . If the file does not exist, returns <code>false</code> .
<code>w+</code>	Opens the file as a text file for reading and writing. Creates a new (initially empty) file whether or not the file already exists.
<code>a+</code>	Opens the file as a text file for reading and writing. Reading and writing commence at the end of the file. If the file does not exist, creates it.
<code>b</code>	When appended to any of the preceding modes, opens the file as a binary file rather than a text file. Applicable only on Windows operating systems.

When an application has finished using a file, it can close the file by calling the `close` method. If the file is not open, `close` fails. This method returns `true` if successful and `false` otherwise.

## Locking Files

Most applications can be accessed by many users simultaneously. In general, however, different users should not try to make simultaneous changes to the same file, because unexpected errors may result.

To prevent multiple users from modifying a file at the same time, use one of the locking mechanisms provided by the Session Management Service, as described in “Sharing Objects Safely with Locking” on page 279. If one user has the file

locked, other users of the application wait until the file becomes unlocked. In general, this means you should precede all file operations with `lock` and follow them with `unlock`.

If only one application can modify the same file, you can obtain the lock within the `project` object. If more than one application can access the same file, however, obtain the lock within the `server` object.

For example, suppose you have created a file called `myFile`. Then you could use it as follows:

```
if ( project.lock() ) {
  myFile.open("r");
  // ... use the file as needed ...
  myFile.close();
  project.unlock();
}
```

In this way, only one user of the application has modify to the file at one time. Alternatively, for finer locking control you could create your own instance of the `Lock` class to control access to a file. This is described in “Using Instances of `Lock`” on page 280.

## Working with Files

The `File` class has a number of methods that you can use once a file is open:

- **Positioning:** `setPosition`, `getPosition`, `eof`. Use these methods to set and get the current pointer position in the file and determine whether the pointer is at the end of the file.
- **Reading from a file:** `read`, `readln`, `readByte`.
- **Writing to a file:** `write`, `writeln`, `writeByte`, `flush`.
- **Converting between binary and text formats:** `byteToString`, `stringToByte`. Use these methods to convert a single number to a character and vice versa.
- **Informational methods:** `getLength`, `exists`, `error`, `clearError`. Use these methods to get information about a file and to get and clear error status.

The following sections describe these methods.

## Positioning Within a File

The physical file associated with a `File` object has a pointer that indicates the current position in the file. When you open a file, the pointer is either at the beginning or at the end of the file, depending on the mode you used to open it. In an empty file, the beginning and end of the file are the same.

The `setPosition` method positions the pointer within the file, returning `true` if successful and `false` otherwise.

```
fileObj.setPosition(position);  
fileObj.setPosition(position, reference);
```

Here, `fileObj` is a `File` object, `position` is an integer indicating where to position the pointer, and `reference` indicates the reference point for `position`, as follows:

- 0: relative to beginning of file
- 1: relative to current position
- 2: relative to end of file
- Other (or unspecified): relative to beginning of file

The `getPosition` method returns the current position in the file, where the first byte in the file is always byte 0. This method returns -1 if there is an error.

```
fileObj.getPosition();
```

The `eof` method returns `true` if the pointer is at the end of the file and `false` otherwise. This method returns `true` after the first read operation that attempts to read past the end of the file.

```
fileObj.eof();
```

## Reading from a File

Use the `read`, `readln`, and `readByte` methods to read from a file.

The `read` method reads the specified number of bytes from a file and returns a string.

```
fileObj.read(count);
```

Here, `fileObj` is a `File` object, and `count` is an integer specifying the number of bytes to read. If `count` specifies more bytes than are left in the file, then the method reads to the end of the file.

The `readln` method reads the next line from the file and returns it as a string.

```
fileObj.readln();
```

Here, `fileObj` is a `File` object. The line-separator characters (either `\r\n` on Windows or just `\n` on Unix or Macintosh) are not included in the string. The character `\r` is skipped; `\n` determines the actual end of the line. This compromise gets reasonable behavior on all platforms.

The `readByte` method reads the next byte from the file and returns the numeric value of the next byte, or `-1`.

```
fileObj.readByte();
```

## Writing to a File

The methods for writing to a file are `write`, `writeln`, `writeByte`, and `flush`.

The `write` method writes a string to the file. It returns `true` if successful and `false` otherwise.

```
fileObj.write(string);
```

Here, `fileObj` is a `File` object, and `string` is a JavaScript string.

The `writeln` method writes a string to the file, followed by `\n` (`\r\n` in text mode on Windows). It returns `true` if the write was successful and `false` otherwise.

```
fileObj.writeln(string);
```

The `writeByte` method writes a byte to the file. It returns `true` if successful and `false` otherwise.

```
fileObj.writeByte(number);
```

Here, `fileObj` is a `File` object and `number` is a number.

When you use any of these methods, the file contents are buffered internally. The `flush` method writes the buffer to the file on disk. This method returns `true` if successful and `false` otherwise.

```
fileObj.flush();
```

## Converting Data

There are two primary file formats: ASCII text and binary. The `byteToString` and `stringToByte` methods of the `File` class convert data between these formats.

The `byteToString` method converts a number into a one-character string. This method is static. You can use the `File` class object itself, and not an instance, to call this method.

```
File.byteToString(number);
```

If the argument is not a number, the method returns the empty string.

The `stringToByte` method converts the first character of its argument, a string, into a number. This method is also static.

```
File.stringToByte(string);
```

The method returns the numeric value of the first character, or 0.

## Getting File Information

You can use several `File` methods to get information on files and to work with the error status.

The `getLength` method returns the number characters in a text file or the number of bytes in any other file. It returns -1 if there is an error.

```
fileObj.getLength();
```

The `exists` method returns `true` if the file exists and `false` otherwise.

```
fileObj.exists();
```

The `error` method returns the error status, or -1 if the file is not open or cannot be opened. The error status is a nonzero value if an error occurred and 0 otherwise (no error). Error status codes are platform dependent; refer to your operating system documentation.

```
fileObj.error();
```

The `clearError` method clears both the error status (the value of `error`) and the value of `eof`.

```
fileObj.clearError();
```



## Example

Netscape servers include the Viewer sample application in its directory structure. Because this application allows you to view any files on the server, it is not automatically installed.

Viewer gives a good example of how to use the `File` class. If you install it, be sure to restrict access so that unauthorized persons cannot view files on your server. For information on restricting access to an application, see “Deploying an Application” on page 70.

The following code from the viewer sample application creates a `File` class, opens it for reading, and generates HTML that echoes the lines in the file, with a hard line break after each line.

```
x = new File("\tmp\names.txt");
fileIsOpen = x.open("r");
if (fileIsOpen) {
    write("file name: " + x + "<BR>");
    while (!x.eof()) {
        line = x.readln();
        if (!x.eof())
            write(line+"<br>");
    }
    if (x.error() != 0)
        write("error reading file" + "<BR>");
    x.close();
}
```

## Working with External Libraries

The recommended way to communicate with external applications is using LiveConnect, as described in Chapter 21, “LiveConnect Overview.” However, you can also call functions written in languages such as C, C++, or Pascal and compiled into libraries on the server. Such functions are called *native functions* or *external functions*. Libraries of native functions, called *external libraries*, are dynamic link libraries on Windows operating systems and shared objects on Unix operating systems.

**Important** Be careful when using native functions with your application. Native functions can compromise security if the native program processes a command-line entry from the user (for example, a program that allows users to enter operating

system or shell commands). This functionality is dangerous because an intruder can attach additional commands using semicolons to append multiple statements. It is best to avoid command-line input, unless you strictly check it.

Using native functions in an application is useful in these cases:

- If you already have complex functions written in native code that you can use in your application.
- If the application requires computation-intensive functions. In general, functions written in native code run faster than those written in JavaScript.
- If the application requires some other task you cannot do in JavaScript.

The sample directory `jsacall` contains source and header files illustrating how to call functions in external libraries from a JavaScript application.

In the Application Manager, you associate an external library with a particular application. However, once associated with any installed application, an external library can be used by all installed applications.

Follow these steps to use a native function library in a JavaScript application:

1. Write and compile an external library of native functions in a form compatible with JavaScript. (See “Guidelines for Writing Native Functions” on page 299.)
2. With the Application Manager, identify the library to be used by installing a new application or modifying installation parameters for an existing application. Once you identify an external library using the Application Manager, all applications on the server can call external functions in that library. (See “Identifying Library Files” on page 299.)
3. Restart the server to load the library with your application. The functions in the external library are now available to all applications on the server.
4. In your application, use the JavaScript functions `registerCFunction` to identify the library functions to be called and `callC` to call those functions. (See “Registering Native Functions” on page 300 and “Using Native Functions in JavaScript” on page 300.)
5. Recompile and restart your application for the changes to take effect.

**Important** You must restart your server to install a library to use with an application. You must restart the server any time you add new library files or change the names of the library files used by an application.

## Guidelines for Writing Native Functions

Although you can write external libraries in any language, JavaScript uses C calling conventions. Your code must include the header file `jsacall.h` provided in `js\samples\jsacall\`.

This directory also includes the source code for a sample application that calls a C function defined in `jsacall.c`. Refer to these files for more specific guidelines on writing C functions for use with JavaScript.

Functions to be called from JavaScript must be exported and must conform to this type definition:

```
typedef void (*LivewireUserCFunction)
(int argc, struct LivewireCCallData argv[],
 struct LivewireCCallData* result, pblock* pb,
 Session* sn, Request* rq);
```

## Identifying Library Files

Before you can run an application that uses native functions in external libraries, you must identify the library files. Using the Application Manager, you can identify libraries when you initially install an application (by clicking Add) or when you modify an application's installation parameters (by clicking Modify). For more information on identifying library files with the Application Manager, see "Installing a New Application" on page 61.

**Important** After you enter the paths of library files in the Application Manager, you must restart your server for the changes to take effect. You must then be sure to compile and restart your application.

Once you have identified an external library using the Application Manager, all applications running on the server can call functions in the library (by using `registerCFunction` and `callC`).

## Registering Native Functions

Use the JavaScript function `registerCFunction` to register a native function for use with a JavaScript application. This function has the following syntax:

```
registerCFunction(JSFunctionName, libraryPath, CFunctionName);
```

Here, `JSFunctionName` is the name of the function as it will be called in JavaScript with the `callC` function. The `libraryPath` parameter is the full pathname of the library, using the conventions of your operating system and the `CFunctionName` parameter is the name of the C function as it is defined in the library. In this method call, you must use the exact case shown in the Application Manager, even on NT.

**Note** Backslash (\) is a special character in JavaScript, so you must use either forward slash (/) or a double backslash (\\) to separate Windows directory and filenames in `libraryPath`.

This function returns `true` if it registers the function successfully and `false` otherwise. The function might fail if JavaScript cannot find the library at the specified location or the specified function inside the library.

An application must use `registerCFunction` to register a function before it can use `callC` to call it. Once the application registers the function, it can call the function any number of times. A good place to register functions is in an application's initial page.

## Using Native Functions in JavaScript

Once your application has registered a function, it can use `callC` to call it. This function has the following syntax:

```
callC(JSFunctionName, arguments);
```

Here, `JSFunctionName` is the name of the function as it was identified with `registerCFunction` and `arguments` is a comma-delimited list of arguments to the native function. The arguments can be any JavaScript values: strings, numbers, Boolean values, objects, or null. The number of arguments must match the number of arguments required by the external function. Although you can specify a JavaScript object as an argument, doing so is rarely useful, because the object is converted to a string before being passed to the external function.

This function returns a string value returned by the external function. The `callC` function can return only string values.

The `jsacall` sample JavaScript application illustrates the use of native functions. The `jsacall` directory includes C source code (in `jsacall.c`) that defines a C function named `mystuff_EchoCCallArguments`. This function accepts any number of arguments and then returns a string containing HTML listing the arguments. This sample illustrates calling C functions from a JavaScript application and returning values.

To run `jsacall`, you must compile `jsacall.c` with your C compiler. Command lines for several common compilers are provided in the comments in the file.

The following JavaScript statements (taken from `jsacall.html`) register the C function as `echoCCallArguments` in JavaScript, call the function `echoCCallArguments`, and then generate HTML based on the value returned by the function.

```
var isRegistered = registerCFunction("echoCCallArguments",
    "c:\\mycode\\mystuff.dll", "mystuff_EchoCCallArguments");
if (isRegistered == true) {
    var returnValue = callC("echoCCallArguments",
        "first arg",
        42,
        true,
        "last arg");
    write(returnValue);
}
else {
    write("registerCFunction() returned false, "
        + "check server error log for details")
}
```

The `echoCCallArguments` function creates a string result containing HTML that reports both the type and the value of each of the JavaScript arguments passed to it. If the `registerCFunction` returns true, the code above generates this HTML:

```
argc = 4<BR>
argv[0].tag: string; value = first arg<BR>
argv[1].tag: double; value = 42<BR>
argv[2].tag: boolean; value = true<BR>
argv[3].tag: string; value = last arg<BR>
```

# Request and Response Manipulation

A typical request sent by the client to the server has no content type. The JavaScript runtime engine automatically handles such requests. However, if the user submits a form, then the client automatically puts a content type into the header to tell the server how to interpret the extra form data. That content type is usually `application/x-www-form-urlencoded`. The runtime engine also automatically handles requests with this content type. In these situations, you rarely need direct access to the request or response header. If, however, your application uses a different content type, it must be able to manipulate the request header itself.

Conversely, the typical response sent from the server to the client has the `text/html` content type. The runtime engine automatically adds that content type to its responses. If you want a different content type in the response, you must provide it yourself.

To support these needs, the JavaScript runtime engine on the server allows your application to access (1) the header of any request and (2) the raw body of a request that has a nonstandard content type. You already control the body of the response through the `SERVER` tag and your HTML tags. The functionality described in this section also allows you to control the header of the response.

You can use this functionality for various purposes. For example, as described in “Using Cookies” on page 239, you can communicate between the client and server processes using cookies. Also, you can use this functionality to support a file upload.

The World Wide Web Consortium publishes online information about the HTTP protocol and information that can be sent using that protocol. See, for example, *HTTP Specifications and Drafts*.

## Request Header

To access the name/value pairs of the header of the client request, use the `httpHeader` method of the `request` object. This method returns an object whose properties and values correspond to the name/value pairs of the header.

For example, if the request contains a cookie, `header["cookie"]` or `header.cookie` is its value. The `cookie` property, containing all of the cookie's name/value pairs (with the values encoded as described in “Using Cookies” on page 239), must be parsed by your application.

The following code prints the properties and values of the header:

```
var header = request.httpHeader();
var count = 0;
var i;

for (i in header ) {
    write(count + ". " + i + " " + header[i] + "<br>\n");
    count++;
}
```

If you submitted a form using the `GET` method, your output might look like this:

```
0. connection Keep-Alive
1. user-agent Mozilla/4.0b1 (WinNT; I)
2. host piccolo:2020
3. accept image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

If you used the `POST` method to submit your form, your output might look like this:

```
0. referer http://piccolo:2020/world/hello.html
1. connection Keep-Alive
2. user-agent Mozilla/4.0b1 (WinNT; I)
3. host piccolo:2020
4. accept image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
5. cookie NETSCAPE_LIVEWIRE.oldname=undefined;
   NETSCAPE_LIVEWIRE.number=0
6. content-type multipart/form-data; boundary=-----
   --79741602416605
7. content-length 208
```

## Request Body

For normal HTML requests, the content type of the request is `application/x-www-form-urlencoded`. Upon receiving a request with this content type, the JavaScript runtime engine on the server processes the request using the data in the body of the request. In this situation, you cannot directly access the raw data of the request body. (Of course, you can access its content through the `request` and `client` objects constructed by the runtime engine.)

If, however, the request has any other content type, the runtime engine does not automatically process the request body. In this situation, it is up to your application to decide what to do with the content.

Presumably, another page of your application posted the request for this page. Therefore, your application must expect to receive unusual content types and should know how to handle them.

To access the body of a request, you use the `getPostData` method of the `request` object. This method takes as its parameter the number of characters of the body to return. If you specify 0, it returns the entire body. The return value is a string containing the requested characters. If there is no available data, the method returns the empty string.

You can use this method to get all of the characters at once, or you can read chunks of data. Think of the body of the request as a stream of characters. As you read them, you can only go forward; you can't read the same characters multiple times.

To assign the entire request body to the `postData` variable, you can use the following statement:

```
postData = request.getPostData(0);
```

If you specify 0 as the parameter, the method gets the entire request. You can explicitly find out how many characters are in the information using the header's `content-length` property, as follows:

```
length = parseInt(header["content-length"], 10);
```

To get the request body in smaller chunks, you can specify a different parameter. For example, the following code processes the request body in chunks of 20 characters:

```
var length = parseInt(header["content-length"], 10);  
var i = 0;
```



```

while (i < length) {
    postData = request.getPostData(20);
    // ...process postData...
    i = i + 20;
}

```

Of course, this would be a sensible approach only if you knew that chunks consisting of 20 characters of information were meaningful in the request body.

## Response Header

If the response you send to the client uses a custom content type, you should encode this content type in the response header. The JavaScript runtime engine automatically adds the default content type (`text/html`) to the response header. If you want a custom header, you must first remove the old default content type from the header and then add the new one. You do so with the `addResponseHeader` and `deleteResponseHeader` functions.

For example, if your response uses `royalairways-format` as a custom content type, you would specify it this way:

```

deleteResponseHeader("content-type");
addResponseHeader("content-type", "royalairways-format");

```

You can use the `addResponseHeader` function to add any other information you want to the response header.

**Important** Remember that the header is sent with the first part of the response. Therefore, you should call these functions early in the script on each page. In particular, you should ensure that the response header is set *before* any of these happen:

- The runtime engine generates 64KB of content for the HTML page (it automatically flushes the output buffer at this point).
- You call the `flush` function to clear the output buffer.
- You call the `redirect` function to change client requests.

For more information, see “Flushing the Output Buffer” on page 226 and “Runtime Processing on the Server” on page 220.



# 4

## *LiveWire Database Service*

- Connecting to a Database
- Working with a Database
- Configuring Your Database
- Data Type Conversion
- Error Handling for LiveWire
- Videoapp and Oldvideo Sample Applications



# Connecting to a Database

This chapter discusses how to use the LiveWire Database Service to connect your application to DB2, Informix, ODBC, Oracle, or Sybase relational databases. It describes how to choose the best connection methodology for your application.

This chapter contains the following sections:

- Interactions with Databases
- Approaches to Connecting
- Database Connection Pools
- Single-Threaded and Multithreaded Databases
- Managing Connection Pools
- Individual Database Connections

# Interactions with Databases

Your JavaScript applications running on Netscape Enterprise Server can use the LiveWire Database Service to access databases on Informix, Oracle, Sybase, and DB2 servers and on servers using the Open Database Connectivity (ODBC) standard. Your applications running on Netscape FastTrack Server can access only databases on servers using the ODBC standard.

The following discussions assume you are familiar with relational databases and Structured Query Language (SQL).

Before you create a JavaScript application that uses LiveWire, the database or databases you plan to connect to should already exist on the database server. Also, you should be familiar with their structure. If you create an entirely new application, including the database, you should design, create, and populate the database (at least in prototype form) before creating the application to access it.

Before you try to use LiveWire, be sure your environment is properly configured. For information on how to configure it, see Chapter 17, “Configuring Your Database.” Also, you can use the `videoapp` sample application, described in Chapter 20, “Videoapp and Oldvideo Sample Applications,” to explore some of LiveWire’s capabilities.

Typically, to interact with a database, you follow these general steps:

1. Use the `database` object or create a `DbPool` object to establish a pool of database connections. This is typically done on the initial page of the application, unless your application requires that users have a special database connection.
2. Connect the pool to the database. Again, this is typically done on the application’s initial page.
3. Retrieve a connection from the pool. This is done implicitly when you use the `database` object or explicitly when you use the `connection` method of a `DbPool` object.
4. If you’re going to change information in the database, begin a transaction. Database transactions are discussed in “Managing Transactions” on page 348.

5. Either create a cursor or call a database stored procedure to work with information from the database. This could involve, for example, displaying results from a query or updating database contents. Close any open cursors, results sets, and stored procedures when you have finished using them. Cursors are discussed in “Manipulating Query Results with Cursors” on page 338; Stored procedures are discussed in “Calling Stored Procedures” on page 354.
6. Commit or rollback an open transaction.
7. Release the database connection (if you’re using `Connection` objects).

This chapter discusses the first three of these steps. Chapter 16, “Working with a Database,” discusses the remaining steps.

## Approaches to Connecting

There are two basic ways to connect to a database with the LiveWire Database Service. You can use `DbPool` and `Connection` objects, or you can use the `database` object.

**Connecting with `DbPool` and `Connection` objects.** In this approach, you create a pool of database connections for working with a relational database. You create an instance of the `DbPool` class and then access `Connection` objects through that `DbPool` object. `DbPool` and `Connection` objects separate the activities of connecting to a database and managing a set of connections from the activities of accessing the database through a connection.

This approach offers a lot of flexibility. Your application can have several database pools, each with its own configuration of database and user. Each pool can have multiple connections for that configuration. This allows simultaneous access to multiple databases or to the same database from multiple accounts. You can also associate the connection pool with the application itself instead of with a single client request and thus have transactions that span multiple client requests. You make this association by assigning the pool to a property of the `project` object and then removing the assignment when you’re finished with the pool.

**Connecting with the database object.** In this approach, you use the predefined `database` object for connecting to a database with a single connection configuration of database and user. The `database` object performs all activities related to working with a database. You can think of the `database` object as a single pool of database connections.

This approach is somewhat simpler, as it involves using only the single `database` object and not multiple `DbPool` and `Connection` objects. However, it lacks the flexibility of the first approach. If you use only the `database` object and want to connect to different databases or to different accounts, you must disconnect from one configuration before connecting to another. Also, when you use the `database` object, a single transaction cannot span multiple client requests, and connections to multiple database sources cannot be simultaneously open.

As described in the following sections, you need to consider two main questions when deciding how to set up your database connections:

- How many configurations of database and user do you need?
- Does a single database connection need to span multiple client requests?



The following table summarizes how the answers to these questions affect how you set up and manage your pool of database connections and the individual connections. The following sections discuss the details of these possibilities.

Table 15.1 Considerations for creating the database pools

Number of database configurations?	Where is the pool connected?	Where is the pool disconnected?	What object(s) hold the pool?	Does your code need to store the pool and connection?	How does your code store the pool and connections in the project object?
1, shared by all clients	Application's initial page	Nowhere	database	No	--
1, shared by all clients	Application's initial page	Nowhere	1 DbPool object	Yes	DbPool: Named property; Connection: 1 array
Fixed set, shared by all clients	Application's initial page	Nowhere	N DbPool objects	Yes	DbPool: Named property; Connection: N arrays
Separate pool for each client	Client request page	Depends <sup>a</sup>	Many DbPool objects	Only if a connection spans client requests	DbPool: 1 array; Connection: 1 array

a. If an individual connection does not span client requests, you can connect and disconnect the pool on each page that needs a connection. In this case, the pool is not stored between requests. If individual connections do span requests, connect on the first client page that needs the connection and disconnect on the last such page. This can result in idle connections, so your application will need to handle that possibility.

# Database Connection Pools

If you want to use the database object, you do not have to create it. It is a predefined object provided for you by the JavaScript runtime engine. Alternatively, if you want the additional capabilities of the `DbPool` class, you create an instance of the `DbPool` class and connect that object to a particular database which creates a pool of connections.

You can either create a generic `DbPool` object and later specify the connection information (using its `connect` method) or you can specify the connection information when you create the pool. A generic `DbPool` object doesn't have any available connections at the time it is created. For this reason, you may want to connect when you create the object. If you use the database object, you must always make the connection by calling `database.connect`.

```
connect (dbtype, serverName, userName, password,
        databaseName, maxConnections, commitFlag);
```

You can specify the following information when you make a connection, either when creating a `DbPool` object or when calling the `connect` method of `DbPool` or `database`:

- `dbtype`: The database type. This must be either "DB2", "INFORMIX", "ODBC", "ORACLE", or "SYBASE". (For applications running on Netscape FastTrack Server, it must be "ODBC".)
- `serverName`: The name of the database server to which to connect. The server name typically is established when the database is installed. If in doubt, see your database or system administrator. For more information on this parameter, see the description of the `connect` method or the `DbPool` constructor in the *Server-Side JavaScript Reference*.
- `username`: The name of the user to connect to the database.
- `password`: The user's password.
- `databaseName`: The name of the database to connect to for the given server. If your database server supports the notion of multiple databases on a single server, supply the name of the database to use. If you provide an empty string, the default database is connected. For Oracle, ODBC, and DB2, you must always provide an empty string.

- `maxConnections`: (Optional) The number of connections to have available in the database pool. Remember that your database client license probably specifies a maximum number of connections. Do not set this parameter to a number higher than your license allows. If you do not supply this parameter for the `DbPool` object, its value is 1. If you do not supply this parameter for the `database` object, its value is whatever you specify in the Application Manager as the value for Built-in Maximum Database Connections when you install the application. (See “Installing a New Application” on page 61 for more information on this parameter.) See “Single-Threaded and Multithreaded Databases” on page 316 for things you should consider before setting this parameter.
- `commitflag`: (Optional) A Boolean value indicating whether to commit or to roll back open transactions when the connection is finalized. Specify `true` to commit open transactions and `false` to roll them back. If you do not supply this parameter for the `DbPool` object, its value is `false`. If you do not supply this parameter for the `database` object, its value is `true`.

For example, the following statement creates a new database pool of five connections to an Oracle database. With this pool, uncommitted transactions are rolled back:

```
pool = new DbPool ("ORACLE", "myserver1", "ENG", "pwd1", "", 5);
```

The `dbadmin` sample application lets you experiment with connecting to different databases as different users.

For many applications, you want to share the set of connections among clients or have a connection span multiple client requests. In these situations, you should make the connection on your application’s initial page. This avoids potential problems that can occur when individual clients make shared database connections.

However, for some applications each client needs to make its own connection. As discussed in “Sharing an Array of Connection Pools” on page 321, the clients may still be sharing objects. If so, be sure to use locks to control the data sharing, as discussed in “Sharing Objects Safely with Locking” on page 279.

The following table shows `DbPool` and database methods for managing the pool of connections. (The database object uses other methods, discussed later, for working with a database connection.) For a full description of these methods, see the *Server-Side JavaScript Reference*.

Table 15.2 `DbPool` and database methods for managing connection pools

<code>connect</code>	Connects the pool to a particular configuration of database and user.
<code>connected</code>	Tests whether the database pool and all of its connections are connected to a database.
<code>connection</code>	( <code>DbPool</code> only) Retrieves an available <code>Connection</code> object from the pool.
<code>disconnect</code>	Disconnects all connections in the pool from the database.
<code>majorErrorCode</code>	Major error code returned by the database server or ODBC.
<code>majorErrorMessage</code>	Major error message returned by the database server or ODBC.
<code>minorErrorCode</code>	Secondary error code returned by vendor library.
<code>minorErrorMessage</code>	Secondary error message returned by vendor library.

## Single-Threaded and Multithreaded Databases

LiveWire supports multithreaded access to your database. That is, it supports having more than one thread of execution access a single database at the same time. This support explains why it makes sense to have a connection pool with more than one connection in it. However, some vendor database libraries are not multithreaded. For those databases, it does not matter how many connections are in your connection pool; only one connection can access the database at a time. For information on which database libraries are single-threaded, see *Enterprise Server 3.x Release Notes*.

**Note** The guidelines below are crucial for single-threaded access. However, you should think about these points even for databases with multithreaded access.

A single-threaded database library has possible serious performance ramifications. Because only one thread can access the database at a time, all other threads must wait for the first thread to stop using the connection before they can access the database. If many threads want to access the database, each could be in for a long wait. You should consider the following when designing your database access:

- Keep your database interactions very short.  
Every thread must wait for every other thread. The shorter your interaction, the shorter the wait.
- Always release connections and close open cursors and stored procedures.  
You should do this anyway. In the case of a single-threaded database, however, it becomes absolutely essential to prevent needless waiting.
- Always use explicit transaction control.  
With explicit transaction control, it is clearer when you're done with a connection.
- Do not keep a connection open while waiting for input from the user.  
Users don't always complete what they start. If a user browses away from your application while it has an open connection, the system won't know when to release the connection. Unless you've implemented a scheme for retrieving idle connections (as discussed in "Retrieving an Idle Connection" on page 328), that connection could be tied up for a very long time, thus restricting other users from accessing the database.
- Do not keep a cursor or transaction open across multiple pages of your application.  
Any time a database interaction spans multiple pages of an application, the risk of a user not completing the transaction becomes even greater.

# Managing Connection Pools

At any given time, a connected `DbPool` or database object and all the connections in the pool are associated with a particular database configuration. That is, everything in a pool is connected to a particular database server, as a particular user, with a particular password, and to a particular database.

If your application always uses the same configuration, then you can easily use a single `DbPool` object or use the `database` object and connect exactly once. In this case, you should make the connection on your application's initial page.

If your application requires multiple configurations, either because it must connect to different databases, or to the same database as different users, or both, you need to decide how to manage those configurations.

If you use the `database` object and have multiple configurations, you have no choice. You must connect, disconnect, and reconnect the `database` object each time you need to change something about the configuration. You do so under the control of the client requests. In this situation, be sure you use locks, as discussed in “Sharing Objects Safely with Locking” on page 279, to gain exclusive access to the `database` object. Otherwise, another client request can disconnect the object before this client request is finished with it. Although you can use the `database` object this way, you're probably better off using `DbPool` objects.

If you use `DbPool` objects and have multiple configurations, you could still connect, disconnect, and reconnect the same `DbPool` object. However, with `DbPool` objects you have more flexibility. You can create as many pools as you need and place them under the control of the `project` object. (See Chapter 13, “Session Management Service,” for information on the `project` object.) Using multiple database pools is more efficient and is generally safer than reusing a single pool (either with the `database` object or with a single `DbPool` object).

In deciding how to manage your pools, you must consider two factors: how many different configurations you want your pools to be able to access, and whether a single connection needs to span multiple client requests. If you have a small number of possible configurations, you can create a separate pool for each configuration. “Sharing a Fixed Set of Connection Pools” on page 320 discusses this approach.

If you have a very large or unknown number of configurations (for example, if all users get their own database user ID), there are two situations to consider. If each connection needs to last for only one client request, then you can create individual database pools on a client page.

However, sometimes a connection must span multiple client requests (for example, if a single database transaction spans multiple client requests). Also, you may just not want to reconnect to the database on each page of the application. If so, you can create an array of pools that is shared. “Sharing an Array of Connection Pools” on page 321 discusses this approach.

Whichever approach you use, when you no longer need an individual connection in a pool, clean up the resources used by the connection so that it is available for another user. To do so, close all open cursors, stored procedures, and result sets. Release the connection back to the pool. (You don’t have to release the connection if you’re using the `database` object.)

If you do not release the connection, when you try to disconnect the pool, the system waits before actually disconnecting for one of two conditions to occur:

- You do release all connections
- The connections go out of scope and get collected by the garbage collector

If you create individual database pools for each user, be sure to disconnect the pool when you’re finished with it. For information on cursors, see “Manipulating Query Results with Cursors” on page 338. For information on stored procedures and result sets, see “Calling Stored Procedures” on page 354.

## Sharing a Fixed Set of Connection Pools

Frequently, an application shares a small set of connection pools among all users of the application. For example, your application might need to connect to three different databases, or it might need to connect to a single database using four different user IDs corresponding to four different departments. If you have a small set of possible connection configurations, you can create separate pools for each configuration. You use `DbPool` objects for this purpose.

In this case, you want the pool of connections to exist for the entire life of the application, not just the life of a client or an individual client request. You can accomplish this by creating each database pool as a property of the `project` object. For example, the application's initial page could contain these statements:

```
project.engpool = new DbPool ("ORACLE", "myserver1", "ENG",
    "pwd1", "", 5, true);
project.salepool = new DbPool ("INFORMIX", "myserver2", "SALES",
    "pwd2", "salsmktg", 2);
project.supppool = new DbPool ("SYBASE", "myserver3", "SUPPORT",
    "pwd3", "suppdb", 3, false);
```

These statements create three pools for different groups who use the application. The `project.eng` pool has five Oracle connections and commits any uncommitted transactions when a connection is released back to the pool. The `project.sales` pool has two Informix connections and rolls back any uncommitted transactions at the end of a connection. The `project.supp` pool has three Sybase connections and rolls back any uncommitted transactions at the end of a connection.

You should create this pool as part of the application's initial page. That page is run only when the application starts. On user-accessible pages, you don't create a pool, and you don't change the connection. Instead, these pages determine which group the current user belongs to and uses an already established connection from the appropriate pool. For example, the following code determines which database to use (based on the value of the `userGroup` property of the `request` object), looks up some information in the database and displays it to the user, and then releases the connection:

```
if (request.userGroup == "SALES") {
    salesconn = project.salepool.connection("A sales connection");
    salesconn.SQLTable ("select * from dept");
    salesconn.release();
}
```



Alternatively, you can choose to create the pool and change the connection on a user-accessible page. If you do so, you'll have to be careful that multiple users accessing that page at the same time do not interfere with each other. For example, only one user should be able to create the pool that will be shared by all users. For information on safe sharing of information, see “Sharing Objects Safely with Locking” on page 279.

## Sharing an Array of Connection Pools

“Sharing a Fixed Set of Connection Pools” on page 320 describes how you can use properties of the `project` object to share a fixed set of connection pools. This approach is useful if you know how many connection pools you will need at the time you develop the application and furthermore you need only a small number of connections.

For some applications, you cannot predict in advance how many connection pools you will need. For others, you can predict, but the number is prohibitively large. For example, assume that, for each customer who accesses your application, the application consults a user profile to determine what information to display from the database. You might give each customer a unique user ID for the database. Such an application requires each user to have a different set of connection parameters (corresponding to the different database user IDs) and hence a different connection pool.

You could create the `DbPool` object and connect and disconnect it on every page of the application. This works only if a single connection does not need to span multiple client requests. Otherwise, you can handle this situation differently.

For this application, instead of creating a fixed set of connection pools during the application's initial page or a pool on each client page, you create a single property of the `project` object that will contain an array of connection pools. The elements of that array are accessed by a key based on the particular user. At initialization time, you create the array but do not put any elements in the array (since nobody has yet tried to use the application), as shown here:

```
project.sharedPools = new Object();
```

The first time a customer starts the application, the application obtains a key identifying that customer. Based on the key, the application creates a `DbPool` pool object and stores it in the array of pools. With this connection pool, it can either reconnect on each page or set up the connection as described in

“Maintaining a Connection Across Requests” on page 325. The following code either creates the pool and or obtains the already created pool, makes sure it is connected, and then works with the database:

```
// Generate a unique index to refer to this client, if that
// hasn't already been done on another page. For information
// on the ssjs_generateClientID function, see
// "Uniquely Referring to the client Object" on page 255
if client.id == null {
    client.id = ssjs_generateClientID();
}

// If there isn't already a pool for this client, create one and
// connect it to the database.
project.lock();
if (project.sharedPools[client.id] == null) {
    project.sharedPools[client.id] = new DbPool ("ORACLE",
        "myserver", user, password, "", 5, false);
}
project.unlock();

// Set a variable to this pool, for convenience.
var clientPool = project.sharedPools[client.id];

// You've got a pool: see if it's connected. If not, try to
// connect it. If that fails, redirect to a special page to
// inform the user.
project.lock();
if (!clientPool.connected()) {
    clientPool.connect("ORACLE", "myserver", user, password,
        "", 5, false);
    if (!clientPool.connected()) {
        delete project.sharedPools[client.id];
        project.unlock();
        redirect("noconnection.htm");
    }
}
project.unlock();

// If you've got this far, you're successfully connected and
// can work with the database.
clientConn = clientPool.connection();
clientConn.SQLTable("select * from customers");
// ... more database operations ...

// Always release a connection when you no longer need it.
clientConn.release();
}
```

The next time the customer accesses the application (for example, from another page in the application), it uses the same code and obtains the stored connection pool and (possibly a stored `Connection` object) from the `project` object.

If you use `ssjs_generateClientID` and store the ID on the `client` object, you may need to protect against an intruder getting access to that ID and hence to sensitive information.

**Note** The `sharedConns` object used in this sample code is not a predefined JavaScript object. It is simply created by this sample and could be called anything you choose.

## Individual Database Connections

Once you've created a pool of connections, a client page can access an individual connection from the pool. If you're using the `database` object, the connection is implicit in that object; that is, you use methods of the `database` object to access the connection. If, however, you're using `DbPool` objects, a connection is encapsulated in a `Connection` object, which you get by calling a method of the `DbPool` object. For example, suppose you have this pool:

```
project.eng = new DbPool ("ORACLE", "myserver", "ENG", "pwd1", "", 5);
```

You can get a connection from the pool with this method call:

```
myconn = project.eng.connection ("My Connection", 60);
```

The parameters to this method are both optional. The first is a name for the connection (used for debugging); the second is an integer indicating a timeout period, in seconds. In this example, if the pool has an available connection, or if one becomes available within 60 seconds, that connection is assigned to the variable `myconn`. If no connection becomes available during this period, the method returns without a connection. For more information on waiting to get a connection from a pool, see “Waiting for a Connection” on page 327. For information on what to do if you don't get one, see “Retrieving an Idle Connection” on page 328.

When you have finished using a connection, return it to the pool by calling the `Connection` object's `release` method. (If you're using the `database` object, you do not have to release the connection yourself.) Before calling the `release` method, close all open cursors, stored procedures, and result sets.

When you call the `release` method, the system waits for these to be closed and then returns the connection to the database pool. The connection is then available to the next user. (For information on cursors, see “Manipulating Query Results with Cursors” on page 338. For information on stored procedures and result sets, see “Calling Stored Procedures” on page 354.)

Once you have a connection (either through the `database` object or a `Connection` object), you can interact with the database. The following table summarizes the database and connection methods for working with a single connection. The `database` object has other methods for managing a connection pool, discussed in “Managing Connection Pools” on page 318.

Table 15.3 database and Connection methods for working with a single connection

Method	Description
<code>cursor</code>	Creates a database cursor for the specified SQL <code>SELECT</code> statement.
<code>SQLTable</code>	Displays query results. Creates an HTML table for results of an SQL <code>SELECT</code> statement.
<code>execute</code>	Performs the specified SQL statement. Use for SQL statements other than queries.
<code>connected</code>	Returns <code>true</code> if the database pool (and hence this connection) is connected to a database.
<code>release</code>	(Connection only) Releases the connection back to its database pool.
<code>beginTransaction</code>	Begins an SQL transaction.
<code>commitTransaction</code>	Commits the current SQL transaction.
<code>rollbackTransaction</code>	Rolls back the current SQL transaction.
<code>storedProc</code>	Creates a stored-procedure object and runs the specified database stored procedure.
<code>majorErrorCode</code>	Major error code returned by the database server or ODBC.
<code>majorErrorMessage</code>	Major error message returned by the database server or ODBC.
<code>minorErrorCode</code>	Secondary error code returned by vendor library.
<code>minorErrorMessage</code>	Secondary error message returned by vendor library.

## Maintaining a Connection Across Requests

In some situations, you may want a single connection to span multiple client requests. That is, you might want to use the same connection on multiple HTML pages.

Typically, you use properties of the `client` object for information that spans client requests. However, the value of a `client` property cannot be an object. For that reason, you cannot store a pool of database connections in the `client` object. Instead, you use a pool of connections stored with the `project` object, managing them as described in this section. If you use this approach, you may want to encrypt user information for security reasons.

**Warning** Take special care with this approach because storing the connection in this way makes it unavailable for other users. If all the connections are unavailable, new requests wait until someone explicitly releases a connection or until a connection times out. This is especially problematic for single-threaded database libraries. (For information setting up connections so that they are retrieved when idle for a long time, see “Retrieving an Idle Connection” on page 328.)

In the following example, a connection and a transaction span multiple client requests. The code saves the connection as a property of the `sharedConns` object, which is itself a property of the `project` object. The `sharedConns` object is not a predefined JavaScript object. It is simply created by this sample and could have any name you choose.

Because the same pool is used by all clients, you should create the `sharedConns` object and create and connect the pool itself on the application’s initial page, with code similar to this:

```
project.sharedConns = new Object();
project.sharedConns.conns = new Object();
project.sharedConns.pool = new DbPool ("SYBASE", "sybaseserver",
    "user", "password", "sybdb", 10, false);
```

Then, on the first client page that accesses the pool, follow this strategy:

```
// Generate a unique index to refer to this client, if that hasn't
// already been done on another page.
if client.id == null {
    client.id = ssjs_generateClientID();
}

// Set a variable to this pool, for convenience.
var clientPool = project.sharedConns.pool;

// See whether the pool is connected. If not, redirect to a
// special page to inform the user.
project.lock();
if (!clientPool.connected()) {
    delete project.sharedConns.pool;
    project.unlock();
    redirect("noconnection.htm");
}
project.unlock();

// Get a connection from the pool and store it in the project object
project.sharedConns.conns[client.id] = clientPool.connection();
var clientConn = project.sharedConns.conns[client.id];

clientConn.beginTransaction();
cursor = clientConn.cursor("select * from customers", true);
// ... more database statements ...
cursor.close();
}
```

Notice that this page does not roll back or commit the transaction. The connection remains open and the transaction continues. (Transactions are discussed in “Managing Transactions” on page 348.) The second HTML page retrieves the connection, based on the value of `client.id` and continues working with the database as follows:

```
// Retrieve the connection.
var clientConn = project.sharedConns.conns[client.id];

// ... Do some more database operations ...
// In here, if the database operations succeed, set okay to 1.
// If there was a database error, set okay to 0. At the end,
// either commit or roll back the transaction on the basis of
// its value.
if (okay)
    clientConn.commitTransaction();
else
    clientConn.rollbackTransaction();

// Return the connection to the pool.
clientConn.release();
```

```
// Get rid of the object property value. You no longer need it.
delete project.sharedConns.conns[client.id];
```

In this sample, the `sharedConns` object stores a single `DbPool` object and the connections for that pool that are currently in use. Your situation could be significantly more complex. If you have a fixed set of database pools, you might predefine a separate object to store the connections for each pool. Alternatively, if you have an array of pools and each pool needs connections that span multiple requests, you need to create an array of objects, each of which stores a pool and an array of its connections. As another wrinkle, instead of immediately redirecting if the pool isn't connected, a client page might try to reestablish the connection.

If you use `ssjs_generateClientID` and store the ID in the `client` object, you may need to protect against an intruder getting access to that ID and hence to sensitive information.

## Waiting for a Connection

There are a fixed number of connections in a connection pool created with `DbPool`. If all connections are in use during an access attempt, then your application waits a specified timeout period for a connection to become free. You can control how long your application waits.

Assume that you've defined the following pool containing three connections:

```
pool = new DbPool ("ORACLE", "myserv", "user", "password", "", 3);
```

Further assume that three clients access the application at the same time, each using one of these connections. Now, a fourth client requests a connection with the following call:

```
myconnection = pool.connection();
```

This client must wait for one of the other clients to release a connection. In this case, because the call to `connection` does not specify a timeout period, the client waits indefinitely until a connection is freed, and then returns that connection.

You can specify a different timeout period by supplying arguments to the `connection` method. The second argument to the `connection` method is a timeout period, expressed in seconds. If you specify 0 as the timeout, the system waits indefinitely. For example, the following code has the connection wait only 30 seconds before timing out:

```
myconnection = pool.connection ("Name of Connection", 30);
```

If no connection becomes available within the specified timeout period, the method returns null, and an error message is set in the `minorErrorMessage` property. You can obtain this message by calling the `minorErrorMessage` method of `pool`. If your call to `connection` times out, you may want to free one by disconnecting an existing connection. For more information, see “Retrieving an Idle Connection” on page 328.

## Retrieving an Idle Connection

When your application requests a connection from a `DbPool` object, it may not get one. Your options at this point depend on the architecture of your application.

If each connection lasts only for the lifetime of a single client request, the unavailability of connections cannot be due to a user’s leaving an application idle for a significant period of time. It can only be because all the code on a single page of JavaScript has not finished executing. In this situation, you should not try to terminate connection that is in use and reuse it. If you terminate the connection at this time, you run a significant risk of leaving that thread of execution in an inconsistent state. Instead, you should make sure that your application releases each connection as soon as it is finished using it. If you don’t want to wait for a connection, you’ll have to present your user with another choice.

If, by contrast, a connection spans multiple client requests, you may want to retrieve idle connections. In this situation, a connection can become idle because the user did not finish a transaction. For example, assume that a user submits data on the first page of an application and that the data starts a multipage database transaction. Instead of submitting data for the continuation of the transaction on the next page, the user visits another site and never returns to this application. By default, the connection remains open and cannot be used by other clients that access the application.



You can manually retrieve the connection by cleaning up after it and releasing it to the database pool. To do so, write functions such as the following to perform these activities:

- `Bucket`: Define an object type (called `bucket` in this example) to hold a connection and a timestamp.
- `MarkBucket`: Mark a `bucket` object with the current timestamp.
- `RetrieveConnections`: Traverse an array of connections looking for `Connection` objects that haven't been accessed within a certain time limit and use `CleanBucket` (described next) to retrieve the object.
- `CleanBucket`: Close cursors (and possibly stored procedures and result sets), roll back or commit any open transaction, and return the connection back to the pool.

Your application could use these functions as follows:

1. When you get a new connection, call `Bucket` to create a `bucket` object.
2. On any page that accesses the connection, call `MarkBucket` to update the timestamp.
3. If the application times out trying to get a connection from the pool, call `RetrieveConnection` to look for idle connections, close any open cursors, commit or rollback pending transactions, and release idle connections back to the pool.
4. If a connection was returned to the pool, then try and get the connection from the pool.

Also, on each page where your application uses a connection, it needs to be aware that another thread may have disconnected the connection before this page was reached by this client.

**Creating a Bucket.** The bucket holds a connection and a timestamp. This sample constructor function takes a connection as its only parameter:

```
// Constructor for Bucket
function Bucket(c)
{
    this.connection = c;
    this.lastModified = new Date();
}
```

You call this function to create a bucket for the connection as soon as you get the connection from the connection pool. You might add other properties to the connection bucket. For instance, your application may contain a cursor that spans client requests. In this case, you could use a property to add the cursor to the bucket, so that you can close an open cursor when retrieving the connection. You store the cursor in the bucket at the time you create it, as shown in the following statement:

```
myBucket.openCursor =
  myBucket.connection.cursor("select * from customer", true);
```

**Marking the Bucket.** The `MarkBucket` function takes a `Bucket` object as a parameter and sets the `lastModified` field to the current time.

```
function MarkBucket(bucket)
{
    bucket.lastModified = new Date();
}
```

Call `MarkBucket` on each page of the application that uses the connection contained in the bucket. This resets `lastModified` to the current date and prevents the connection from appearing idle and hence ripe for retrieval.

**Retrieving Old Connections.** `RetrieveConnections` scans an array of `Bucket` objects, searching for connection buckets whose timestamp predates a certain time. If one is found, then the function calls `CleanBucket` (described next) to return the connection to the database pool.

```
// Retrieve connections idle for the specified number of minutes.
function RetrieveConnections(BucketArray, timeout)
{
    var i;
    var count = 0;
    var now;

    now = new Date();

    // Do this loop for each bucket in the array.
    for (i in BucketArray) {

        // Compute the time difference between now and the last
        // modified date. This difference is expressed in milliseconds.
        // If it is greater than the timeout value, then call the clean
        // out function.

        if ((now - i.lastModified)/60000) > timeout) {
            CleanBucket(i);

            // Get rid of the bucket, because it's no longer being used.
            delete i;
        }
    }
}
```

```

        count = count + 1;
    }
}
return count;
}

```

**Cleaning Up a Bucket.** Once it has been determined that a connection should be retrieved (with the `RetrieveConnections` function), you need a function to clean up the details of the connection and then release it back to the database pool. This sample function closes open cursors, rolls back open transactions, and then releases the connection.

```

function CleanBucket(bucket)
{
    bucket.openCursor.close();
    bucket.connection.rollbackTransaction();
    bucket.connection.release();
}

```

`CleanBucket` assumes that this bucket contains an open cursor and its connection has an open transaction. It also assumes no stored procedures or result sets exist. In your application, you may want to do some other checking.

**Pulling It All Together.** The following sample code uses the functions just defined to retrieve connections that haven't been referenced within 10 minutes. First, create a shared connections array and a database pool with five connections:

```

if ( project.sharedConns == null ) {
    project.sharedConns = new Object();
    project.sharedConns.pool = new DbPool ( "ORACLE", "mydb",
        "user", "password", "", 5, false);
    if ( project.sharedConns.pool.connected() ) {
        project.sharedConns.connections = new Object();
    }

    else {
        delete project.sharedConns;
    }
}

```

Now use the following code to try to get a connection. After creating the pool, generate a client ID and use that as an index into the connection array. Next, try to get a connection. If a timeout occurs, then call `RetrieveConnections` to return old connections to the pool. If `RetrieveConnections` returns a connection to the pool, try to get the connection again. If you still can't get a

connection, redirect to another page saying there are no more free connections. If a connection is retrieved, store it in a new connection bucket and store that connection bucket in the shared connections array.

```

if ( project.sharedConns != null ) {
    var pool = project.sharedConns.pool;

    // This code is run only if the pool is already connected.
    // If it is not, presumably you'd have code to connect.
    if ( pool.connected() == true ) {

        // Generate the client ID.
        client.id = ssjs_generateClientID();

        // Try to get a connection.
        var connection = pool.connection("my connection", 30);

        // If the connection is null, then none was available within
        // the specified time limit. Try and retrieve old connections.
        if (connection == null) {

            // Retrieve connections not used for the last 10 minutes.
            var count = RetrieveConnections(project.sharedConns, 10);

            // If count is nonzero, you made some connections available.
            if (count != 0){
                connection = pool.connection("my connection", 30);
                // If connection is still null, give up.
                if (connection == null)
                    redirect("nofreeconnections.htm");
            }
            else {
                // Give up.
                redirect("nofreeconnections.htm");
            }
        }

        // If you got this far, you have a connection and can proceed.
        // Put this connection in a new bucket, start a transaction,
        // get a cursor, store that in the bucket, and continue.
        project.sharedConns.connections[client.id] =
            new Bucket(connection);
        connection.beginTransaction();
        project.sharedConns.connections[client.id].cursor =
            connection.cursor("select * from customer", true);

        // Mark the connection bucket as used.
        MarkBucket(project.sharedConns.connections[client.id]);

        // Database statements.
        ...
    }
}

```

In the next page of the multipage transaction, perform more database operations on the connection. After the last database operation to the connection, mark the connection bucket:

```
var Bucket = project.sharedConns.connections[client.id];

if ( Bucket == null) {
    // Reconnect
}

else {

    // Interact with the database.
    ...

    // The last database operation on the page.
    row = Bucket.cursor.next();
    row.customerid = 666;
    Bucket.openCursor.insertRow("customer");

    // Mark the connection bucket as having been used on this page.
    MarkBucket(Bucket);
}
```



# Working with a Database

This chapter discusses working with DB2, Informix, ODBC, Oracle, or Sybase relational databases. It describes how to retrieve information from the database and use it in your application, how to work with database transactions, and how to execute database stored procedures.

Remember that if your application runs on Netscape FastTrack Server instead of Netscape Enterprise Server, it can access only databases on servers using the ODBC standard.

This chapter contains the following sections:

- Automatically Displaying Query Results
- Executing Arbitrary SQL Statements
- Manipulating Query Results with Cursors
- Managing Transactions
- Working with Binary Data
- Calling Stored Procedures

The LiveWire Database Service allows you to interact with a relational database in many ways. You can do all of the following:

- Perform database queries and have the runtime engine automatically format the results for you.
- Use cursors to perform database queries and present the results in an application-specific way or use the results in performing calculations.
- Use cursors to change information in your database.
- Use transactions to manage your database interactions.
- Perform SQL processing not involving cursors.
- Run database stored procedures.

For information on how to set up and manage your database connections, see Chapter 15, “Connecting to a Database.”

## Automatically Displaying Query Results

The simplest and quickest way to display the results of database queries is to use the `SQLTable` method of the `database` object or a `Connection` object. The `SQLTable` method takes an SQL `SELECT` statement and returns an HTML table. Each row and column in the query is a row and column of the table. The HTML table also has column headings for each column in the database table.

The `SQLTable` method does not give you control over formatting of the output. Furthermore, if that output contains a `Blob` object, that object does not display as an image. (For information on blobs, see “Working with Binary Data” on page 351.) If you want to customize the appearance of the output, use a database cursor to create your own display function. For more information, see “Manipulating Query Results with Cursors” on page 338.

As an example, if `myconn` is a `Connection` object, the following JavaScript statement displays the results of the database query in a table:

```
myconn.SQLTable("select * from videos");
```



The following is the first part of the table that could be generated by these statements:

Title	ID	Year	Category	Quantity	On Hand	Synopsis
A Clockwork Orange	1	1975	Science Fiction	5	3	Little Alex and his droogies stop by the Miloko bar for a refreshing libation before a wild night on the town.
Philadelphia Story	1	1940	Romantic Comedy			Katherine Hepburn and Cary Grant are reunited on the eve of her remarriage, with Jimmy Stewart for complications.

## Executing Arbitrary SQL Statements

The `execute` method of the database object or a `Connection` object enables an application to execute an arbitrary SQL statement. Using `execute` is referred to as performing **passthrough SQL**, because it passes SQL directly to the server.

You can use `execute` for any data definition language (DDL) or data manipulation language (DML) SQL statement supported by the database server. Examples include `CREATE`, `ALTER`, and `DROP`. While you can use it to execute any SQL statement, you cannot return data with the `execute` method.

Notice that `execute` is for performing standard SQL statements, not for performing extensions to SQL provided by a particular database vendor. For example, you cannot call the Oracle `describe` function or the Informix `load` function from the `execute` method.

To perform passthrough SQL statements, simply provide the SQL statement as the parameter to the `execute` method. For example, you might want to remove a table from the database that is referred to by the `project` object's `oldtable` property. To do so, you can use this method call:

```
connobj.execute("DROP TABLE " + project.oldtable);
```

**Important** When using `execute`, your SQL statement must strictly conform to the SQL syntax requirements of the database server. For example, some servers require each SQL statement to be terminated by a semicolon. For more information, see your database server documentation.

If you have not explicitly started a transaction, the single statement is committed automatically. For more information on transaction control, see “Managing Transactions” on page 348.

To perform some actions, such as creating or deleting a table, you may need to have privileges granted by your database administrator. Refer to your database server documentation for more information, or ask your database administrator.

# Manipulating Query Results with Cursors

In many situations, you do not simply want to display a table of query results. You may want to change the formatting of the result or even do arbitrary processing, rather than displaying it at all. To manipulate query results, you work with a database cursor returned by a database query. To create an instance of the `Cursor` class, call the database object’s or a `Connection` object’s `cursor` method, passing an SQL `SELECT` statement as its parameter.

You can think of a cursor as a virtual table, with rows and columns specified by the query. A cursor also implies the notion of a **current row**, which is essentially a pointer to a row in the virtual table. When you perform operations with a cursor, they usually affect the current row.

When finished, close the database cursor by calling its `close` method. A database connection cannot be released until all associated cursors have been closed. For example, if you call a `Connection` object’s `release` method and that connection has an associated cursor that has not been closed, the connection is not actually released until you close the cursor.

The following table summarizes the methods and properties of the `Cursor` class.

Table 16.1 `Cursor` properties and methods

Method or Property	Description
<code>colName</code>	Properties corresponding to each column in the cursor. The name of each <code>colName</code> property is the name of a column in the database.
<code>close</code>	Disposes of the cursor.
<code>columns</code>	Returns the number of columns in the cursor.

Table 16.1 `Cursor` properties and methods (Continued)

Method or Property	Description
<code>columnName</code>	Returns the name of a column in the cursor.
<code>next</code>	Makes the next row in the cursor the current row.
<code>insertRow</code>	Inserts a new row into the specified table.
<code>updateRow</code>	Updates records in the current row of the specified table.
<code>deleteRow</code>	Deletes the current row of the specified table.

For complete information on these methods, see the description of the `Cursor` class in the *Server-Side JavaScript Reference*.

## Creating a Cursor

Once an application is connected to a database, you can create a cursor by calling the `cursor` method of the associated database or `Connection` object. Creating the `Cursor` object also opens the cursor in the database. You do not need a separate open command. You can supply the following information when creating a `Cursor` object:

- An SQL `SELECT` statement supported by the database server. To ensure database independence, use SQL 89/92-compliant syntax. The cursor is created as a virtual table of the results of this SQL statement.
- An optional Boolean parameter indicating whether you want an updatable cursor. Use this parameter only if you want to change the content of the database, as described in “Changing Database Information” on page 346. It is not always possible to create an updatable cursor for every SQL statement; this is controlled by the database. For example, if the `SELECT` statement is `"select count(*) from videos"`, you cannot create an updatable cursor.

For example, the following statement creates a cursor for records from the `CUSTOMER` table. The records contain the columns `id`, `name`, and `city` and are ordered by the value of the `id` column.

```
custs = connobj.cursor ("select id, name, city
    from customer order by id");
```

This statement sets the variable `custs` to a `Cursor` object. The SQL query might return the following rows:

```
1 Sally Smith Suva
2 Jane Doe Cupertino
3 John Brown Harper's Ferry
```

You can then access this information using methods of the `custs` `Cursor` object. This object has `id`, `name`, and `city` properties, corresponding to the columns in the virtual table.

When you initially create a `Cursor` object, the pointer is positioned just before the first row in the virtual table. The following sections describe how you can get information from the virtual table.

You can also use the string concatenation operator (+) and string variables (such as `client` or `request` property values) when constructing a `SELECT` statement. For example, the following call uses a previously stored customer ID to further constrain the query:

```
custs = connobj.cursor ("select * from customer where id = "
    + client.customerID);
```

You can encounter various problems when you try to create a `Cursor` object. For example, if the `SELECT` statement in your call to the `cursor` method refers to a nonexistent table, the database returns an error and the `cursor` method returns null instead of a `Cursor` object. In this situation, you should use the `majorErrorCode` and `majorErrorMessage` methods to determine what error has occurred.

As a second example, suppose the `SELECT` statement refers to a table that exists but has no rows. In this case, the database may not return an error, and the `cursor` method returns a valid `Cursor` object. However, since that object has no rows, the first time you use the `next` method on the object, it returns `false`. Your application should check for this possibility.

## Displaying Record Values

When you create a cursor, it acquires a *colName* property for each named column in the virtual table (other than those corresponding to aggregate functions), as determined by the *SELECT* statement. You can access the values for the current row using these properties. In the example above, the cursor has properties for the columns *id*, *name*, and *city*. You could display the values of the first returned row using the following statements:

```
// Create the Cursor object.
custs = connobj.cursor ("select id, name, city
    from customer order by id");

// Before continuing, make sure a real cursor was returned
// and there was no database error.
if ( custs && (connobj.majorErrorCode() == 0) ) {

    // Get the first row
    custs.next();

    // Display the values
    write ("<B>Customer Name:</B> " + custs.name + "<BR>");
    write ("<B>City:</B> " + custs.city + "<BR>");
    write ("<B>Customer ID:</B> " + custs.id);

    //Close the cursor
    custs.close();
}
```

Initially, the current row is positioned before the first row in the table. The execution of the *next* method moves the current row to the first row. For example, suppose this is the first row of the cursor:

```
1 Sally Smith Suva
```

In this case, the preceding code displays the following:

**Customer Name:** Sally Smith  
**City:** Suva  
**Customer ID:** 1

You can also refer to properties of a *Cursor* object (or indeed any JavaScript object) as elements of an array. The zero-index array element corresponds to the first column, the one-index array element corresponds to the second column, and so on.

For example, you could use an index to display the same column values retrieved in the previous example:

```
write ("<B>Customer Name:</B> " + custs[1] + "<BR>");
write ("<B>City:</B> " + custs[2] + "<BR>");
write ("<B>Customer ID:</B> " + custs[0]);
```

This technique is particularly useful inside a loop. For example, you can create a Cursor object named `custs` and display its query results in an HTML table with the following code:

```
// Create the Cursor object.
custs = connobj.cursor ("select id, name, city
    from customer order by id");

// Before continuing, make sure a real cursor was returned
// and there was no database error.
if ( custs && (connobj.majorErrorCode() == 0) ) {
    write ("<TABLE BORDER=1>");
    // Display column names as headers.
    write("<TR>");
    i = 0;
    while ( i < custs.columns() ) {
        write("<TH>", custs.columnName(i), "</TH>");
        i++;
    }
    write("</TR>");

    // Display each row in the virtual table.
    while(custs.next()) {
        write("<TR>");
        i = 0;
        while ( i < custs.columns() ) {
            write("<TD>", custs[i], "</TD>");
            i++;
        }
        write("</TR>");
    }
    write ("</TABLE>");

    // Close the cursor.
    custs.close();
}
```

This code would display the following table:

ID	NAME	CITY
1	Sally Smith	Suva
2	Jane Doe	Cupertino
3	John Brown	Harper's Ferry

This example uses methods discussed in the following sections.

## Displaying Expressions and Aggregate Functions

`SELECT` statements can retrieve values that are not columns in the database, such as aggregate values and SQL expressions. For such values, the `Cursor` object does not have a named property. You can access these values only by using the `Cursor` object's property array index for the value.

The following example creates a cursor named `empData`, navigates to the row in that cursor, and then displays the value retrieved by the aggregate function `MAX`. It also checks to make sure the results from the database are valid before using them:

```
empData = connobj.cursor ("select min(salary), avg(salary),
    max(salary) from employees");
if ( empData && (connobj.majorErrorCode() == 0) ) {
    rowexists = empData.next();
    if (rowexists) { write("Highest salary is ", empData[2]); }
}
```

This second example creates a cursor named `empRows` to count the number of rows in the table, navigates to the row in that cursor, and then displays the number of rows, once again checking validity of the data:

```
empRows = connobj.cursor ("select count(*) from employees");
if ( empRows && (connobj.majorErrorCode() == 0) ) {
    rowexists = empRows.next();
    if (rowexists) { write ("Number of rows in table: ", empRows[0]); }
}
```

## Navigating with Cursors

Initially, the pointer for a cursor is positioned before the first row in the virtual table. Use the `next` method to move the pointer through the records in the virtual table. This method moves the pointer to the next row and returns `true` as long it found another row in the virtual table. If there is not another row, `next` returns `false`.

For example, suppose a virtual table has columns named `title`, `rentalDate`, and `dueDate`. The following code uses `next` to iterate through the rows and display the column values in a table:

```
// Create the cursor.
custs = connobj.cursor ("select * from customer");

// Check for validity of the cursor and no database errors.
if ( custs && (connobj.majorErrorCode() == 0) ) {

    write ("<TABLE>");

    // Iterate through rows, displaying values.
    while (custs.next()) {
        write ("<TR><TD>" + custs.title + "</TD>" +
            "<TD>" + custs.rentalDate + "</TD>" +
            "<TD>" + custs.dueDate + "</TD></TR>");
    }

    write ("</TABLE>");

    // Always close your cursors when finished!
    custs.close();
}
```

This code could produce output such as the following:

Clockwork Orange	6/3/97	9/3/97
Philadelphia Story	8/1/97	8/5/97

You cannot necessarily depend on your place in the cursor. For example, suppose you create a cursor and, while you're working with it, someone else adds a row to the table. Depending on the settings of the database, that row may appear in your cursor. For this reason, when appropriate (such as when updating rows) you may want your code to have tests to ensure it's working on the appropriate row.



## Working with Columns

The `columns` method of the `Cursor` class returns the number of columns in a cursor. This method takes no parameters:

```
custs.columns()
```

You might use this method if you need to iterate over each column in a cursor.

The `columnName` method of the `Cursor` class returns the name of a column in the virtual table. This method takes an integer as a parameter, where the integer specifies the ordinal number of the column, starting with 0. The first column in the virtual table is 0, the second is 1, and so on.

For example, the following expression assigns the name of the first column in the `custs` cursor to the variable `header`:

```
header = custs.columnName(0)
```

If your `SELECT` statement uses a wildcard (\*) to select all the columns in a table, the `columnName` method does not guarantee the order in which it assigns numbers to the columns. That is, suppose you have this statement:

```
custs = connobj.cursor ("select * from customer");
```

If the `customer` table has 3 columns, `ID`, `NAME`, and `CITY`, you cannot tell ahead of time which of these columns corresponds to `custs.columnName(0)`. (Of course, you are guaranteed that successive calls to `columnName` have the same result.) If the order matters to you, you can instead hard-code the column names in the select statement, as in the following statement:

```
custs = connobj.cursor ("select ID, NAME, CITY from customer");
```

With this statement, `custs.columnName(0)` is `ID`, `custs.columnName(1)` is `NAME`, and `custs.columnName(2)` is `CITY`.

## Changing Database Information

You can use an **updatable cursor** to modify a table based on the cursor's current row. To request an updatable cursor, add an additional parameter of `true` when creating the cursor, as in the following example:

```
custs = connobj.cursor ("select id, name, city from customer", true)
```

For a cursor to be updatable, the `SELECT` statement must be an updatable query (one that allows updating). For example, the statement cannot retrieve rows from more than one table or contain a `GROUP BY` clause, and generally it must retrieve key values from a table. For more information on constructing updatable queries, consult your database vendor's documentation.

When you use cursors to make changes to your database, you should always work inside an explicit transaction. You do so using the `beginTransaction`, `commitTransaction`, and `rollbackTransaction` methods, as described in “Managing Transactions” on page 348. If you do not use explicit transactions in these situations, you may get errors from your database.

For example, Informix and Oracle both return error messages if you use a cursor without an explicit transaction. Oracle returns `Error ORA-01002: fetch out of sequence`; Informix returns `Error -206: There is no current row for UPDATE/DELETE cursor`.

As mentioned in “Navigating with Cursors” on page 344, you cannot necessarily depend on your position in the cursor. For this reason, when making changes to the database, be sure to test that you're working on the correct row before changing it.

Also, remember that when you create a cursor, the pointer is positioned before any of the rows in the cursor. So, to update a row, you must call the `next` method at least once to establish the first row of the table as the current row. Once you have a row, you can assign values to columns in the cursor.

The following example uses an updatable cursor to compute the bonus for salespeople who met their quota. It then updates the database with this information:

```
connobj.beginTransaction ();

emps = connobj.cursor(
    "select * from employees where dept='sales'", true);

// Before proceeding make sure the cursor was created and
// there was no database error.
if ( emps && (connobj.majorErrorCode() == 0) ) {

    // Iterate over the rows of the cursor, updating information
    // based on the return value of the metQuota function.
    while ( emps.next() ) {
        if (metQuota (request.quota, emps.sold)) {
            emps.bonus = computeBonus (emps.sold);
        }
        else emps.bonus = 0;
        emps.updateRow ("employees");
    }

    // When done, close the cursor and commit the transaction.
    emps.close();
    connobj.commitTransaction();
}
else {
    // If there wasn't a cursor to work with, roll back the transaction.
    connobj.rollbackTransaction();
}
```

This example creates an updatable cursor of all employees in the Sales department. It iterates over the rows of that cursor, using the user-defined JavaScript function `metQuota` to determine whether or not the employee met quota. This function uses the value of `quota` property of the `request` object (possibly set in a form on a client page) and the `sold` column of the cursor to make this determination. The code then sets the bonus appropriately and calls `updateRow` to modify the `employees` table. Once all rows in the cursor have been accessed, the code commits the transaction. If no cursor was returned by the call to the `cursor` method, the code rolls back the transaction.

In addition to the `updateRow` method, you can use the `insertRow` and `deleteRow` methods to insert a new row or delete the current row. You do not need to assign values when you use `deleteRow`, because it simply deletes an entire row.

When you use `insertRow`, the values you assign to columns are used for the new row. If you have previously called the cursor's `next` method, then the values of the current row are used for any columns without assigned values; otherwise, the unassigned columns are null. Also, if some columns in the table are not in the cursor, then `insertRow` inserts null in these columns. The location of the inserted row depends on the database vendor library. If you need to access the row after you call the `insertRow` method, you must first close the existing cursor and then open a new cursor.

**Note** DB2 has a `Time` data type. JavaScript does not have a corresponding data type. For this reason, you cannot update rows with values that use the DB2 `Time` data type

## Managing Transactions

A **transaction** is a group of database actions that are performed together. Either all the actions succeed together or all fail together. When you apply all actions, making permanent changes to the database, you are said to **commit** a transaction. You can also **roll back** a transaction that you have not committed; this cancels all the actions.

Transactions are important for maintaining data integrity and consistency. Although the various database servers implement transactions slightly differently, the LiveWire Database Service provides the same methods for transaction management with all databases. Refer to the database vendor documentation for information on data consistency and isolation levels in transactions.

You can use explicit transaction control for any set of actions. For example, actions that modify a database should come under transaction control. These actions correspond to SQL `INSERT`, `UPDATE`, and `DELETE` statements. Transactions can also be used to control the consistency of the data you refer to in your application.

For most databases, if you do not control transactions explicitly, the runtime engine uses the underlying database's autocommit feature to treat each database statement as a separate transaction. Each statement is either committed or rolled back immediately, based on the success or failure of the individual statement. Explicitly managing transactions overrides this default behavior.

In some databases, such as Oracle, autocommit is an explicit feature that LiveWire turns on for individual statements. In others, such as Informix, autocommit is the default behavior when you do not create a transaction. In general, LiveWire hides these differences and puts an application in autocommit mode whenever the application does not use `beginTransaction` to explicitly start a transaction.

For Informix ANSI databases, LiveWire does not use autocommit. For these databases, an application always uses transactions even if it never explicitly calls `beginTransaction`. The application must use `commitTransaction` or `rollbackTransaction` to finish the transaction.

**Note** You are strongly encouraged to use explicit transaction control any time you make changes to a database. This ensures that the changes succeed or fail together. In addition, any time you use updatable cursors, you should use explicit transactions to control the consistency of your data between the time you read the data (with `next`) and the time you change it (with `insertRow`, `updateRow`, or `deleteRow`). As described in “Changing Database Information” on page 346, using explicit transaction control with updatable cursors is necessary to avoid errors in some databases such as Oracle and Informix.

## Using the Transaction-Control Methods

Use the following methods of the `database` object or a `Connection` object to explicitly manage transactions:

- `beginTransaction` starts a new transaction. All actions that modify the database are grouped with this transaction, known as the **current transaction**.
- `commitTransaction` commits the current transaction. This method attempts to commit all the actions since the last call to `beginTransaction`.
- `rollbackTransaction` rolls back the current transaction. This method undoes all modifications since the last call to `beginTransaction`.

Of course, if your database does not support transactions, you cannot use them. For example, an Informix database created using the `NO LOG` option does not support transactions, and you will get an error if you use these methods.

The LiveWire Database Service does not support nested transactions. If you call `beginTransaction` multiple times before committing or rolling back the first transaction you opened, you'll get an error.

For the database object, the maximum scope of a transaction is limited to the current client request (HTML page) in the application. If the application exits the page before calling the `commitTransaction` or `rollbackTransaction` method, then the transaction is automatically either committed or rolled back, based on the setting of the `commitflag` parameter provided when you connected to the database.

For `Connection` objects, the scope of a transaction is limited to the lifetime of that object. If you release the connection or close the pool of connections before calling the `commitTransaction` or `rollbackTransaction` method, then the transaction is automatically either committed or rolled back, based on the setting of the `commitflag` parameter provided when you made the connection, either with the `connect` method or in the `DbPool` constructor.

If there is no current transaction (that is, if the application has not called `beginTransaction`), calls to `commitTransaction` and `rollbackTransaction` can result in an error from the database.

You can set your transaction to work at different levels of granularity. The example described in “Changing Database Information” on page 346 creates a single transaction for modifying all rows of the cursor. If your cursor has a small number of rows, this approach is sensible.

If, however, your cursor returns thousands of rows, you may want to process the cursor in multiple transactions. This approach can both cut down the transaction size and improve the concurrency of access to that information.

If you do break down your processing into multiple transactions, be certain that a call to `next` and an associated call to `updateRow` or `deleteRow` happen within the same transaction. If you get a row in one transaction, finish that transaction, and *then* attempt to either update or delete the row, you may get an error from your database.

How you choose to handle transactions depends on the goals of your application. You should refer to your database vendor documentation for more information on how to use transactions appropriately for that database type.

# Working with Binary Data

Binary data for multimedia content such as an image or sound is stored in a database as a binary large object (BLOB). You can use one of two techniques to handle binary data in JavaScript applications:

- Store filenames in the database and keep the data in separate files.
- Store the data in the database as BLOBs and access it with `Blob` class methods.

If you do not need to keep BLOB data in a database, you can store the filenames in the database and access them in your application with standard HTML tags. For example, if you want to display an image for each row in a database table, you could have a column in the table called `imageFileName` containing the name of the desired image file. You could then use this HTML expression to display the image for each row:

```
<IMG SRC='mycursor.imageFileName'>
```

As the cursor navigates through the table, the name of the file in the `IMG` tag changes to refer to the appropriate file.

If you need to manipulate actual binary data in your database, the JavaScript runtime engine recognizes when the value in a column is BLOB data. That is, when the software creates a `Cursor` object, if one of the database columns contains BLOB data, the software creates a `Blob` object for the corresponding value in the `Cursor` object. You can then use the `Blob` object's methods to display that data. Also, if you want to insert BLOB data into a database, the software provides a global function for you to use.

The following table outlines the methods and functions for working with BLOB data.

Table 16.2 Methods and functions for working with Blobs

Method or Function	Description
<code>blobImage</code>	Method to use when displaying BLOB data stored in a database. Returns an HTML <code>IMG</code> tag for the specified image type (GIF, JPEG, and so on).
<code>blobLink</code>	Method to use when creating a link that refers to BLOB data with a hyperlink. Returns an HTML hyperlink to the BLOB.
<code>blob</code>	Global function to use to insert or update a row containing BLOB data. Assigns BLOB data to a column in a cursor.

The `blobImage` method fetches a BLOB from the database, creates a temporary file of the specified format, and generates an HTML `IMG` tag that refers to the temporary file. The runtime engine removes the temporary file after the page is generated and sent to the client.

The `blobLink` method fetches BLOB data from the database, creates a temporary file, and generates an HTML hypertext link to the temporary file. The runtime engine removes the temporary file after the user clicks the link or 60 seconds after the request has been processed.

The following example illustrates using `blobImage` and `blobLink` to create temporary files. In this case, the `FISHTBL` table has four columns: an ID, a name, and two images. One of these is a small thumbnail image; the other is a larger image. The example code writes HTML for displaying the name, the thumbnail, and a link to the larger image.

```

cursor = connobj.cursor ("select * from fishtbl");

if ( cursor && (connobj.majorErrorCode() == 0) ) {
    while (cursor.next()) {
        write (cursor.name);
        write (cursor.picture.blobImage("gif"));
        write (cursor.picture.blobLink("image\gif", "Link" + cursor.id));
        write ("<BR>");
    }
    cursor.close();
}

```



If `FISHTBL` contains rows for four fish, the example could produce the following HTML:

```
Cod <IMG SRC="LIVEWIRE_TEMP9">
  <A HREF="LIVEWIRE_TEMP10">Link1 </A> <BR>
Anthia <IMG SRC="LIVEWIRE_TEMP11">
  <A HREF="LIVEWIRE_TEMP12">Link2 </A> <BR>
Scorpion <IMG SRC="LIVEWIRE_TEMP13">
  <A HREF="LIVEWIRE_TEMP14">Link3 </A> <BR>
Surgeon <IMG SRC="LIVEWIRE_TEMP15">
  <A HREF="LIVEWIRE_TEMP16">Link4 </A> <BR>
```

If you want to add BLOB data to a database, use the `blob` global function. This function assigns BLOB data to a column in an updatable cursor. As opposed to `blobImage` and `blobLink`, `blob` is a top-level function, not a method.

The following statements assign BLOB data to one of the columns in a row and then update that row in the `FISHTBL` table of the database. The cursor contains a single row.

```
// Begin a transaction.
database.beginTransaction();

// Create a cursor.
fishCursor = database.cursor ("select * from fishtbl where
    name='Harlequin Ghost Pipefish'", true);

// Make sure cursor was created.
if ( fishCursor && (database.majorErrorCode() == 0) ) {

    // Position the pointer on the row.
    rowexists = fishCursor.next();

    if ( rowexists ) {

        // Assign the blob data.
        fishCursor.picture = blob ("c:\\data\\fish\\photo\\pipe.gif");

        // Update the row.
        fishCursor.updateRow ("fishtbl");

        // Close the cursor and commit the changes.
        fishCursor.close();
        database.commitTransaction();
    }
    else {
        // Close the cursor and roll back the transaction.
        fishCursor.close();
        database.rollbackTransaction();
    }
}
else {
```

```
// Never got a cursor; rollback the transaction.
database.rollbackTransaction();
}
```

Remember that the backslash (\) is the escape character in JavaScript. For this reason, you must use two backslashes in NT filenames, as shown in the example.

## Calling Stored Procedures

Stored procedures are an integral part of operating and maintaining a relational database. They offer convenience by giving you a way to automate processes that you do often, but they offer other benefits as well:

- *Limited access.* You can limit access to a sensitive database by giving users access only through a stored procedure. A user has access to the data, but only within the stored procedure. Any other access is denied.
- *Data integrity.* Stored procedures help you make sure that information is provided and entered in a consistent way. By automating complicated transactions, you can reduce the possibility of user error.
- *Efficiency.* A stored procedure is compiled once, when executed for the first time. Later executions run faster because they skip the compilation step. This also helps lighten the load on your network, because the stored procedure code is downloaded only once.

The LiveWire Database Service provides two classes for working with stored procedures, `Stproc` and `Resultset`. With the methods of these classes you can call a stored procedure and manipulate the results of that procedure.

## Exchanging Information

Stored procedures work differently for the various databases supported by the LiveWire Database Service. The most important distinction for LiveWire is how you pass information to and from the stored procedure in a JavaScript application. You always use input parameters to the stored procedure to pass information into a stored procedure.

However, conceptually there are several distinct ways you might want to retrieve information from a stored procedure. Not every database vendor lets you retrieve information in all of these ways.

## Result Sets

A stored procedure can execute one or more `SELECT` statements, retrieving information from the database. You can think of this information as a virtual table, very similar to a read-only cursor. (For information on cursors, see “Manipulating Query Results with Cursors” on page 338.)

LiveWire uses an instance of the `ResultSet` class to contain the rows returned by a single `SELECT` statement of a stored procedure. If the stored procedure allows multiple `SELECT` statements, you get a separate `ResultSet` object for each `SELECT` statement. You use the `resultSet` method of the `Stproc` class to obtain a result set object and then you use that object’s methods to manipulate the result set.

Different database vendors return a result set in these varying ways:

- Sybase stored procedures can directly return the result of executing one or more `SELECT` statements.
- Informix stored procedures can have multiple return values. Multiple return values are like the columns in a single row of a table, except that these columns are not named. In addition, if you use the `RESUME` feature, the stored procedure can have a set of these multiple return values. This set is like the rows of a table. LiveWire creates a single result set to contain this virtual table.
- Oracle stored procedures use ref cursors to contain the rows returned by a `SELECT` statement. You can open multiple ref cursors in an Oracle stored procedure to contain rows returned by several `SELECT` statements. LiveWire creates a separate `ResultSet` object for each ref cursor.
- DB2 stored procedures use open cursors to return result sets.

## Output and Input/Output Parameters

In addition to standard input parameters, some database vendors allow other types of parameters for their stored procedures. Output parameters store information on return from the procedure and input/output parameters both pass in information and return information.

For most databases, you use the `outParamCount` and `outParameters` methods of the `Stproc` class to access output and input/output parameters. However, Informix does not allow output or input/output parameters. Therefore, you should not use the `outParamCount` and `outParameters` methods with Informix stored procedures.

## Return Values

Seen as a simple function call, a stored procedure can have a return value. For Oracle and Sybase, this return value is in addition to any result sets it returns.

You use the `returnValue` method of the `Stproc` class to access the return value. However, the return values for Informix stored procedures are used to generate its result set. For this reason, `returnValue` always returns null for Informix stored procedures. In addition, return values are not available for ODBC and DB2 stored procedures.

## Steps for Using Stored Procedures

Once you have a database connection, the steps for using a stored procedure in your application vary slightly for the different databases:

1. (DB2 only) Register the stored procedure in the appropriate system tables. (You do this outside of JavaScript.)
2. (DB2, ODBC, and Sybase) Define a prototype for your stored procedure.
3. (All databases) Execute the stored procedure.
4. (All databases) Create a `resultSet` object and get the data from that object.
5. (DB2, ODBC, and Sybase) Complete the execution by accessing the return value.
6. (DB2, ODBC, Oracle, and Sybase) Complete the execution by getting the output parameters.

Notice that for several databases you can complete execution of your stored procedure either by getting the return value or by accessing the output parameters. Once you have done either of these things, you can no longer work with any result sets created by execution of the stored procedure.

The following sections describe each of these steps in more detail.

## Registering the Stored Procedure

This step applies only to DB2.

DB2 has various system tables in which you can record your stored procedure. In general, entering a stored procedure in these tables is optional. However, to use your stored procedure with LiveWire, you must make entries in these tables. You perform this step outside of the JavaScript application.

For DB2 common server, you must create the `DB2CLI.PROCEDURES` system table and enter your DB2 stored procedures in it. `DB2CLI.PROCEDURES` is a pseudo-catalog table.

If your DB2 is for IBM MVS/EA version 4.1 or later, you must define the name of your stored procedures in the `SYSIBM.SYSPROCEDURES` catalog table.

Remember you use C, C++, or another source language to write a DB2 stored procedure. The data types you use with those languages do not match the data types available in DB2. Therefore, when you add the stored procedure to `DB2CLI.PROCEDURES` or `SYSIBM.SYSPROCEDURES`, be sure to record the corresponding DB2 data type for the stored procedure parameters and not the data types of the source language.

For information on DB2 data types and on how to make entries in these tables, see your DB2 documentation.

## Defining a Prototype for a Stored Procedure

This step is relevant only for DB2, ODBC, and Sybase stored procedures, both user-defined and system stored procedures. You do not need to define a prototype for stored procedures for Oracle or Informix databases.

For DB2, ODBC, and Sybase, the software cannot determine at runtime whether a particular parameter is for input, for output, or for both. Consequently, after you connect to the database, you must create a prototype providing information about the stored procedure you want to use, using the `storedProcArgs` method of the database or `DbPool` object.

You need exactly one prototype for each stored procedure in your application. The software ignores additional prototypes for the same stored procedure.

In the prototype, you provide the name of the stored procedure and the type of each of its parameters. A parameter must be for input (`IN`), output (`OUT`), or input and output (`INOUT`). For example, to create a prototype for a stored procedure called `newhire` that has two input parameters and one output parameter, you could use this method call:

```
poolobj.storedProcArgs("newhire", "IN", "IN", "OUT");
```

## Executing the Stored Procedure

This step is relevant to all stored procedures.

To execute a stored procedure, you create a `Stproc` object using the database or `Connection` object's `storedProc` method. Creating the object automatically invokes the stored procedure. When creating a stored-procedure object, you specify the name of the procedure and any parameters to the procedure.

For example, assume you have a stored procedure called `newhire` that takes one string and one integer parameter. The following method call creates the `spObj` stored-procedure object and invokes the `newhire` stored procedure:

```
spObj = connobj.storedProc("newhire", "Fred Jones", 1996);
```

In general, you must provide values for all input and input/output parameters to the stored procedure. If a stored procedure has a default value defined for one of its parameters, you can use the `"/Default/"` directive to specify that default value. Similarly, if a stored procedure can take a null value for one of its parameters, you can specify the null value either with the `"/Null/"` directive or by passing in the null value itself.

For example, assume the `demosp` stored procedure takes two string parameters and one integer parameter. You could supply all the parameters as follows:

```
spobj = connobj.storedProc("demosp", "Param_1", "Param_2", 1);
```

Alternatively, to pass null for the second parameter and to use the default value for third parameter, you could use either of these statements:

```
spobj = connobj.storedProc("demosp", "Param_1", "/Null/", "/Default/");
spobj = connobj.storedProc("demosp", "Param_1", null, "/Default/");
```

**Note** On Informix, default values must occur only after all specified values. For example, you cannot use `/Default/` for the second parameter of a stored procedure and then specify a value for the third parameter.

You can also use the `"/Default/"` and `"/Null/"` directives for input/output parameters.

An Oracle stored procedure can take ref cursors as input/output or output parameters. For example, assume you have an Oracle stored procedure named `procl` that takes four parameters: a ref cursor, an integer value, another ref cursor, and another integer value. The call to that stored procedure from SQL Plus might look as follows:

```
execute procl (refcursor1, 3, refcursor2, 5);
```

When you call this stored procedure from within a JavaScript application, however, you do not supply the ref cursor parameters. Instead, the equivalent call would be:

```
spobj = connobj.storedProc("procl", 3, 5);
```

For information on output parameters, see “Working with Output Parameters” on page 367. Output parameters cannot be null; however, you can assign a null value to input or input/output parameters.

The following table summarizes the methods of a stored-procedure object.

Table 16.3 Stproc methods

Method	Description
resultSet	Returns the next result set for the stored procedure. For Informix, you can have zero or one result set. For other databases, you can have zero, one, or more result sets.
returnValue	Retrieves the return value of the stored procedure. For Informix, DB2, and ODBC, this method always returns null.
outParameters	Returns the specified output parameter. Because Informix stored procedures do not use output parameters, do not use this method with Informix.
outParamCount	Returns the number of output parameters. For Informix, this method always returns 0, because Informix stored procedures do not use output parameters.

## Working with Result Sets

This step is relevant for all stored procedures.

As described in “Result Sets” on page 355, different databases returns result sets in different ways. For example, assume you have the `CUSTINFO` table with the columns `id`, `city`, and `name`. In Sybase, you could use this stored procedure to get the first 200 rows of the table:

```
create proc getcusts as
begin
    select id, name, city from custinfo where custno < 200
end
```

If `CUSTINFO` were an Informix table, the equivalent Informix stored procedure would be this:

```
create procedure getcusts returning int, char(15), char(15);
define rcity, rname char (15);
define i int;

foreach
    select id, name, city into i, rname, rcity
    from custinfo
    where id < 200;
```



```

        return i, rname, rcity with resume;
    end foreach;
end procedure;
```

If CUSTINFO were an Oracle table, the equivalent Oracle stored procedure would be:

```

create or replace package orapack as
    type custcurtype is ref cursor return custinfo%rowtype
end orapack;

create or replace custresultset (custcursor inout orapack.custcurtype)
as begin
    open custcursor for select id, name, city from custinfo
        where id < 200
end custresultset;
```

In all cases, you create a `resultSet` object to retrieve the information from the stored procedure. You do so by using the stored-procedure object's `resultSet` method, as follows:

```
resObj = spObj(resultSet());
```

As for `Cursor` objects, `resultSet` objects have a current row, which is simply the row being pointed to in the result set. Initially, the pointer is positioned before the first row of the result set. To see the values in the rows of the result set, you use the `next` method to move the pointer through the rows in the result set, as shown in the following example:

```

spobj = connobj.storedProc("getcusts");

if ( spobj && (connobj.majorErrorCode() == 0) ) {
    // Creates a new resultSet object.
    resobj = spobj(resultSet());

    // Make sure you got a result set before continuing.
    if ( resobj && (connobj.majorErrorCode() == 0) ) {
        // Initially moves the resultSet object pointer to the first
        // result set row and then loops through the rows.
        while (resObj.next())
        {
            write("<TR><TD>" + resObj.name + "</TD>");
            write("<TD>" + resObj.city + "</TD>");
            write("<TD>" + resObj.id + "</TD></TR>");
        }
        resobj.close();
    }
}
```

As long as there is another row in the result set, the `next` method returns `true` and moves the pointer to the next row. When the pointer reaches the last row in the result set, the `next` method returns `false`.

The preceding example works for a Sybase stored procedure. In that case, the `resultSet` object contains a named property for each column in the result set. For Informix and DB2 stored procedures, by contrast, the object does not contain named columns. In this case, you can get the values by referencing the column position. So, for Informix and DB2, you would use this code to display the same information:

```
spobj = connobj.storedProc("getcusts");

if ( spobj && (connobj.majorErrorCode() == 0) ) {

    // Creates a new resultSet object.
    resobj = spobj.resultSet();

    // Make sure you got a result set before continuing.
    if ( resobj && (connobj.majorErrorCode() == 0) ) {

        // Initially moves the resultSet object pointer to the first
        // result set row and then loops through the rows.
        while (resObj.next())
        {
            write("<TR><TD>" + resObj[1] + "</TD>");
            write("<TD>" + resObj[2] + "</TD>");
            write("<TD>" + resObj[0] + "</TD></TR>");
        }
        resobj.close();
    }
}
```

You can use the column position for result sets with any database, not just with Informix and DB2. You can use the column name for stored procedures for all database types other than Informix or DB2.

## Multiple Result Sets

A Sybase, Oracle, DB2, or ODBC stored procedure can create multiple result sets. If it does, the stored procedure provides one `resultSet` object for each. Suppose your stored procedure executes these SQL statements:

```
select name from customers where id = 6767
select * from orders where id = 6767
```

You could use the multiple `resultSet` objects generated by these statements as follows:

```
// This statement is needed for DB2, ODBC, and Sybase.
poolobj.storedProcArgs("GetCustOrderInfo","IN");

spobj = connobj.storedProc("GetCustOrderInfo",6767);

if ( spobj && (connobj.majorErrorCode() == 0) ) {
    resobj1 = spobj.resultSet();
    // Make sure result set exists before continuing.
    if ( resobj1 && (connobj.majorErrorCode() == 0) ) {
        // This first result set returns only one row.
        // Make sure that row contains data.
        rowexists = resobj1.next();
        if ( rowexists )
            write("<P>Customer " + resobj1.name +
                " has the following orders:</P>");
        resobj1.close();

        // The second result set returns one row for each order placed
        // by the customer. Make sure the rows have data.
        resobj2 = spobj.resultSet();
        var i = 0;

        if ( resobj2 && (connobj.majorErrorCode() == 0) ) {
            write("\nOrder# Quantity Total</P>");
            while(resobj2.next()) {
                write(resobj2.orderno + " " + resobj2.quantity
                    + " " + resobj2.Totalamount + "</P>");
                i++;
            }
            resobj2.close();
            write("Customer has " + i + " orders.</P>");
        }
        else write("Customer has no orders.</P>");
    }
}

spobj.close();
```

For an example of using multiple Oracle ref cursors in a stored procedure, see the description of the `ResultSet` class in the *Server-Side JavaScript Reference*.

## Result Set Methods and Properties

The following table summarizes the methods and properties of the `ResultSet` class.

Table 16.4 `ResultSet` methods and properties

Method or Property	Description
<code>columnName</code>	Properties corresponding to each of the columns in the result set. The name of each property is the name of the column in the database. Since Informix and DB2 stored procedures do not return named columns, these properties are not created for Informix or DB2 stored procedures.
<code>columns</code>	Returns the number of columns in the result set. For Informix, this method returns the number of return values for a single row.
<code>columnName</code>	Returns the name of a column in the result set. Because Informix and DB2 stored procedures do not have associated column names, do not use this method for stored procedures for those databases.
<code>close</code>	Disposes of the <code>ResultSet</code> object.
<code>next</code>	Makes the next row in the result set the current row. Returns <code>false</code> if the current row is the last row in the result set; otherwise, returns <code>true</code> .

A `resultSet` object is a read-only, sequential-style object. For this reason, the class does not have the `insertRow`, `deleteRow`, and `updateRow` methods defined for `Cursor` objects.

## When You Can Use Result Sets

A `resultSet` object is not valid indefinitely. In general, once a stored procedure starts, no interactions are allowed between the database client and the database server until the stored procedure has completed. In particular, there are three circumstances that cause a result set to be invalid.

1. If you create a result set as part of a transaction, you must finish using the result set during that transaction. Once you either commit or roll back the transaction, you can't get any more data from a result set, and you can't get any additional result sets. For example, the following code is illegal:

```
database.beginTransaction();
spobj = database.storedProc("getcusts");
resobj = spobj.resultSet();
database.commitTransaction();
// Illegal! Result set no longer valid!
coll = resobj[0];
```

2. For Sybase, ODBC, and DB2, you must retrieve `resultSet` objects before you call a stored-procedure object's `returnValue` or `outParameters` methods. Once you call either of these methods, you can't get any more data from a result set, and you can't get any additional result sets. See "Working with Return Values" on page 366, for more information about these methods.

```
spobj = database.storedProc("getcusts");
resobj = spobj.resultSet();
retval = spobj.returnValue();
// Illegal! Result set no longer valid!
coll = resobj[0];
```

3. For Sybase, you must retrieve `resultSet` objects before you call the `cursor` or `SQLTable` method of the associated connection. Once you call `cursor` or `SQLTable`, the result set is no longer available. For example, the following code is illegal:

```
spobj = database.storedProc("getcusts");
resobj = spobj.resultSet();
curobj = database.cursor ("select * from orders");
// Illegal! The result set is no longer available!
coll = resobj[0];
```

4. For ODBC, a slightly different restriction holds. Again, you must work with the `resultSet` objects before you call the associated connection's `cursor` or `SQLTable` method. For ODBC, if you get a cursor, then access the result set, and then use the cursor, the `Cursor` object is no longer available. For example, the following code is illegal:

```
spbobj = database.storedProc("getcusts");
resobj = spobj.resultSet();
curobj = database.cursor ("select * from orders");
coll = resobj[0];
// Illegal! The cursor is no longer available.
curobj.next();
```

## Working with Return Values

This step is relevant to Sybase and Oracle stored procedures. For Informix, ODBC, and DB2 stored procedures, the `returnValue` method always returns null.

If your stored procedure has a return value, you can access that value with the `returnValue` method.

On DB2, ODBC, and Sybase, you must use stored procedures and cursors sequentially. You cannot intermix them. For this reason, you must let the system know that you have finished using the stored procedure before you can work with a cursor. You do this by calling the `returnValue` method of the stored procedure object. This method provides the stored procedure's return value (if it has one) and completes the execution of the stored procedure. You should also close all objects related to stored procedures when you have finished using them.

**Note** For DB2, ODBC, and Sybase, you must retrieve `resultSet` objects before you call the `returnValue` method. Once you call `returnValue`, you can't get any more data from a result set, and you can't get any additional result sets. You should call `returnValue` after you have processed the result set and before you retrieve the output parameters.

## Working with Output Parameters

This step is relevant to Sybase, Oracle, DB2, or ODBC stored procedures. For Informix stored procedures, the methods discussed here are not applicable.

To determine how many output parameters the procedure has (including both output and input/output parameters), you use the `outParamCount` method. You can work with the output parameters of a stored procedure by using the object's `outParameters` method. If `outParamCount` returns 0, the stored procedure has no output parameters. In this situation, do not call `outParameters`.

For example, suppose you created a stored procedure that finds the name of an employee when given an ID. If there is an employee name associated with the given ID, the stored procedure returns 1, and its output parameter contains the employee name. Otherwise, the output parameter is empty. The following code either displays the employee name or a message indicating the name wasn't found:

```
id = 100;
getNameProc = connobj.storedProc("getName", id);
returnValue = getNameProc.returnValue();
if (returnValue == 1)
    write ("Name of employee is " + getNameProc.outParameters(0));
else
    write ("No employee with id = " + id);
```

Assume a stored procedure has one input parameter, one input/output parameter, and one output parameter. Further, assume the call to the stored procedure sends a value for the input parameter and the input/output parameter as shown here:

```
spobj = connobj.storedProc("myinout", 34, 56);
```

The `outParameters` method returns any input/output parameters before it returns the first output parameter.

In the preceding example, if you call `outParameters(1)`, it returns the value returned from the stored procedure. By contrast, if you call `outParameters(0)`, the method returns 56. This is the value passed to the stored procedure in the input/output parameter position.

**Note** Output parameters cannot be null; however, you can assign a null value to input or input/output parameters.

For DB2, ODBC, and Sybase, you must retrieve `resultSet` objects and use the `returnValue` method before you call `outParameters`. Once you call `returnValue` or `outParameters`, you can't get any more data from a result set, and you can't get any additional result sets. You should call `outParameters` after you have processed the result set and any return values.

## Informix and Sybase Exceptions

Informix and Sybase stored procedures can return error codes using exceptions. After you run the stored procedure, you can retrieve these error codes and error messages using the `majorErrorCode` and `majorErrorMessage` methods of the associated database or `Connection` object.

For example, assume you have the following Informix stored procedure:

```
create procedure usercheck (user varchar(20))
if user = 'LiveWire' then
raise exception -746, 0, 'User not Allowed';
endif
end procedure
```

When you run this stored procedure, you could check whether an error occurred and then access the error code and message as follows:

```
spobj = connobj.storedProc("usercheck");

if ( connobj.majorErrorCode() ) {
    write("The procedure returned this error code: " +
        connobj.majorErrorCode());
    write("The procedure returned this error message: " +
        connobj.majorErrorMessage());
}
```



# Configuring Your Database

This chapter describes how to set up your database to run with the LiveWire Database Service. You should read this chapter and “Configuration Information” on page 49 before you try to use LiveWire with your JavaScript applications.

**Note** There may have been changes to the database clients that are supported. For the latest information, see the *Enterprise Server 3.x Release Notes*.

This chapter contains the following sections:

- Checking Your Database Configuration
- Supported Database Clients and ODBC Drivers
- DB2
- Informix
- ODBC
- Oracle
- Sybase

Unlike in earlier releases, 3.x versions of Netscape servers require that you install a database client library (and a particular version of that library) if you wish to use the LiveWire Database Service. You must also configure the client library for use with LiveWire.

Netscape servers do not ship with any database client libraries. You must contact your database vendor for the appropriate library. You need only install and configure the database client libraries for the databases you will use.

If you install your database on a machine other than the one on which the web server is installed, you must have a client database library installed on the machine that has the web server. You must obtain the proper license arrangements directly from your database vendor. Netscape does not make these arrangements for you.

The requirements for configuring your database may differ if your database and your web server are installed on the same machine or on different machines. If they are on the same machine, the following information refers to it as a *local* configuration; if on different machines, as a *remote* configuration.

This chapter describes only those aspects of installing the database client that are specific to installing it for use with LiveWire. For general information on installing a database client, refer to the appropriate database vendor documentation.

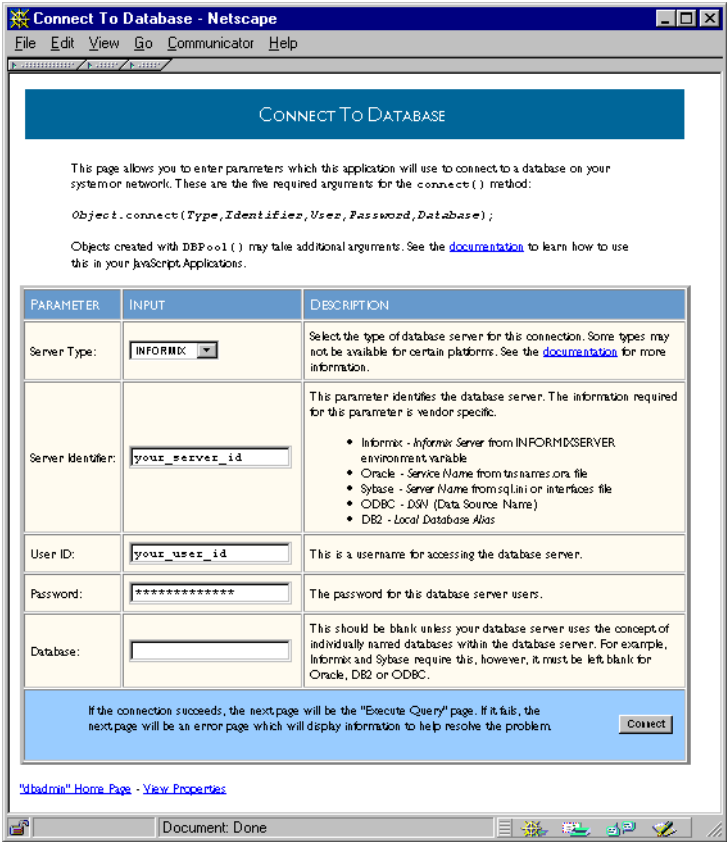
# Checking Your Database Configuration

After you've done the setup described in this chapter, you can use the `dbadmin` sample application to verify that your database connection works properly. You use this JavaScript sample application to connect to your database server and perform various simple tasks such as executing a `SELECT` statement and displaying the results or sending an arbitrary SQL command to the server.

Because you can use `dbadmin` to modify and delete database tables, access to it is automatically restricted if you choose to protect the Application Manager. For more information on restricting the Application Manager, see "Controlling Access to an Application" on page 65.

The first thing you must do when using `dbadmin` is to connect to a database. Choose **Connect to Database**. A form, shown in Figure 17.1, appears in which you can enter connection information. Enter the parameters, and click **Connect** to attempt to connect to the server. For information on the parameters you use to connect, see "Database Connection Pools" on page 314; for further information, see the description of the `connect` method in the *Server-Side JavaScript Reference*.

Figure 17.1 The dbadmin connection page



If this connection succeeds, the Execute Query page appears. In this case, your database is properly configured for working with the LiveWire Database Service. If the connection fails, the Database Connect Error page appears. In this case, make sure you've followed the instructions for your particular database and hardware configuration.

# Supported Database Clients and ODBC Drivers

The following table summarizes the specific database vendors supported on each platform for Netscape Enterprise Server. These database vendors are not supported for Netscape FastTrack Server.

Table 17.1 Database vendor client libraries supported on each platform by Netscape Enterprise Server

Database Vendor	AIX	DEC	Irix 6.2 & 6.4	HP-UX	Solaris 2.5/ 2.5.1	Windows NT 3.51/4.0
DB2	CAE 2.1.2	Not supported	CAE 2.1.2	CAE 2.1.2	CAE 2.1.2 with APAR #JR10150	CAE 2.1.2
Informix	Informix Client 7.22	Informix Client 7.22	Informix Client 7.22	Informix Client 7.22	Informix Client 7.22	Informix Client 7.20
Oracle <sup>a</sup>	Oracle Client 7.3.x	Oracle Client 7.3.x	Oracle Client 7.3.x	Oracle Client 7.3.x	Oracle Client 7.3.x	Oracle Client 7.3.2
Sybase	OpenClient/ C 11.1	OpenClient/ C 11.1	OpenClient/ C 10.0.3sC	OpenClient/ C 11.1	OpenClient/ C 11.1	OpenClient/ C 10.0.3 and 11.1

a. Oracle SQL\*Net version 1.1 is no longer supported.

The following table summarizes support for ODBC on Windows NT for both Netscape Enterprise Server and Netscape FastTrack Server.

Table 17.2 Windows NT ODBC Support

ODBC Component	Windows NT 3.51/4.0
ODBC Manager	MS ODBC Manager 2.5
ODBC Drivers	
MS SQL Server 6.5	MS SQL Server Driver 2.65 (sqlsrv32.dll)
MS SQL Server 6.0	MS SQL Server Driver 2.50.0121 (sqlsrv32.dll)
MS Access 7.0	MS Access Driver 3.5 (odbcjt32.dll) with patch WX1350 from Microsoft
Sybase SQL Anywhere 5.0	Sybase SQL Anywhere Driver 5.5.01 (wod50t.dll)
MS FoxPro x.0	MS FoxPro Driver 3.5 (odbcjt32.dll) with patch WX1350 from Microsoft
MS Excel 7.0	MS Excel Driver 3.5 (odbcjt32.dll) with patch WX1350 from Microsoft

The following table summarizes support for ODBC on each Unix platform for both Netscape Enterprise Server and Netscape FastTrack Server. ODBC is not supported on DEC or AIX.

Table 17.3 Unix ODBC Support

ODBC Component	AIX and HP-UX	Irix 6.2 & 6.4	Solaris 2.5/2.5.1
ODBC Manager	Visigenic 2.0	Visigenic 2.0	Visigenic 2.0
ODBC Drivers			
MS SQL Server 6.5	Visigenic MS SQL Server Driver version 2.00.100 (vsmssql.so.1)	Visigenic MS SQL Server Driver version 2.00.1200 (vsmssql.so.1)	Visigenic MS SQL Server Driver version 2.00.0600 (vsmssql.so.1) and version 2.00.1200 or OpenLink Generic ODBC client version 1.5 (using this client requires the request broker from the OpenLink Workgroup Edition ODBC Driver on the NT server)
MS SQL Server 6.0	Visigenic MS SQL Server Driver version 2.00.100 (vsmssql.so.1)	Visigenic MS SQL Server Driver 2.00.0200 (vsmssql.so.1)	Visigenic MS SQL Server Driver 2.00.0600 (vsmssql.so.1) or OpenLink Generic ODBC client version 1.5 (using this client requires the request broker from the OpenLink Workgroup Edition ODBC Driver on the NT server)

The following table lists the capabilities of the supported ODBC drivers on NT.

Table 17.4 ODBC driver capabilities on NT

SQL Database	Connect	SQL passthrough	Read-only cursor	Updatable cursor	Stored procedures
MS-SQL Server 6.0/6.5	Yes	Yes	Yes	Yes	Yes
Sybase SQL Anywhere	Yes	Yes	Yes	Yes	Not tested
Access	Yes	Yes	Yes	No	N/A
Foxpro	Yes	Yes	Yes	No	N/A
Excel	Yes	Yes	Yes	No	N/A

The following table lists the capabilities of the supported ODBC drivers on Unix platforms.

Table 17.5 ODBC driver capabilities on Unix

Unix	Connect	SQL passthrough	Read-only cursor	Updatable cursor	Stored procedures
AIX	Yes	Yes	Yes	Yes	Yes
HP-UX	Yes	Yes	Yes	Yes	Yes
Irix	Yes	Yes	Yes	Yes	Yes
Solaris (Visigenics)	Yes	Yes	Yes	Yes	Yes
Solaris (OpenLink)	Yes	Yes	Yes	No	No

# DB2

To use a DB2 server, you must have Netscape Enterprise Server. You cannot access DB2 from Netscape FastTrack Server.

If the database and the web server are on different machines, follow the instructions in “DB2 Remote” on page 376.

If the database and the web server are on the same machine, follow the instructions in “DB2 Local” on page 377.

## DB2 Remote

**All platforms:** Install the DB2 client, version 2.1.2. For Solaris, you need APAR #JR10150. For information, see the DB2 documentation at <http://www.software.ibm.com/data/db2/>.

To determine if you can connect to the DB2 server, you can issue the following command from the DB2 command line:

```
DB2 TERMINATE # this command allows the catalog command to take effect
DB2 CONNECT TO databasename USERID userid USING password
```

If you use the BLOB or CLOB data types in your application, you must set the `longdatacompat` option in your `$DB2PATH/db2cli.ini` file to 1. For example:

```
[Database name]
longdatacompat=1
```

If you make changes to the `db2cli.ini` file, you must restart your web server for them to take effect.



**Unix only:** You must set the following environment variables:

DB2INSTANCE	Specifies the name of the connection port defined on both the server and client. This name is also in the <code>dbm</code> configuration file for the <code>SVCENAME</code> configuration parameter.
DB2PATH	Specifies the top-level directory in which DB2 is installed. For example: <code>/home/\$DB2INSTANCE/sql1lib</code>
DB2COMM	Verify that this variable specifies the protocol that will be used. For example: <code>DB2COMM=TCPIP</code>
PATH	Must include <code>\$DB2PATH/misc:\$DB2PATH/adm:\$DB2PATH/bin</code>
LD_LIBRARY_PATH	(Solaris and Irix) Must include the DB2 <code>lib</code> directory. For example, on Solaris it must include <code>/opt/IBMDB2/v2.1/lib</code> .
SHLIB_PATH	(HP-UX) Must include the DB2 <code>lib</code> directory.
LIBPATH	(AIX) Must include the DB2 <code>lib</code> directory.

## DB2 Local

**All platforms:** Install the DB2 client, version 2.1.2. For Solaris, you need APAR #JR10150. For more detailed information, see the DB2 documentation at <http://www.software.ibm.com/data/db2>.

If you use the `BLOB` or `CLOB` data types in your application, you must set the `longdatacompat` option in your `$DB2PATH/db2cli.ini` file to 1. For example:

```
[Database name]
longdatacompat=1
```

If you make changes to the `db2cli.ini` file, you must restart your web server for them to take effect.

**Unix only:** You must set the same environment variables as for a remote connection.

# Informix

To use an Informix server, you must have Netscape Enterprise Server. You cannot access Informix from Netscape FastTrack Server.

If the database and the web server are on different machines, follow the instructions in “Informix Remote” on page 378.

If the database and the web server are on the same machine, follow the instructions in “Informix Local” on page 379.

## Informix Remote

**Unix only:** Install an Informix ESQL/C Runtime Client 7.22 (also called Informix I-Connect) and then set the following environment variables:

<code>INFORMIXDIR</code>	Specifies the top-level directory in which Informix is installed.
<code>INFORMIXSERVER</code>	Specifies the name of your default Informix server.
<code>INFORMIXSQLHOSTS</code>	Specifies the path of the <code>sqlhosts</code> file if you move it somewhere other than <code>\$INFORMIXDIR/etc/sqlhosts</code> . You do not need to set this variable if you leave the <code>sqlhosts</code> file in this directory.
<code>SHLIB_PATH</code>	(HP-UX) Must include <code>\$INFORMIXDIR/lib</code> and <code>\$INFORMIXDIR/lib/esql</code> .

You must also modify `$INFORMIXDIR/etc/sqlhosts` to match the service name in the `/etc/services` file. For information on how to do so, see your Informix documentation.

**NT only:** Install an Informix ESQL/C Runtime Client 7.20 (also called Informix I-Connect.) During installation all necessary environment variables are set. You use the appropriate Informix utility to enter the necessary information about the remote server you wish to connect to.

If you run your web Server as a System, be sure that you have run `regcopy.exe`.

**All platforms:** Depending on your name service, you may also need to edit the appropriate file to add the IP address of the remote host machine you are connecting to. On NT, this file is `winnt\system32\drivers\etc\hosts` file under the NT SystemRoot. On Unix, this file is `/etc/hosts`.

In the same directory, add the following line to the `services` file:

```
ifmx_srvc port/tcp # Informix Online Server
```

where *ifmx\_srvc* is the name of your service and *port* is its port number. The port number must match the port on the remote machine that the Informix server listens to. To make this line valid, you must either insert at least one space after `tcp` or place a comment at the end of the line. For example, if your Informix service is named `ifmx1` and the port number is 1321, you add this line:

```
ifmx1 1321/tcp # Informix Online Server
```

## Informix Local

If you install Informix locally, you must install the Informix client before you install the Informix server.

**Unix only:** If you use 7.22 Online Server for Unix, the installation process creates the appropriate directory structure and `sqlhosts` file. You must set the environment variables as for a remote server.

**NT only:** You should install the Online Server 7.20. This installs the client; no additional steps are necessary. If you run your web Server as a System, be sure that you have run `regcopy.exe`.

## ODBC

**All platforms:** For information on the capabilities of the supported ODBC drivers, see “Supported Database Clients and ODBC Drivers” on page 372.

You need to have the appropriate ODBC drivers for the database you are connecting to. You also need to have additional ODBC connectivity files.

Most software products that provide (or advertise) ODBC connectivity supply an ODBC driver or drivers and ODBC connectivity.

**NT only:** Currently Netscape has certified with ODBC Manager version 2.5. If you have access to an ODBC driver, but not to the ODBC connectivity files, you can obtain them from the MS ODBC SDK. To get updated files for Access, Foxpro, and Excel, you may need patch WX1350 from Microsoft.

**Unix only:** For ODBC on Unix, you can use either the driver from Visigenic or from OpenLink. If you're using the Visigenic driver, follow the instructions in "Visigenic ODBC Driver (Unix only)" on page 381. If you're using the OpenLink driver, follow the instructions in "OpenLink ODBC Driver (Solaris only)" on page 380.

## ODBC Data Source Names (NT only)

Two types of data sources can be created:

- **System DSN:** If you're using a system DSN, the web server must be started using the System account.
- **User DSN:** If you're using a user DSN, the web server must be started using an appropriate NT user account.

The data source describes the connection parameter for each database needing ODBC connectivity. The data source is defined using the ODBC administrator. When ODBC is installed, an administrator is also installed. Each ODBC driver requires different pieces of information to set up the data source.

## OpenLink ODBC Driver (Solaris only)

Install the request broker, OpenLink Workgroup Edition ODBC Driver, on the database server. This must be running before you can connect to the database using the OpenLink request agent.

Install the request agent, in OpenLink Generic ODBC client version 1.5, on the database client machine.

Rename or copy the request agent's driver manager file from `libiodbc.so` to `libodbc.so` in the `$ODBCDIR/lib` directory, where `$ODBCDIR` is the directory in which ODBC is installed.

When you installed your server, you installed it to run as some user, either root, nobody, or a particular server user. The user you pick must have a real home directory, which you may have to create. For example, the default home directory for the nobody user on Irix is `/dev/null`. If you install your server on Irix as nobody, you must give the nobody user a different home directory.

In that home directory, you must have an `.odbc.ini` file. For example, if you run the server as root, this file is under the root (`/`) directory.

Set the following environment variables:

<code>LD_LIBRARY_PATH</code>	(Solaris and Irix) Add the location of the ODBC libraries to this variable.
<code>UDBCINI</code>	Specifies the location of the <code>.odbc.ini</code> file.

## Visigenic ODBC Driver (Unix only)

When you installed your server, you installed it to run as some user, either root, nobody, or a particular server user. The user you pick must have a real home directory, which you may have to create. For example, the default home directory for the nobody user on Irix is `/dev/null`. If you install your server on Irix as nobody, you must give the nobody user a different home directory.

In that home directory, you must have an `.odbc.ini` file. For example, if you run the server as root, this file is under the root (`/`) directory.

Set the following environment variable:

<code>LD_LIBRARY_PATH</code>	(Solaris and Irix) Add the location of the ODBC libraries to this variable. In the preceding example, this would be <code>/u/my-user/odbcsdk/lib</code> .
<code>SHLIB_PATH</code>	(HP-UX) Add the location of the ODBC libraries to this variable.
<code>LIBPATH</code>	(AIX) Add the location of the ODBC libraries to this variable.

## Oracle

To use an Oracle server, you must have Netscape Enterprise Server. You cannot access Oracle from Netscape FastTrack Server.

If the database and the web server are on different machines, follow the instructions in “Oracle Remote” on page 382.

If the database and the web server are on the same machine, follow the instructions in “Oracle Local” on page 383.

**Unix only:** Make sure you can connect to your Oracle database via SQL\*Net. When you have finished installation, you can use a loopback test to verify that you connected correctly. For example, from within `sqlplus`, you can try to connect to your database with the following command:

```
connect username/password@service_name
```

Or, from the Unix command line, you could use this command:

```
sqlplus username/password@service_name
```

In these commands, you use the *service\_name* from your `tnsnames.ora` file.

## Oracle Remote

**NT only:** You must install the Oracle 7.3.2 client software for NT. Oracle 7.1 and 7.2 clients are not supported. You must also create the Oracle configuration files using the appropriate Oracle configuration utility.

**Unix only:** Before you can connect to Oracle under Irix, you must have the appropriate Irix patches. See *Enterprise Server 3.x Release Notes* for information on the patches you need.

You must install the Oracle 7.3.x client software for Unix. Oracle 7.1 and 7.2 clients are not supported.

You must set the following environment variables:

ORACLE_HOME	Specifies the top-level directory in which Oracle is installed.
TNS_ADMIN	Specifies the location of configuration files, for example, <code>\$ORACLE_HOME/network/admin</code> . After installation Oracle creates the configuration files under <code>/var/opt/oracle</code> . If <code>listener.ora</code> and <code>tnsnames.ora</code> are in this directory, you might not need to set <code>TNS_ADMIN</code> , because by default Oracle uses <code>/var/opt/oracle</code> .

If you do not set these environment variables properly, Oracle returns the ORA-1019 error code the first time you attempt to connect. For information on error handling, see Chapter 19, “Error Handling for LiveWire.”

## Oracle Local

**Unix only:** Before you can connect to Oracle under Irix, you must have the appropriate Irix patches. See *Enterprise Server 3.x Release Notes* for information on the patches you need.

**All platforms:** You must install an Oracle Workgroup, Enterprise Server 7.3.2 (NT), or Enterprise Server 7.3.x (Unix). Oracle 7.1 and 7.2 clients are not supported. Check with your server vendor to verify that the Oracle server version is compatible with the Oracle client.

You must set the following environment variables:

ORACLE_HOME	Specifies the top-level directory in which Oracle is installed.
ORACLE_SID	Specifies the Oracle System Identifier.

When your Oracle database server is local, you must pass the empty string as the second argument to the `connect` method of the `database` or `DbPool` object or to the `DbPool` constructor. This way, those methods use the value of the `ORACLE_SID` environment variable. For example:

```
database.connect ( "ORACLE", " " "user", "password", " " );
```

For more information on Oracle installation, see Oracle's documentation.

## Sybase

To use a Sybase server, you must have Netscape Enterprise Server. You cannot access Sybase from Netscape FastTrack Server.

If the database and the web server are on different machines, follow the instructions in "Sybase Remote" on page 384.

If the database and the web server are on the same machine, follow the instructions in "Sybase Local" on page 384.

In addition, if you're using a Unix platform and Sybase has a multithreaded driver for that platform, follow the instructions in "Sybase (Unix only)" on page 385. See *Enterprise Server 3.x Release Notes* for a list of the Unix platforms on which Sybase has a multithreaded driver.

## Sybase Remote

**Unix only:** Set the following environment variable:

<code>SYBASE</code>	The top-level directory in which Sybase is installed
<code>LD_LIBRARY_PATH</code>	(DEC) Must include <code>\$SYBASE/lib</code> .

For Solaris, you must also follow the instructions in “Sybase (Unix only)” on page 385.

**All platforms:** You must install SYBASE Open Client/C. Supported versions are listed in “Supported Database Clients and ODBC Drivers” on page 372.

You can use the appropriate Sybase utility to enter, in the `sql.ini` file (NT) and the `interfaces` file (all platforms), the information about the remote server you want to connect to. For more information, see your Sybase documentation.

## Sybase Local

**Unix only:** Set the following environment variable:

<code>SYBASE</code>	The top-level directory in which Sybase is installed
<code>LD_LIBRARY_PATH</code>	(DEC) Must include <code>\$SYBASE/lib</code> .

For Solaris, you must also follow the instructions in “Sybase (Unix only)” on page 385.

**All platforms:** Install a Sybase SQL Server, version 11.1; the client portion is installed with the server. Supported versions are listed in “Supported Database Clients and ODBC Drivers” on page 372.

You can use the appropriate Sybase utility to enter the information about the remote server you want to connect to in the `sql.ini` file (NT) and the `interfaces` file (all platforms). For more information, see your Sybase documentation.



## Sybase (Unix only)

On some Unix platforms, Sybase has both a single-threaded driver and a multithreaded driver. If Sybase has a multithreaded driver for a particular Unix machine, you must use the multithreaded driver with LiveWire. On these platforms, your web server will behave unpredictably with the single-threaded driver. This requirement applies for both a local and a remote connection. It does not apply to Windows platforms.

See *Enterprise Server 3.x Release Notes* for a list of the Unix platforms on which Sybase has a multithreaded driver.

To ensure that you use the multithreaded driver, you must edit your `$SYBASE/config/libtcl.cfg` file. This file contains a pair of lines that enable either the single-threaded or the multithreaded driver. You must have one of these lines commented out and the other active. For example, on Solaris locate these lines:

```
[DRIVERS]
;libtli.so=tcp unused ; This is the nonthreaded tli driver.
libtli_r.so=tcp unused ; This is the threaded tli driver.
```

Make sure that the line for the single-threaded driver is commented out and that the line for the multithreaded driver is not commented out. The filename differs on each platform, but the lines are always in the `DRIVERS` section and are always commented to indicate which is the single-threaded and which the multithreaded driver.

**Note** If you wish to use the Sybase `isql` utility, you must use the nonthreaded `tli` driver. In this case, the line for `libtli_r.so` must be commented out. For information on using this driver, see your Sybase documentation.



# Data Type Conversion

This chapter describes how the JavaScript runtime engine on the server converts between the more complex data types used in relational databases and the simpler ones defined for JavaScript.

This chapter contains the following sections:

- Working with Dates and Databases
- Data-Type Conversion by Database

Databases have a rich set of data types. The JavaScript runtime engine on the server converts these data types to JavaScript values, primarily either strings or numbers. A JavaScript number is stored as a double-precision floating-point value. In general, the runtime engine converts character data types to strings, numeric data types to numbers, and dates to JavaScript `Date` objects. It converts null values to JavaScript null.

**Note** Because JavaScript does not support fixed or packed decimal notation, some precision may be lost when reading and writing packed decimal data types. Be sure to check results before inserting values back into a database, and use appropriate mathematical functions to correct for any loss of precision.

# Working with Dates and Databases

Date values retrieved from databases are converted to JavaScript `Date` objects. To insert a date value in a database, use a JavaScript `Date` object, as follows:

```
cursorName.dateColumn = dateObj
```

Here, `cursorName` is a cursor, `dateColumn` is a column corresponding to a date, and `dateObj` is a JavaScript `Date` object. You create a `Date` object using the `new` operator and the `Date` constructor, as follows:

```
dateObj = new Date(dateString)
```

where `dateString` is a string representing a date. If `dateString` is the empty string, it creates a `Date` object for the current date. For example:

```
custs.orderDate = new Date("Jan 27, 1997")
```

- Note** DB2 databases have `time` and `timestamp` data types. These data types both convert to the `Date` type in JavaScript.
- Warning** The LiveWire Database Service cannot handle dates after February 5, 2037.
- For more information on working with dates in JavaScript, see the *Client-Side JavaScript Guide*.

## Data-Type Conversion by Database

The following table shows the conversions made by the JavaScript runtime engine for DB2 databases.

Table 18.1 DB2 data-type conversions

DB2 Data Type	JavaScript Data Type
<code>char(n)</code> , <code>varchar(n)</code> , <code>long varchar</code> , <code>clob(n)</code>	<code>string</code>
<code>integer</code> , <code>smallint</code>	<code>integer</code>
<code>decimal</code> , <code>double</code>	<code>double</code>
<code>date</code> , <code>time</code> , <code>timestamp</code>	<code>Date</code>
<code>blob</code>	<code>Blob</code>

The following table shows the conversions made by the JavaScript runtime engine for Informix databases.

Table 18.2 Informix data-type conversions

Informix Data Type	JavaScript Data Type
char, nchar, text, varchar, nvarchar	string
decimal(p,s), double precision, float, integer, money(p,s), serial, smallfloat, smallint	number
date, datetime <sup>a</sup>	Date
byte	Blob
interval	Not supported

a. The Informix datetime data type has variable precision defined by the user. Server-side JavaScript displays datetime data with the format of YEAR to SECOND. If a datetime variable has been defined with another precision, such as MONTH to DAY, it may display incorrectly. In this situation, the data is not corrupted by the incorrect display.

ODBC translates a vendor's data types to ODBC data types. For example, the Microsoft SQL Server varchar data type is converted to the ODBC SQL\_VARCHAR data type. For more information, see the ODBC SDK documentation. The following table shows the conversions made by the JavaScript runtime engine for ODBC databases.

Table 18.3 ODBC data-type conversions

ODBC Data Type	JavaScript Data Type
SQL_LONGVARCHAR, SQL_VARCHAR, SQL_CHAR	string
SQL_SMALLINT, SQL_INTEGER, SQL_DOUBLE, SQL_FLOAT, SQL_REAL, SQL_BIGINT, SQL_NUMERIC, SQL_DECIMAL	number
SQL_DATE, SQL_TIME, SQL_TIMESTAMP	Date
SQL_BINARY, SQL_VARBINARY, SQL_LONGBINARY	Blob

The following table shows the conversions made by the JavaScript runtime engine for Oracle databases.

Table 18.4 Oracle data-type conversions

Oracle Data Type	JavaScript Data Type
long, char(n), varchar2(n), rowid	string
number(p,s), number(p,0), float(p)	number
date	Date
raw(n), long raw	Blob

The following table shows the conversions made by the JavaScript runtime engine for Sybase databases.

Table 18.5 Sybase data-type conversions

Sybase Data Type	JavaScript Data Type
char(n), varchar(n), nchar(n), nvarchar(n), text	string
bit, tinyint, smallint, int, float(p), double precision, real, decimal(p,s), numeric(p,s), money, smallmoney	number <sup>a</sup>
datetime, smalldatetime	Date
binary(n), varbinary(n), image	Blob

a. The Sybase client restricts numeric data types to 33 digits. If you insert a JavaScript number with more digits into a Sybase database, you'll get an error.

# Error Handling for LiveWire

This chapter describes the types of errors you can encounter when working with relational databases.

This chapter contains the following sections:

- Return Values
- Error Methods
- Status Codes

When writing a JavaScript application, you should be aware of the various error conditions that can occur. In particular, when you use the LiveWire Database Service to interact with a relational database, errors can occur for a variety of reasons. For example, SQL statements can fail because of referential integrity constraints, lack of user privileges, record or table locking in a multiuser database, and so on. When an action fails, the database server returns an error message indicating the reason for failure.

Your code should check for error conditions and handle them appropriately.

# Return Values

The return value of the methods of the LiveWire objects may indicate whether or not an error occurred. Methods can return values of various types. Depending on the type, you can infer different information about possible errors.

## Number

When a method returns a number, the return value can either represent an actual numeric value or a status code. For example, `Cursor.columns` returns the number of columns in a cursor, but `Cursor.updateRow` returns a number indicating whether or not an error occurred.

The `Cursor.columns` and `ResultSet.columns` methods return an actual numeric value. The following methods return a numeric value that indicates a status code:

```

Connection.beginTransaction
Connection.commitTransaction
Connection.execute
Connection.majorErrorCode
Connection.minorErrorCode
Connection.release
Connection.rollbackTransaction
Connection.SQLTable

Cursor.close
Cursor.deleteRow
Cursor.insertRow
Cursor.updateRow

database.beginTransaction
database.connect
database.commitTransaction
database.disconnect
database.execute
database.majorErrorCode
database.minorErrorCode
database.rollbackTransaction
database.SQLTable
database.storedProcArgs

```



```

DbPool.connect
DbPool.disconnect
DbPool.majorErrorCode
DbPool.minorErrorCode
DbPool.storedProcArgs

Resultset.close

Stproc.close

```

If the numeric return value of a method indicates a status code, 0 indicates successful completion and a nonzero number indicates an error. If the status code is nonzero, you can use the `majorErrorCode` and `majorErrorMessage` methods of the associated `Connection`, `database`, or `DbPool` object to find out information about the error. In some cases, the `minorErrorCode` and `minorErrorMessage` methods provide additional information about the error. For information on the return values of these error methods, see “Error Methods” on page 396.

## Object

When a method returns an object, it can either be a real object or it can be null. If the method returns null, a JavaScript error probably occurred. In most cases, if an error occurred in the database, the method returns a valid object, but the software sets an error code.

The `blob` global function returns an object. In addition, the following methods return an object:

```

Connection.cursor
Connection.storedProc
database.cursor
database.storedProc
DbPool (constructor)
DbPool.connection
Stproc.resultSet

```

Whenever you create a cursor, result set, or stored procedure, you should check for both the existence of the created object and for a possible return code. You can use the `majorErrorCode` and `majorErrorMessage` methods to examine an error.

For example, you might create a cursor and verify its correctness with code similar to the following:

```
// Create the Cursor object.
custs = connobj.cursor ("select id, name, city
    from customer order by id");

// Before continuing, make sure a real cursor was returned
// and there was no database error.
if ( custs && (connobj.majorErrorCode() == 0) ) {

    // Get the first row
    custs.next();

    // ... process the cursor rows ...

    //Close the cursor
    custs.close();
}

else
    // ... handle the error condition ...
```

## Boolean

The following methods return Boolean values:

```
Connection.connected
Cursor.next
database.connected
DbPool.connected
ResultSet.next
```

When a method returns a Boolean value, `true` usually indicates successful completion, whereas `false` indicates some other condition. A return value of `false` does not indicate an actual error; it may indicate a successful termination condition.

For example, `Connection.connected` returns `false` to indicate the `Connection` object is not currently connected. This can mean that an error occurred when the `Connection` object was created, or it can indicate that a previously used connection was intentionally disconnected. Neither of these is an error of the `connected` method. If an error occurred when the connection was created, your code should catch the error with that method. If the connection was terminated, you can reconnect.

As a second example, `Cursor.next` returns `false` when you get to the end of the rows in the cursor. If the `SELECT` statement used to create the `Cursor` object finds the table but no rows match the conditions of the `SELECT` statement, then an empty cursor is created. The first time you call the `next` method for that cursor, it returns `false`. Your code should anticipate this possibility.

## String

When a method returns a string, you usually do not get any error information. If, however, the method returns null, check the associated error method.

The following methods return a string:

```
Connection.majorErrorMessage
Connection.minorErrorMessage
Cursor.columnName
database.majorErrorMessage
database.minorErrorMessage
DbPool.majorErrorMessage
DbPool.minorErrorMessage
ResultSet.columnName
```

## Void

Some methods do not return a value. You cannot tell anything about possible errors from such methods. The following methods do not return a value:

```
Connection.release
Cursor.close
database.disconnect
DbPool.disconnect
ResultSet.close
```

# Error Methods

As discussed earlier, many methods return a numeric status code. When a method returns a status code, there may be a corresponding error code and message from the database server. LiveWire provides four methods for the `Connection`, `DbPool`, and database objects to access database error codes and messages. The methods are:

- `majorErrorMessage`: major error message returned by the database.
- `minorErrorMessage`: secondary message returned by the database.
- `majorErrorCode`: major error code returned by the database. This typically corresponds to the server's `SQLCODE`.
- `minorErrorCode`: secondary error code returned by the database.

The results returned by these methods depend on the database server being used and the database status code. Most of the time you need to consider only the major error code or error message to understand a particular error. The minor error code and minor error message are used in only a small number of situations.

**Note** Calling another method of `Connection`, `DbPool`, or database can reset the error codes and messages. To avoid losing error information, be sure to check these methods before proceeding.

After receiving an error message, your application may want to display a message to the user. Your message may include the string returned by `majorErrorMessage` or `minorErrorMessage` or the number returned by `majorErrorCode` or `minorErrorCode`. Additionally, you may want to process the string or number before displaying it.

In computing the string returned by `majorErrorMessage` and `minorErrorMessage`, LiveWire returns the database vendor string, with additional text prepended. For details on the returned text, see the descriptions of these methods in the *Server-Side JavaScript Reference*.

# Status Codes

The following table lists the status codes returned by various methods. Netscape recommends that you do not use these values directly. Instead, if a method returns a nonzero value, use the associated `majorErrorCode` and `majorErrorMessage` methods to determine the particular error.

Table 19.1 Status codes for LiveWire methods

Status Code	Explanation	Status Code	Explanation
0	No error	14	Null reference parameter
1	Out of memory	15	database object not found
2	Object never initialized	16	Required information is missing
3	Type conversion error	17	Object cannot support multiple readers
4	Database not registered	18	Object cannot support deletions
5	Error reported by server	19	Object cannot support insertions
6	Message from server	20	Object cannot support updates
7	Error from vendor's library	21	Object cannot support updates
8	Lost connection	22	Object cannot support indices
9	End of fetch	23	Object cannot be dropped
10	Invalid use of object	24	Incorrect connection supplied
11	Column does not exist	25	Object cannot support privileges
12	Invalid positioning within object (bounds error)	26	Object cannot support cursors
13	Unsupported feature	27	Unable to open



# Videoapp and Oldvideo Sample Applications

This chapter describes the `videoapp` sample application, which illustrates the use of the LiveWire Database Service. It describes how to configure your environment to run the `videoapp` and `oldvideo` sample applications.

This chapter contains the following sections:

- Configuring Your Environment
- Running Videoapp
- Looking at the Source Files

Netscape servers come with two sample database applications, `videoapp` and `oldvideo`, that illustrate the LiveWire Database Service. These applications are quite similar; they track video rentals at a fictional video store. The `videoapp` application demonstrates the use of the `DbPool` and `Connection` objects. The `oldvideo` application demonstrates the use of the predefined database object.

There are a small number of restrictions on the use of these applications:

- The `videoapp` application cannot currently be used with the Informix database. The `oldvideo` application, however, can be used with Informix.
- While these sample applications can be used with ODBC and SQL Server, if the driver on your platform does not support updatable cursors, the applications will not work. For information on which drivers support updatable cursors, see “Supported Database Clients and ODBC Drivers” on page 372.

- The `videoapp` application uses cursors that span multiple HTML pages. If your database driver is single-threaded, these cursors may hold locks on the database and prevent other users from accessing it. For information on which database drivers are single-threaded, see the *Enterprise Server 3.x Release Notes*.

## Configuring Your Environment

Before you can run these applications, you must make minor changes to the source files and create a database of videos. This section tells you which files you must change and which procedures you use to make these changes and to create the database for each of the supported database servers. See the section for your database server for specific information.

**Note** Your database server must be up and running before you can create your video database, and you must configure your database server and client as described in Chapter 17, “Configuring Your Database.”

In addition, the database-creation scripts use database utilities provided by your database vendor. You should be familiar with how to use these utilities.

## Connecting to the Database and Recompiling

The `videoapp` application is in the `$NSHOME\js\samples\videoapp` directory, where `$NSHOME` is the directory in which you installed the Netscape server. The `oldvideo` application is in the `$NSHOME\js\samples\oldvideo` directory.

For each application, you must change the connect string in the HTML source file, `start.htm`, to match your database environment. For information on the parameters you use to connect, see “Database Connection Pools” on page 314; for even more information, see the description of the `connect` method in the *Server-Side JavaScript Reference*.

For the `videoapp` application, change this line:

```
project.sharedConnections.pool =  
    new DbPool ("<Server Type>", "<Server Identifier>",  
        "<User>", "<Password>", "<Database>", 2, false)
```



For the `oldvideo` application, change this line:

```
database.connect ( "INFORMIX", "yourserver", "informix",
    "informix", "lw_video" )
```

Save your changes and recompile the application. To recompile one of the applications from the command line, run its build file, located in the application's directory. Be sure your `PATH` environment variable includes the path to the compiler (usually `$NSHOME\bin\https`).

Restart the application in the JavaScript Application Manager.

## Creating the Database

There are two sets of creation scripts for `videoapp` and `oldvideo`, in their respective application directories. The sets of scripts are identical. If you run one set, both applications will be able to use the database.

The first time you run the scripts you might see errors about dropping databases or tables that do not exist. These error messages are normal; you can safely ignore them.

## Informix

Before using the following instructions, you must configure your Informix client as described in “Informix” on page 378. In addition, make sure your `PATH` environment variable includes `$INFORMIXDIR\bin` and that your client is configured to use the Informix utilities.

The SQL files for creating the video database (`lw_video`) on Informix are in these two directories:

```
$NSHOME\js\samples\videoapp\ifx
$NSHOME\js\samples\oldvideo\ifx
```

**Note** Remember that path names in this manual are given in NT format if they are for both NT and Unix. On Unix, you would use `$NSHOME/js/samples/videoapp/` `ifx`.

1. On Unix, log in as “informix” user and run the `ifx_load.csh` shell script for `videoapp` and for `oldvideo`.

On NT, in the Informix Server program group, double-click the Command-Line Utilities icon to open a DOS window, and then run the following commands:

```
cd c:\netscape\server\js\samples\videoapp\ifx
ifx_load.bat
```

You can also run the commands from the `oldvideo\ifx` directory:

2. You can now run the application by making the changes described in “Connecting to the Database and Recompiling” on page 400.

## Oracle

Before using the following instructions, you must configure your Oracle client as described in “Oracle” on page 381. In addition, your client must be configured to run the Oracle utilities. To run SQLPlus, you may need to set the `ORACLE_SID` environment variable.

The SQL files for creating the video database on Oracle are in these two directories:

```
$NSHOME\js\samples\videoapp\ora
$NSHOME\js\samples\oldvideo\ora
```

1. On both Unix and NT, start SQL Plus. From the `SQL>` prompt, enter this command:

```
Start $NSHOME\js\samples\videoapp\ora\ora_video.sql
```

You can also run the script from the `oldvideo` directory. This SQL script does not create a new database. Instead, it creates the Oracle tables in the current instance.

2. On Unix, run the `ora_load` script file to load the video tables with data. On NT, run the `ora_load.bat` batch file to load the video tables with data. You must edit the appropriate file to connect to your server; the instructions for doing so are in the file.

3. You can now run the application by making the changes described in “Connecting to the Database and Recompiling” on page 400.

## Sybase

Before using the following instructions, you must configure your Sybase client as described in “Sybase” on page 383. In addition, on Unix be sure your `PATH` environment variable includes `$SYBASE/bin` and set `DSQUERY` to point to your server.

The SQL files for creating the video database on Sybase are in these two directories:

```
$NSHOME\js\samples\videoapp\syb
$NSHOME\js\samples\oldvideo\syb
```

1. Run the appropriate script from the command line. On Unix, the script is:

```
syb_video.csh userid password
```

For example:

```
$NSHOME\js\samples\videoapp\syb\syb_load.csh sa
```

On NT, the script is:

```
syb_load userid password
```

For example:

```
c:\netscape\server\js\samples\videoapp\syb\syb_load sa
```

Alternatively, you can run the script from the `oldvideo` directory.

2. You can now run the application by making the changes described in “Connecting to the Database and Recompiling” on page 400.

**Note** If you have both Sybase and MS SQL Server or DB2 installed on your machine, there is a potential naming confusion. These vendors have utilities with the same name (`bcp` and `isql`). When running this script, be certain that your environment variables are set so that you run the correct utility.

## Microsoft SQL Server (NT only)

Before using the following instructions, you must configure your Sybase client as described in “ODBC” on page 379. In addition on Unix, set `DSQUERY` to point to your server.

The SQL files for creating the video database on MS SQL Server are in these two directories:

```
$NSHOME\js\samples\videoapp\mss
$NSHOME\js\samples\oldvideo\mss
```

1. From a DOS prompt, run this batch file:

```
mss_load userid password
```

For example:

```
c:\netscape\server\js\samples\videoapp\mss\mss_load sa
```

2. You can now run the application by making the changes described in “Connecting to the Database and Recompiling” on page 400.

**Note** If you have both MS SQL Server and Sybase or DB2 installed on your machine, there is a potential naming confusion. These vendors have utilities with the same name (`bcp` and `isql`). When running this script, be certain that your environment variables are set so that you run the correct utility.

## DB2

The SQL files for creating the video database on DB2 are in these two directories:

```
$NSHOME\js\samples\videoapp\db2
$NSHOME\js\samples\oldvideo\db2
```

1. (Unix only) Your `PATH` environment variable must include the `$DB2PATH/bin`, `$DB2PATH/misc`, and `$DB2PATH/adm` directories.
2. Before you can run these scripts, you must have installed the DB2 Software Developer's Kit (DB2 SDK).

3. Also, before you can run the script to create the tables, you must edit it to modify some parameters. On Unix, the script is in `db2_load.csh`; on NT, it is in `db2_load.bat`. Edit the appropriate `db2_load` file and modify the following parameters to reflect your environment:
  - `<nodename>`: node name alias
  - `<hostname>`: host name of the node where the target database resides
  - `<service-name>`: service name or instance name from the services file
  - `<database-name>`: database name
  - `<user>`: authorized user
  - `<password>`: user's password
4. Make sure your `/etc/services` file has entries for your instance or service name if you are creating the database in a remote DB2 server.
5. Run the appropriate version of the script from the DB2 command window. The `db2_load` script runs the `db2_video.sql` and `import.sql` scripts. These subsidiary scripts create the video tables and load them with data from the `*.del` files. They do not create a new database. Instead, they create the DB2 tables in the local database alias specified in the `db2_load` script.

**Note** If you have both DB2 and Sybase or MS SQL Server installed on your machine, there is a potential naming confusion. These vendors have utilities with the same name (`bcp` and `isql`). When running this script, be certain that your environment variables are set so that you run the correct utility.

# Running Videoapp

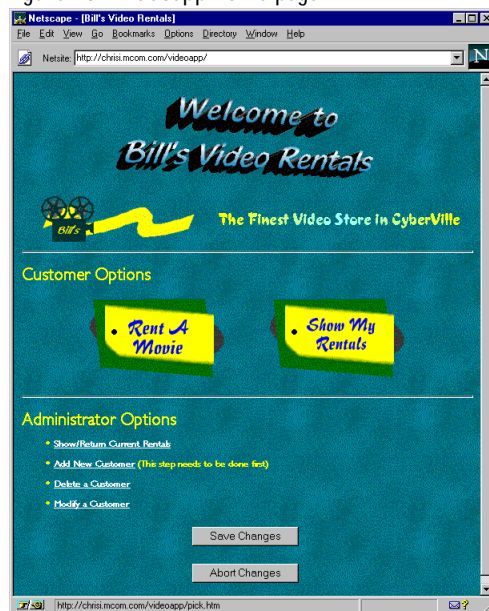
In this section, you get the `videoapp` sample application up and running. This sample is significantly more complex than the samples discussed in Chapter 11, “Quick Start with the Sample Applications.” This chapter only gives an overview of it. You should look at some of the files to start familiarizing yourself with it.

Once you have created the video database and changed the database connection parameters, you can access the application here:

`http://server.domain/videoapp`

After connecting to the database, the Application Manager displays the `videoapp` home page, as shown in Figure 20.1.

Figure 20.1 Videoapp home page



If you cannot connect to the database, you see an error message. Make sure you have entered the correct database connection parameters, as described in “Connecting to the Database and Recompiling” on page 400, recompiled, and restarted the application.

The first thing you must do when you're connected is to add a new customer. Until you have done this, there are no customers to use for any of the other activities.

You can use `videoapp` as a customer or as an administrator. As a customer, you can:

- Rent a movie
- Show all the movies you currently have rented

As an administrator, you can:

- Show all movies and who has them rented
- Return a video for a customer
- Add a new customer entry
- Delete a customer entry
- Modify a customer entry

Run the application and make a few choices to perform different actions.

## Looking at the Source Files

The source HTML files for `videoapp`, listed in the following table, are copiously commented.

Table 20.1 Primary `videoapp` source files

<code>home.htm</code>	The application default page. Has links to <code>pick.htm</code> , <code>status.htm</code> , <code>rentals.htm</code> , <code>customer.htm</code> , and <code>delete.htm</code> . If not connected to the database, this page redirects the client to <code>start.htm</code> .
<code>start.htm</code>	Connects the application to the database, starts a transaction, and then redirects back to <code>home.htm</code> .
<code>abort.htm</code>	Cancels a transaction and begins a new transaction.
<code>save.htm</code>	Commits a transaction and begins a new transaction.

Table 20.1 Primary videoapp source files (Continued)

<code>pick.htm</code>	Allows the customer to rent a movie. It contains frames for <code>category.htm</code> , <code>videos.htm</code> , and <code>pickmenu.htm</code> . The <code>category.htm</code> file displays video categories. The <code>videos.htm</code> file displays all videos in selected category, linked to <code>rent.htm</code> to rent a particular video. The <code>pickmenu.htm</code> file displays choices of other pages to visit.
<code>status.htm</code>	Displays the videos the customer currently has rented. If the customer has not selected an ID, redirects to <code>client.htm</code> , which lets the customer select the ID.
<code>rentals.htm</code>	Displays a list of all rented videos. When the administrator clicks on one, it submits the choice to <code>return.htm</code> , which performs the logic to return the video, then redirects back to <code>rentals.htm</code> .
<code>customer.htm</code>	Allows the administrator to add a new customer. Submits form input to <code>add.htm</code> , which performs logic to add a customer, then redirects back to <code>customer.htm</code> .
<code>delete.htm</code>	Allows the administrator to delete a customer. Displays a list of customers with links to <code>remove.htm</code> , which deletes the specified row from the customer table, then redirects back to <code>delete.htm</code> .
<code>modify.htm</code>	Allows the administrator to modify a customer entry. Displays a list of the first five customers with links to <code>modify1.htm</code> and <code>modify2.htm</code> . Those pages update a specific row in the customer table and then redirect back to <code>modify.htm</code> . The <code>modify3.htm</code> file displays additional customers five at a time.

## Application Architecture

This section orients you to the implementation of some of the functionality in `videoapp`. It describes only how the application works with the database and details the procedure for renting a movie. Other tasks are similar.

### Connection and Workflow

When a user initiates a session with `videoapp` by accessing its default page (`home.htm`), `videoapp` checks whether it is already connected to the database. If so, `videoapp` assumes not only that the application is connected, but also that this user is already connected, and it proceeds from there.



If not connected, `videoapp` redirects to `start.htm`. On this page, it creates a single pool of database connections to be used by all customers, gets a connection for the user, and starts a database transaction for that connection. It then redirects back to `home.htm` to continue. The user never sees the redirection.

The database transaction started on `start.htm` stays open until the user explicitly chooses either to save or discard changes, by clicking the Save Changes or Abort Changes button. When the user clicks one of those buttons, `save.htm` or `abort.htm` is run. These pages commit or roll back the open transaction and then immediately start another transaction. In this way, the customer's connection always stays open.

Once it has a database connection, `videoapp` presents the main page to the user. From that page, the user makes various choices, such as renting a movie or adding a new customer. Each of those options involves displaying various pages that contain server-side JavaScript statements. Many of those pages include statements that use the connection to interact with the database, displaying information or making changes to the database.

The first thing you must do when you're connected is to add a new customer. Until you have done this, there are no customers to use for any of the other activities.

## Renting a Movie

The `pick.htm` page contains a frameset for allowing a customer to rent a movie. The frameset consists of the pages `category.htm`, `videos.htm`, and `pickmenu.htm`.

The `category.htm` page queries the database for a list of the known categories of movie. It then displays those categories as links in a table in the left frame. If the user clicks one of those links, `videoapp` displays `video.htm` in the right frame. There are a few interesting things about the server-side code that accomplishes these tasks. If you look at this page early on, you see these lines:

```
var userId = unscramble(client.userId)
var bucket = project.sharedConnections.connections[userId]
var connection = bucket.connection
```

These statements occur in most of `videoapp`'s pages. They retrieve the connection from where it is stored in the `project` object. The next line then gets a new cursor applicable for this task:

```
cursor = connection.cursor("select * from categories");
```

A variant of this statement occurs at the beginning of most tasks.

Here is the next interesting set of statements:

```
<SERVER>
...
while (cursor.next()) {
    catstr = escape(cursor.category)
</SERVER>

<TR><TD><A HREF="videos.htm?category=" + catstr` TARGET="myright">
<SERVER>write(cursor.category);</SERVER></A>
</TD>
</TR>
<SERVER>

} // bottom of while loop
```

This loop creates a link for every category in the cursor. Notice this statement in particular:

```
<A HREF="videos.htm?category=" + catstr` TARGET="myright">
```

This line creates the link to `videos.htm`. It includes the name of the category in the URL. Assume the category is Comedy. This statement produces the following link:

```
<A HREF="videos.htm?category=Comedy" TARGET="myright">
```

When the user clicks this link, the server goes to `videos.htm` and sets the value of the `request` object's `category` property to Comedy.

The `videos.htm` page can be served either from `pick.htm` or from `category.htm`. In the first case, the `category` property is not set, so the page displays a message requesting the user choose a category. If the `category` property is set, `videos.htm` accesses the database to display information about all the movies in that category. This page uses the same technique as `category.htm` to construct that information and create links to the `rent.htm` page.

The `rent.htm` page actually records the rental for the customer. It gets information from the request and then updates a table in the database to reflect the new rental. This page performs the update, but does not commit the change. That doesn't happen until the user chooses Save Changes or Abort Changes.

The `pickmenu.htm` page simply displays buttons that let you either return to the home page or to the page for adding a new customer.

## Modifying videoapp

As way of getting used to the LiveWire functionality, consider modifying `videoapp`. Here are some features you might add:

- Change the assumption that the existence of the `sharedConnections` array implies that this particular user is connected. You can change `start.htm` to check whether there is an ID for this user in that array and whether the connection stored in that location is currently valid. See “Sharing an Array of Connection Pools” on page 321.
- This application never releases connections back to the pool. Consequently, once a small number of users have connected, nobody else can connect. You can modify this in a couple of ways: add a new command that lets the user indicate completion or implement a scheme to cleanup unused connections. See “Retrieving an Idle Connection” on page 328.



# 5

## *Working with LiveConnect*

- **LiveConnect Overview**
- **Accessing CORBA Services**



# LiveConnect Overview

This chapter describes using LiveConnect technology to let Java and JavaScript code communicate with each other. The chapter assumes you are familiar with Java programming.

This chapter contains the following sections:

- What Is LiveConnect?
- Working with Wrappers
- JavaScript to Java Communication
- Java to JavaScript Communication
- Data Type Conversions

For additional information on using LiveConnect, see the JavaScript technical notes on the DevEdge site.

# What Is LiveConnect?

LiveConnect lets you connect server-side JavaScript applications to Java components or classes on the server. Through Java, you can connect to CORBA-compliant distributed objects using Netscape Internet Service Broker for Java.

Your JavaScript application may want to communicate with code written in other languages, such as Java or C. To communicate with Java code, you use JavaScript's LiveConnect functionality. To communicate with code written in other languages, you have several choices:

- You can wrap your code as a Java object and use LiveConnect directly.
- You can wrap your code as a CORBA-compliant distributed object and use LiveConnect in association with an object request broker.
- You can directly include external libraries in your application.

This chapter discusses using LiveConnect to access non-JavaScript code from JavaScript applications.

Ultimately, LiveConnect allows the JavaScript objects in your application to interact with Java objects. These Java objects are instances of classes on the server's `CLASSPATH`. See "Setting Up for LiveConnect" on page 50 for information on setting `CLASSPATH` appropriately. LiveConnect works for both client-side and server-side JavaScript but has different capabilities appropriate to each environment.

If you have a CORBA service and you have the IDL for it, you can generate Java stubs. The Java stubs can then be accessed from JavaScript using LiveConnect, thus giving you access to your service from JavaScript. For the most part, connecting to CORBA services in this way is just like accessing any other Java code. For this reason, this chapter first talks about using LiveConnect to communicate between Java and JavaScript. Later, it describes what you need to do to access CORBA services.

This chapter assumes you are familiar with Java programming. For information on using Java with Netscape servers, see *Enterprise Server 3.0: Notes for Java Programmers*<sup>1</sup>. For other information on LiveConnect, see the DevEdge Library<sup>2</sup>.

---

1. <http://developer.netscape.com/docs/manuals/enterprise/javanote/index.html>



For all available Java classes, you can access static public properties or methods of the class, or create instances of the class and access public properties and methods of those instances. Unlike on the client, however, you can access *only* those Java objects that were created by your application or created by another JavaScript application and then stored as a property of the `server` object.

If a Java object was created by a server application other than a server-side JavaScript application, you cannot access that Java object. For example, you cannot access a Java object created by a WAI plug-in, NSAPI extension, or an HTTP applet.

When you call a method of a Java object, you can pass JavaScript objects to that method. Java code can set properties and call methods of those JavaScript objects. In this way, you can have both JavaScript code that calls Java code and Java code that calls JavaScript code.

Java code can access a JavaScript application *only* in this fashion. That is, a Java object cannot invoke a JavaScript application unless that JavaScript application (or another JavaScript application) has itself accessed an appropriate Java object and invoked one of its methods.

## 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 it to its original JavaScript object type. The `JSObject` class provides an interface for invoking JavaScript methods and examining JavaScript properties.

---

2. <http://developer.netscape.com/docs/manuals/index.html?content=javascript.html>

# 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 21.1 The LiveConnect Objects

Object	Description
JavaScriptArray	A wrapped Java array, accessed from within JavaScript code.
JavaScriptClass	A JavaScript reference to a Java class.
JavaScriptObject	A wrapped Java object, accessed from within JavaScript code.
JavaScriptPackage	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 429 for complete 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 `JavaScriptObject` because it holds an instance of the Java object `String`. As a `JavaScriptObject`, `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 `JavaScriptObject`, and you can call them as follows:

```
myString.length() // returns 11
```

You access constructors, fields, and methods in a class with the same syntax that you use in Java. For example, the following JavaScript code uses properties of the `request` object to create a new instance of the `Bug` class and then

assigns that new instance to the JavaScript variable `bug`. Because the Java class requires an integer for its first field, this code first converts the `request` string property to an integer before passing it to the constructor.

```
var bug = new Packages.bugbase.Bug(
    parseInt(request.bugId),
    request.bugPriority,
    request);
```

## 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")
```

By default, `$NSHOME\js\samples` directory, where `$NSHOME` is the directory in which the server was installed, is on the server's `CLASSPATH`. You can put your packages in this directory. Alternatively, you can choose to put your Java packages and classes in any other directory. If you do so, make sure the directory is on your `CLASSPATH`.

## 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`:

```
theInt = java.lang.Class.forName("java.lang.Integer")
x = java.lang.reflect.Array.newInstance(theInt, 10)
```

Like the JavaScript `Array` object, `JavaArray` has a `length` property which 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.

`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.

```
theClass = java.lang.Class.forName("java.lang.String")
theArray = java.lang.reflect.Array.newInstance(theClass, 5)
```

## Arguments of Type char

You cannot pass a one-character string to a Java method which requires an argument of type `char`. You must pass such methods an integer which corresponds to the Unicode value of the character. For example, the following code assigns the value “H” to the variable `c`:

```
c = new java.lang.Character(72)
```

## Example of JavaScript Calling Java

The `$NSHOME\js\samples\bugbase` directory includes a simple application illustrating the use of LiveConnect. This section describes the JavaScript code in that sample application. See “Example of Calling Server-Side JavaScript” on page 427 for a description of this application’s Java code.

The `bugbase` application represents a simple bug database. You enter a bug by filling in a client-side form with the bug number, priority, affected product, and a short description. Another form allows you to view an existing bug.

The following JavaScript processes the enter action:

```
// Step 1. Verify that ID was entered.
if (request.bugId != "") {
    // Step 2. Create Bug instance and assign to variable.
    var bug = new Packages.bugbase.Bug(parseInt(request.bugId),
        request.bugPriority, request);

    // Step 3. Get access to shared array and store instance there.
    project.bugsLock.lock();
    project.bugs[parseInt(request.bugId)] = bug;
    project.bugsLock.unlock();

    // Step 4. Display information.
    write("<P><b><I>====>Committed bug: </I></b>");
    write(bug, "<BR>");
}
// Step 5. If no ID was entered, alert user.
else {
    write("<P><b><I>====>Couldn't commit bug: please complete
        all fields.</I></b>");
}
```

The steps in this code are:

1. Verify that the user entered an ID for the bug. Enter the bug only in this case.
2. Create an instance of the Java class `Bug`, and assign that instance to the `bug` variable. The `Bug` class constructor takes three parameters: two of them are properties of the `request` object; the third is the JavaScript `request` object itself. Because they are form elements, these `request` properties are both JavaScript strings. The code changes the ID to an integer before passing it to the Java constructor. Having passed the `request` object to the Java constructor, that constructor can then call its methods. This process is discussed in “Example of Calling Server-Side JavaScript” on page 427.
3. Use `project.bugsLock` to get exclusive access to the shared `project.bugs` array and then store the new `Bug` instance in that array, indexed by the bug number specified in the form. Notice that this code stores a Java object reference as the value of a property of a JavaScript object. For information on locking, see “Sharing Objects Safely with Locking” on page 279.
4. Display information to the client about the bug you have just stored.
5. If no bug ID was entered, display a message indicating that the bug couldn't be entered in the database.

## 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.

These classes are delivered in either a `.jar` or a `.zip` file. See the *Server-Side JavaScript Reference* 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. You can specify an environment variable in Windows NT by double-clicking the System icon in the Control Panel and creating 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 429 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 which handles exceptions of type `netscape.javascript.JSException`. This allows your Java code to handle errors in JavaScript code execution which 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 JSEExceptions here
        this.dogBreed = (String)jsDog.getMember("breed");
        this.dogColor = (String)jsDog.getMember("color");
        this.dogSex = (String)jsDog.getMember("sex");
    }
}
```

Notice that 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`.

**Note** A more realistic example would place the call to `getMember` inside a `try...catch` statement to handle errors of type `JSEException`. See “Handling JavaScript Exceptions in Java” on page 425 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. If the JavaScript code is not evaluated, either due to a JavaScript compilation error or to some other error that occurs at run time, the JavaScript interpreter generates an error message that is converted into an instance of `JSException`.

For example, you can use a `try...catch` statement such as the following to handle `LiveConnect` exceptions:

```
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.

## Accessing Server-Side JavaScript

Now let's look specifically at using Java to access server-side JavaScript. For a Java method to access server-side JavaScript objects, it must have been called from a server-side JavaScript application. In client-side JavaScript, Java can initiate an interaction with JavaScript. On the server, Java cannot initiate this interaction.

**Note** When you recompile a Java class that is used in a JavaScript application, the new definition may not take effect immediately. If any JavaScript application running on the web server has a live reference to an object created from the old class definition, all applications continue to use the old definition. For this reason, when you recompile a Java class, you should restart any JavaScript applications that access that class.

## Threading

Java allows you to create separate threads of execution. You need to be careful using this feature when your Java code interacts with JavaScript code.

Every server-side JavaScript request is processed in a thread known as the **request thread**. This request thread is associated with state information such as the JavaScript context being used to process the request, the HTTP request information, and the HTTP response buffer.

When you call Java code from a JavaScript application, that Java code runs in the same request thread as the original JavaScript application. The Java code in that thread can interact with the JavaScript application and be guaranteed that the environment is as it expects. In particular, it can rely on the associated state information.

However, you can create a new thread from your Java code. If you do, that new thread *cannot* interact with the JavaScript application and *cannot* rely on the state information associated with the original request thread. If it attempts to do so, the behavior is undefined. For example, a Java thread you create cannot initiate any execution of JavaScript code using `JSObject`, nor can it use `writeHttpOutput`, because this method requires access to the HTTP response buffer.

## Example of Calling Server-Side JavaScript

The `$NSHOME\js\samples\bugbase` directory includes a simple application that illustrates the use of LiveConnect. This section describes the sample application's Java code. See “Example of JavaScript Calling Java” on page 421 for a description of the basic workings of this application and of its JavaScript code.

```
// Step 1. Import the needed Java objects.
package Bugbase;
import netscape.javascript.*;
import netscape.server.serverenv.*;

// Step 2. Create the Bug class.
public class Bug {
    int id;
    String priority;
    String product;
    String description;
    String submitter;

    // Step 3. Define the class constructor.
    public Bug(int id, String priority, JSObject req)
        throws java.io.IOException
    {
        // write part of http response
        NetscapeServerEnv.writeHttpOutput("Java constructor: Creating
            a new bug.<br>");
        this.id = id;
        this.priority = priority;
        this.product = (String)req.getMember("bugProduct");
        this.description = (String)req.getMember("bugDesc");
    }

    // Step 4. Return a string representation of the object.
    public String toString()
    {
        StringBuffer result = new StringBuffer();
        result.append("\r\nId = " + this.id
            + "; \r\nPriority = " + this.priority
            + "; \r\nProduct = " + this.product
            + "; \r\nDescription = " + this.description);
        return result.toString();
    }
}
```

Many of the steps in this code are not specific to communicating with JavaScript. It is only in steps 1 and 3 that JavaScript is relevant.

1. Specify the package being used in this file and import the `netscape.javascript` and `netscape.server.serverenv` packages. If you omit this step, you cannot use JavaScript objects.
2. Create the Java `Bug` class, specifying its fields.
3. Define the constructor for this class. This constructor takes three parameters: an integer, a string, and an object of type `JSObject`. This final parameter is the representation of a JavaScript object in Java. Through the methods of this object, the constructor can access properties and call methods of the JavaScript object. In this case, it uses the `getMember` method of `JSObject` to get property values from the JavaScript object. Also, this method uses the `writeHttpOutput` method of the predefined `NetscapeServerEnv` object (from the `netscape.server.serverenv` package) to print information during object construction. This method writes a byte array to the same output stream used by the JavaScript `write` function.
4. Define the `toString` method. This is a standard method for a Java object that returns a string representation of the fields of the object.

# 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. These conversions are described in the following sections:

- JavaScript to Java Conversions
- Java to JavaScript Conversions

## 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
- Boolean Values
- String Values
- Null Values
- `JavaArray` and `JavaObject` Objects
- `JavaClass` Objects
- Other JavaScript Objects

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 430.

## 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:

Java parameter type	Conversion rules
double	The exact value is transferred to Java without rounding and without a loss of magnitude or sign.
<code>java.lang.Double</code> <code>java.lang.Object</code>	A new instance of <code>java.lang.Double</code> is created, and the exact value is transferred to Java without rounding and without a loss of magnitude or sign.
float	<ul style="list-style-type: none"> <li>• Values are rounded to float precision.</li> <li>• Values which are unrepresentably large or small are rounded to +infinity or -infinity.</li> </ul>
byte char int long short	<ul style="list-style-type: none"> <li>• Values are rounded using round-to-negative-infinity mode.</li> <li>• Values which are unrepresentably large or small result in a run-time error.</li> <li>• NaN values are converted to zero.</li> </ul>
<code>java.lang.String</code>	Values are converted to strings. For example, <ul style="list-style-type: none"> <li>• 237 becomes "237"</li> </ul>
boolean	<ul style="list-style-type: none"> <li>• 0 and NaN values are converted to false.</li> <li>• Other values are converted to true.</li> </ul>

When a JavaScript number is passed as a parameter to a Java method which 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:

Java parameter type	Conversion rules
boolean	All values are converted directly to the Java equivalents.
<code>java.lang.Boolean</code> <code>java.lang.Object</code>	A new instance of <code>java.lang.Boolean</code> is created. Each parameter creates a new instance, not one instance with the same primitive value.
<code>java.lang.String</code>	Values are converted to strings. For example: <ul style="list-style-type: none"> <li>• true becomes "true"</li> <li>• false becomes "false"</li> </ul>
byte char double float int long short	<ul style="list-style-type: none"> <li>• true becomes 1</li> <li>• false becomes 0</li> </ul>

When a JavaScript Boolean is passed as a parameter to a Java method which 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:

Java parameter type	Conversion rules
<code>java.lang.String</code> <code>java.lang.Object</code>	A JavaScript string is converted to an instance of <code>java.lang.String</code> with an ASCII value.
<code>byte</code> <code>double</code> <code>float</code> <code>int</code> <code>long</code> <code>short</code>	All values are converted to numbers as described in ECMA-262.
<code>char</code>	All values are converted to numbers.
<code>boolean</code>	<ul style="list-style-type: none"><li>• The empty string becomes false.</li><li>• All other values become true.</li></ul>

## 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:

Java parameter type	Conversion rules
Any class Any interface type	The value becomes null.
<code>byte</code> <code>char</code> <code>double</code> <code>float</code> <code>int</code> <code>long</code> <code>short</code>	The value becomes 0.
<code>boolean</code>	The value becomes false.



## JSONArray and JSONObject Objects

In most situations, when you pass a JavaScript `JSONArray` or `JSONObject` 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:

Java parameter type	Conversion rules
Any interface or class that is assignment-compatible with the unwrapped object.	The object is unwrapped.
<code>java.lang.String</code>	The object is unwrapped, the <code>toString</code> method of the unwrapped Java object is called, and the result is returned as a new instance of <code>java.lang.String</code> .
byte char double float int long short	The object is unwrapped, and either of the following situations occur: <ul style="list-style-type: none"> <li>• If the unwrapped Java object has a <code>doubleValue</code> method, the <code>JSONArray</code> or <code>JSONObject</code> is converted to the value returned by this method.</li> <li>• If the unwrapped Java object does not have a <code>doubleValue</code> method, an error occurs.</li> </ul>
boolean	The object is unwrapped and either of the following situations occur: <ul style="list-style-type: none"> <li>• If the unwrapped object has a <code>booleanValue</code> method, the source object is converted to the return value.</li> <li>• If the object does not have a <code>booleanValue</code> method, the conversion fails.</li> </ul>

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:

Java parameter type	Conversion rules
<code>java.lang.Class</code>	The object is unwrapped.
<code>java.lang.JSObject</code> <code>java.lang.Object</code>	The <code>JavaClass</code> object is wrapped in a new instance of <code>java.lang.JSObject</code> .
<code>java.lang.String</code>	The object is unwrapped, the <code>toString</code> method of the unwrapped Java object is called, and the result is returned as a new instance of <code>java.lang.String</code> .
<code>boolean</code>	<p>The object is unwrapped and either of the following situations occur:</p> <ul style="list-style-type: none"> <li>• If the unwrapped object has a <code>booleanValue</code> method, the source object is converted to the return value.</li> <li>• If the object does not have a <code>booleanValue</code> method, the conversion fails.</li> </ul>

## 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:

Java parameter type	Conversion rules
<code>java.lang.JSObject</code> <code>java.lang.Object</code>	The object is wrapped in a new instance of <code>java.lang.JSObject</code> .
<code>java.lang.String</code>	The object is unwrapped, the <code>toString</code> method of the unwrapped Java object is called, and the result is returned as a new instance of <code>java.lang.String</code> .

Java parameter type	Conversion rules
byte char double float int long short	The object is converted to a value using the logic of the <code>ToPrimitive</code> operator described in ECMA-262. The <i>PreferredType</i> hint used with this operator is <code>Number</code> .
boolean	<p>The object is unwrapped and either of the following situations occur:</p> <ul style="list-style-type: none"> <li>• If the unwrapped object has a <code>booleanValue</code> method, the source object is converted to the return value.</li> <li>• If the object does not have a <code>booleanValue</code> method, the conversion fails.</li> </ul>

## 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 calls the `booleanValue` method, if possible, and fails otherwise.

Note that 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 + "";
```

# Accessing CORBA Services

This chapter describes using LiveConnect to access CORBA-compliant distributed objects. Through LiveConnect, you can access Java; through Java, you can connect to CORBA objects using Netscape Internet Service Broker for Java.

This chapter contains the following sections:

- About CORBA Services
- Flexi Sample Application
- Deployment Alternatives

## About CORBA Services

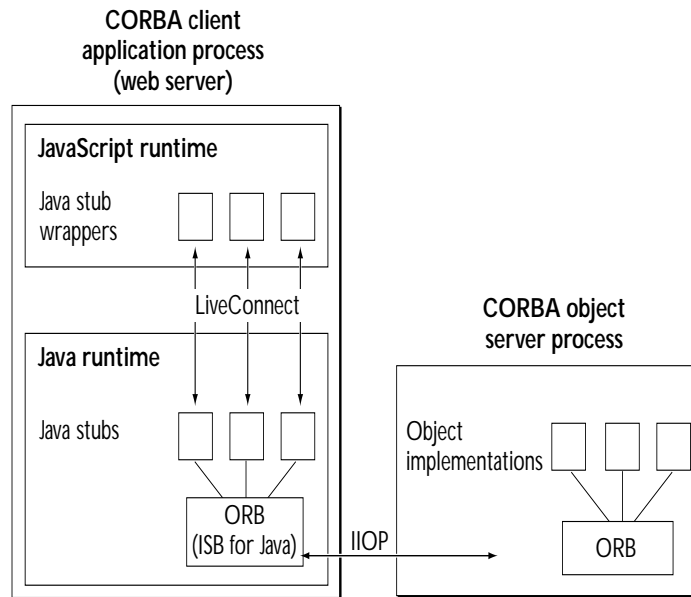
Netscape Internet Service Broker for Java (ISB for Java) is Netscape's object request broker. ISB for Java communicates with itself and with other object request brokers (ORBs) using the Internet InterORB Protocol (IIOP).

ISB for Java enables your JavaScript application to access CORBA-compliant distributed objects deployed in an IIOP-capable ORB (including ISB for Java itself). These objects may be part of a distributed application. To access such a distributed object, you must have a Java stub, and that stub class must be on your `CLASSPATH`. Conversely, you can use Java and LiveConnect to expose parts of your server-side JavaScript application as CORBA-compliant distributed objects.

It is beyond the scope of this manual to tell you how to create CORBA-compliant distributed objects using ISB for Java or how to make Java stubs for such objects. For this information, see the *Netscape Internet Service Broker for Java Programmer's Guide*.

Server-side JavaScript applications can access a distributed object regardless of how it is deployed. The simplest alternative to consider is that the distributed object is created and run as a separate process, as illustrated in the following figure.

Figure 22.1A JavaScript application as a CORBA client



As shown in this illustration, the Java and JavaScript runtime environments are together in the same web server. They communicate using LiveConnect in the standard way described earlier in this chapter. Methods called on the stub wrapper in JavaScript result in method calls on the Java stub object in Java. The stub uses the Java ORB to communicate with the remote service. With this architecture, the object server process can be on any machine that has an ORB and can be written in any language.

The `flexi` sample application illustrates this. In this sample, `FlexiServer` is a stand-alone Java application that has implementations of a number of distributed objects. This example is discussed in “Flexi Sample Application” on page 439.

After you have worked with `flexi`, read “Deployment Alternatives” on page 447 for a discussion of more complicated deployment alternatives.

## Flexi Sample Application

The `flexi` sample application illustrates using server-side JavaScript to access remote services running on an IIOP-enabled ORB and also illustrates a remote service written entirely in Java using ISB for Java. Both the source files and the application executables for the `flexi` sample application are installed in the `$NSHOME\js\samples\flexi` directory.

A flexible spending account (FSA) is an account in which employees may deposit pretax dollars to be used for medical expenses. Employees typically elect to sign up for this plan with the administrator of the plan and select a dollar amount that they want deposited into their account. When an employee incurs a medical expense, the employee submits a claim which, if approved, results in a withdrawal from the account and the remittance of the approved amount to the employee.

The `flexi` sample application provides support for managing flexible spending accounts. With this application, an administrator has these options:

- Create a new account with a given balance.
- Select an existing account by providing the employee’s name.
- Deposit more funds into a selected account.
- Close a selected account.
- Accept or reject a pending claim submitted by the employee.

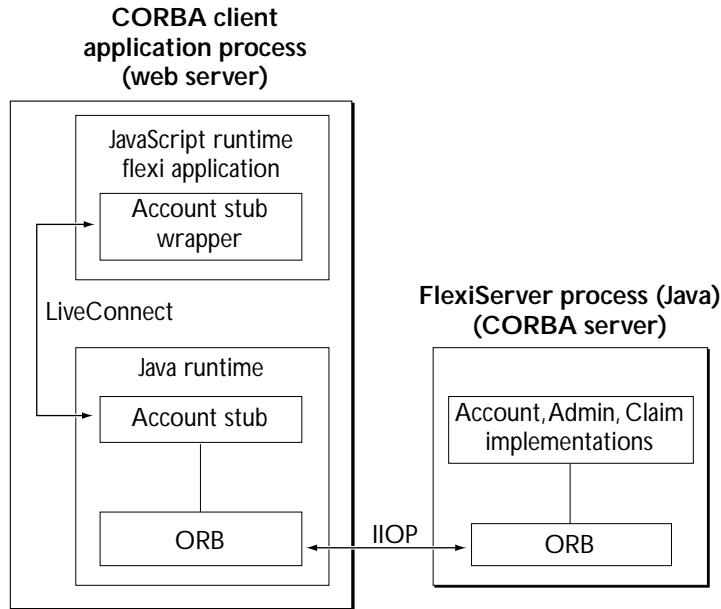
The employee has these options:

- View the status of the account, including the status of any pending claim.
- Submit a new claim by filling out the provided form.

## CORBA Client and Server Processes

Figure 22.2 shows the two primary parts of `flexi`. These implement the CORBA client and service.

Figure 22.2 The `flexi` sample application



The CORBA client is the server-side JavaScript application known as `flexi`. This application implements the administrator and employee user interfaces described earlier. This application connects with the FSA-Admin object (described next) in a separate process or even on a separate machine. The application then uses that object, and other objects returned by FSA-Admin, to perform most of its operations.

The CORBA server is a stand-alone Java application run from the shell. It contains implementations for all the interfaces defined in the IDL file `Flexi.idl`. This stand-alone application, called `FlexiServer`, implements the primary functionality of the FSA system. Upon startup, this application creates an instance of an object implementing the interface `::FSA::Admin` and registers it with the name “FSA-Admin.” Clients of this service (such as the



`flexi` JavaScript application) obtain access to this object first by resolving its name. Clients use this object to create other objects and to get remote references to them.

## Starting FlexiServer

`FlexiServer` is a stand-alone Java application. You can run it on any machine that has JDK 1.0.2. In Enterprise Server 3.01 and FastTrack Server 3.01, you can also run it on a machine that has JDK 1.1.2. Before running `FlexiServer`, you need to ensure that your environment is correct.

From the shell where you're going to start `FlexiServer`, make sure that your `PATH` environment variable includes `$JDK\bin` and that `CLASSPATH` includes the following:

```
...
$NSHOME\js\samples\flexi
$NSHOME\wai\java\nisb.zip
$JDK\lib\classes.zip
```

In these variables, `$JDK` is the directory in which the JDK is installed and `$NSHOME` is the directory in which your web server is installed.

Once the environment is correct, you can start `FlexiServer` as follows:

```
cd $NSHOME\js\samples\flexi\impl
java FlexiServer
```

You should see a message such as the following:

```
Started FSA Admin: Admin[Server,oid=PersistentId[repId=IDL:Flexi/
Admin:1.0,objectName=FSA-Admin]]
```

At this point, `FlexiServer` has started as a CORBA service and registered with the ORB an object with interface `::FSA::Admin` and name `FSA-Admin`. `FlexiServer` runs in the background, waiting for service requests.

## Starting Flexi

You must start `FlexiServer` before you start `flexi`, because `flexi`'s start page attempts to connect to `FlexiServer`.

Add `$NSHOME\js\samples\flexi` to the `CLASSPATH` for your web server. For information on how to do so, see “Setting Up for LiveConnect” on page 15.

Using the Application Manager, install the `flexi` JavaScript application as described in “Installing a New Application” on page 39. The parameters you set for `flexi` are shown in the following table.

Table 22.1 Flexi application settings

Setting	Value
Name	<code>flexi</code>
Web File Path	<code>\$NSHOME\js\samples\flexi\flexi.web</code>
Default Page	<code>fsa.html</code>
Initial Page	<code>start.html</code>
Client Object Maintenance	<code>client-cookie</code>

## Using Flexi

To start `flexi`, you can run it from the Application Manager or enter the following URL:

```
http://server-name/flexi
```

The default page lets the user be identified as an administrator or an employee. To get a quick feel for the application, follow this scenario:

1. Administrator creates an account for a user with a certain balance.
2. Employee selects the account.
3. Employee submits a claim.
4. Administrator selects employee's account.

- 5. Administrator accepts claim, which results in a reduction in the employee's account balance and a remittance of a check for the claim amount.
- 6. Employee selects the account.
- 7. Employee views the status of the account.
- 8. Administrator selects employee's account.
- 9. Administrator deletes claim.

The system can handle only one claim per employee at any time. Once the claim has been deleted, a new claim may be submitted.

## Looking at the Source Files

The following table shows the primary files and directories for `flexi`.

Table 22.2 Flexi files and directories

<code>flexi.idl</code>	File defining the interface to the remote service, including <code>Admin</code> , <code>Account</code> , <code>Claim</code> .
<code>Flexi\</code>	Directory containing code generated from <code>Flexi.idl</code> by the <code>idl2java</code> program. This directory includes the skeletons and stubs for the interfaces.
<code>impl\</code>	Directory containing implementations in Java for all the interfaces defined in <code>Flexi.idl</code> . It also contains the class <code>FlexiServer</code> which implements the main program for the Java application that runs the service.
<code>*.html</code>	Files implementing the server-side JavaScript application. It also includes the application's web file, <code>flexi.web</code> .

Browse through these files to get a clear understanding of this application. Only a few highlights are discussed here.

## Setting Up FlexiServer as a CORBA Server

The main routine of the stand-alone Java application is implemented in `flexi\impl\FlexiServer.java`. Its code is as follows:

```
import org.omg.CORBA.*;

class FlexiServer {
    public static void main(String[] args) {
        try {
            // Initialize the orb and boa.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
            org.omg.CORBA.BOA boa = orb.BOA_init();

            // Create the server object.
            Admin __admin = new Admin();

            // Inform boa that the server object is ready.
            boa.obj_is_ready(__admin);

            // Register the name of the object with the name service.
            // First, determine the name service host;
            // by default use <localhost>:80.
            String _nameServiceHost = null;
            if (args.length > 0) {
                // Assume the first arg is the hostname of the name
                // service host. Expected format: <hostname>:<port>
                _nameServiceHost = args[0];
            }

            else {
                String _localHostName = null;
                try {
                    _localHostName =
                        java.net.InetAddress.getLocalHost().getHostName();
                    _nameServiceHost = _localHostName + ":80";
                }
                catch (java.net.UnknownHostException e) {
                    System.out.println("Couldn't determine local host;
                        can't register name.");
                }
            }

            String _regURL = "http://" + _nameServiceHost + "/FSA-Admin";
            System.out.println("Registering Admin object at URL: " + _regURL);

            // Register the server object.
            netscape.WAI.Naming.register(_regURL, __admin);
            System.out.println("Started FSA Admin: " + __admin);

            boa.impl_is_ready();
        }
    }
}
```

```

        catch (org.omg.CORBA.SystemException e) {
            System.err.println(e);
        }
    }
}

```

This code initializes the ORB and creates an instance of the `Admin` class. It then registers the instance as a distributed object, with a URL of the form `http://host:port/FSA-Admin`. By default, `host` is the name of the host on which `FlexiServer` is run and `port` is 80. You can supply your own value for `host:port` by passing it as an argument to `FlexiServer` when you start it. To use the local host but a different port number, you need to change the sample code and recompile. Once the code has an appropriate name, it registers the object using the `register` method of the `netscape.WAI.Naming` object. For more information, see *Netscape Internet Service Broker for Java Reference Guide*.

Finally, if successful the code prints a message to the console and then waits for requests from CORBA clients. In this case, the only CORBA client that knows about it is the `flexi` JavaScript application.

## Setting up flexi as a CORBA client

The file `start.html` is the initial page of the JavaScript `flexi` application. This page uses `LiveConnect` to initialize ISB for Java and establish the connection to `FSA-Admin`.

```

<server>
// Initialize the orb.
project.orb = Packages.org.omg.CORBA.ORB.init();

// Establish connection to the "FSA-Admin" service.
// By default, assume name service is running on this server.
nameHost = "http://" + server.hostname;
serviceName = "/FSA-Admin";
serviceURL = nameHost + serviceName;

// Resolve name and obtain reference to Admin stub.
project.fsa_admin = Packages.Flexi.AdminHelper.narrow(
    netscape.WAI.Naming.resolve(serviceURL));

</server>

```

The first statement initializes ISB for Java by calling the static `init` method of the Java class `org.omg.CORBA.ORB`. It stores the returned object as a property on the `project` object, so that it lasts for the entire application.

The second set of statements determine the URL that was used to register the FSA-Admin object. If you used a different URL when you registered this object (as described in the last section), you need to make appropriate changes to these statements. The URL used in the CORBA server must be exactly the same as the URL used in the CORBA client.

The code then calls the `resolve` method of the `netscape.WAI.Naming` object to establish the connection to the Admin object that was registered by FlexiServer as FSA-Admin. Finally, it calls the `narrow` method of `AdminHelper` to cast the returned object to the appropriate Java object type. That Java method returns a Java object corresponding to the distributed object. The JavaScript runtime engine wraps the Java object as a JavaScript object and then stores that object as a property on the `project` object. At this point, you can call methods and access properties of that returned object as you would any other Java object. The other pages in `flexi` work through this object.

Once again, for more details on how the CORBA objects work, see *Netscape Internet Service Broker for Java Reference Guide*.

## Using the Admin Object to Administer and View Accounts

Other code in `flexi` creates and then accesses objects in FlexiServer other than the Admin object. These other objects are created by calls to methods of the Admin object. For example, if the employee chooses to submit a claim, a new claim is created in the `account-empl.html` with the following statement:

```
__claim = __account.submitClaim(
    parseFloat(request.claimAmount),
    request.serviceDate,
    request.providerName,
    request.details);
```

This code calls the `submitClaim` method of the `Account` object to create a new employee claim. The implementation of that method, in the file `impl\Account.java`, creates a new `Claim` object, which the code registers with the ORB and then returns, as follows:

```
public Flexi.Claim submitClaim(float amount, String serviceDate,
    String providerName, String details)
{
    Claim __clm = new Claim(this, amount, serviceDate,
        providerName, details);
    org.omg.CORBA.ORB.init().BOA_init().obj_is_ready(__clm);
    _current_clm = __clm;
    System.out.println("***Created a new claim: " + __clm);
    return __clm;
};
```

## Deployment Alternatives

There are two other alternatives for deployment of a CORBA-compliant distributed object that are of interest when working with server-side JavaScript:

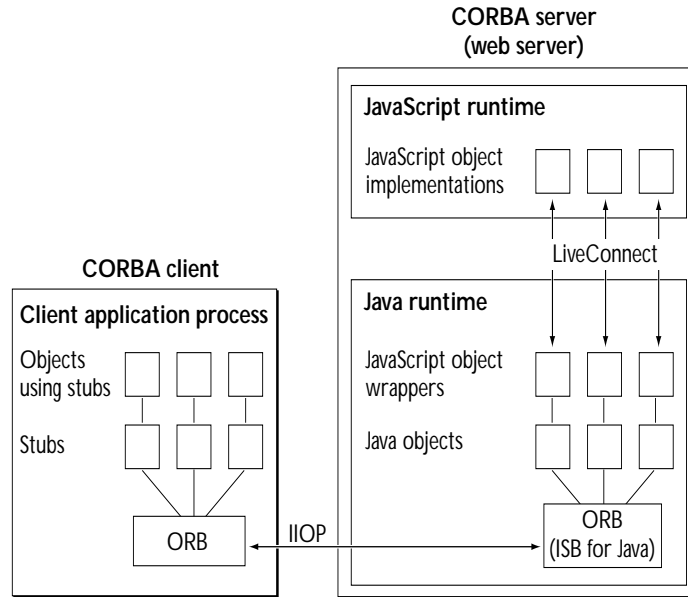
- The object may be created by the web server (but not by a JavaScript application) and run in the web server.
- The object may be created by a JavaScript application and run in the web server.

In these alternatives, the CORBA client and the CORBA server both run in the same web server process.

From the point of view of JavaScript, if the CORBA client is not a JavaScript application, the first alternative is for all practical purposes the same as having the CORBA server run as a separate process.

However, the second alternative, creating a distributed object in a JavaScript application, in effect makes that application the CORBA service. Figure 22.3 illustrates this alternative.

Figure 22.3A JavaScript application as a CORBA server



Once again, the Java and JavaScript runtime environments are together in the same web server. They communicate using LiveConnect in the standard way described earlier in this chapter. In this case, however, the Java and JavaScript processes act together to be the CORBA service. This service then communicates with a CORBA client through ISB for Java in its standard way. The bank sample application is an example of a JavaScript application implementing a CORBA service.

Here, the CORBA client can be on any machine that has an IIOP-capable ORB and can be written in any language. One interesting possibility is that the CORBA client can be a client-side Java application (and through LiveConnect on the client, a client-side JavaScript application). This provides a completely different way for a client-side JavaScript application to communicate with a server-side JavaScript application.



# Glossary

This glossary defines terms useful in understanding JavaScript applications.

<b>active application</b>	A JavaScript application that has been started, and can therefore be run, or accessed, by clients.
<b>application URL</b>	A page in a JavaScript application, relative to the base application URL. Clients use the application URL to access a particular page in the application.
<b>ASCII</b>	American Standard Code for Information Interchange. Defines the codes used to store characters in computers.
<b>base application URL</b>	The name of a JavaScript application, as specified in Application Manager. Clients use the application URL to access the default page of an application.
<b>BLOB</b>	Binary large object. The format of binary data stored in a relational database.
<b>bytecodes</b>	Platform-independent object code, intermediate between source code and platform-specific machine code.
<b>CGI</b>	Common Gateway Interface. A specification for communication between an HTTP server and gateway programs on the server. CGI is a popular interface used to create server-based web applications with languages such as Perl or C.
<b>client</b>	A web browser, such as Netscape Navigator.
<b>client cookie</b>	One of the methods that JavaScript uses to maintain properties of the <code>client</code> object. This method stores name/values pairs as cookies on the client machine.
<b>client-side JavaScript</b>	Core JavaScript plus extensions that control a browser (Navigator or another web browser) and its DOM. For example, client-side extensions allow an application to place elements on an HTML form and respond to user events such as mouse clicks, form input, and page navigation. <i>See also</i> core JavaScript, server-side JavaScript.
<b>client URL encoding</b>	One of the methods that JavaScript uses to maintain properties of the <code>client</code> object. This method appends name/value pairs to a URL string.
<b>commit</b>	To perform all the database actions in a transaction; the attempt to commit may succeed or fail, depending on the actions and the state of the database.

<b>cookie</b>	A mechanism by which the Navigator client can store small items of information on the client machine.
<b>CORBA</b>	Common Object Request Broker Architecture. A standard endorsed by the OMG (Object Management Group), the Object Request Broker (ORB) software that handles the communication between objects in a distributed computing environment.
<b>core JavaScript</b>	The elements common to both client-side and server-side JavaScript. Core JavaScript contains a core set of objects, such as <code>Array</code> , <code>Date</code> , and <code>Math</code> , and a core set of language elements such as operators, control structures, and statements. <i>See also</i> client-side JavaScript, server-side JavaScript.
<b>critical section</b>	A section of code in which you need exclusive access to an object or property to ensure data consistency.
<b>current row</b>	A row in a table referred to by a database cursor.
<b>current transaction</b>	In a database application, the active transaction under which all database actions are performed.
<b>cursor</b>	A data structure returned by a database query, consisting of a virtual table and a pointer to a row in the virtual table; the JavaScript <code>Cursor</code> object has corresponding properties and methods.
<b>DDL</b>	Data Definition Language. Database statements to create, alter, or delete database objects such as tables, keys, stored procedures, and so on.
<b>deadlock</b>	The situation in which two processes each wait for the other to finish a task before continuing. If each waits for the other, neither can continue.
<b>default page</b>	The page, specified in the Application Manager, that a client accesses if the user requests an application URL, but no specific page in the application. Compare to initial page.
<b>deploy</b>	To transfer an application to a location where others can access it. The location can be on the local server's file system or a remote server connected to the Internet.
<b>deployment server</b>	A server on which a JavaScript application is installed that is accessible to end users; also called a production server. Should be different from the development server.

<b>deprecate</b>	To discourage use of a feature without removing the feature from the product. When a JavaScript feature is deprecated, an alternative is typically recommended; you should no longer use the deprecated feature because it might be removed in a future release.
<b>development server</b>	A server, typically inside a firewall, on which you develop and test JavaScript applications, not accessible to end users. Should be different from the deployment server.
<b>DML</b>	Data Manipulation Language. Database statements to select, update, insert, or delete rows in tables.
<b>ECMA</b>	European Computer Manufacturers Association. The international standards association for information and communication systems.
<b>ECMAScript</b>	A standardized, international programming language based on core JavaScript. This standardization version of JavaScript behaves the same way in all applications that support the standard. Companies can use the open standard language to develop their implementation of JavaScript. <i>See also</i> core JavaScript.
<b>external function</b>	A function defined in a native library that can be used in a JavaScript application.
<b>HTML</b>	Hypertext Markup Language. A markup language used to define pages for the World Wide Web.
<b>HTTP</b>	Hypertext Transfer Protocol. The communication protocol used to transfer information between web servers and clients.
<b>initial page</b>	The page, specified in the Application Manager, that the Application Manager runs when the application is first started. Compare to default page.
<b>IP address</b>	A set of four numbers between 0 and 255, separated by periods, that specifies a location for the TCP/IP protocol.
<b>IP address technique</b>	One of JavaScript's techniques for maintaining the <code>client</code> object, in which the server uses the client's IP address to refer to a data structure containing <code>client</code> property values.
<b>LiveConnect</b>	Lets Java and JavaScript code communicate with each other. From JavaScript, you can instantiate Java objects and access their public methods and fields. From Java, you can access JavaScript objects, properties, and methods.
<b>MIME</b>	Multipart Internet Mail Extension. A standard specifying the format of data transferred over the internet.

<b>Netscape cookie protocol</b>	Netscape's format for specifying the parameters of a cookie in the HTTP header.
<b>ODBC</b>	Open Database Connectivity. Microsoft's interface for relational database programming.
<b>primitive value</b>	<p>Data that is directly represented at the lowest level of the language. A JavaScript primitive value is a member of one of the following types: <code>undefined</code>, <code>null</code>, <code>Boolean</code>, <code>number</code>, or <code>string</code>. The following examples show some primitive values.</p> <pre> a=true                // Boolean primitive value b=42                 // number primitive value c="Hello world"      // string primitive value if (x==undefined) {} // undefined primitive value if (x==null) {}      // null primitive value </pre>
<b>roll back</b>	To cancel all the database actions within one transaction.
<b>server cookie</b>	One of JavaScript's techniques for maintaining the <code>client</code> object, in which the server generates a unique name for a client, stored in the cookie file on the client, and later uses the stored name to refer to a data structure containing <code>client</code> property values.
<b>server-side JavaScript</b>	Core JavaScript plus extensions relevant only to running JavaScript on a server. For example, server-side extensions allow an application to communicate with a relational database, provide continuity of information from one invocation to another of the application, or perform file manipulations on a server. <i>See also</i> client-side JavaScript, core JavaScript.
<b>server URL encoding</b>	One of JavaScript's techniques for maintaining the <code>client</code> object, in which the server generates a unique name for a client, appends it to URLs, and later uses the stored name to refer to a data structure containing <code>client</code> property values.
<b>Session Management Service</b>	JavaScript's four predefined objects <code>request</code> , <code>client</code> , <code>project</code> , and <code>server</code> , and one class, <code>Lock</code> , that provide a foundation for sharing data among requests, clients, and applications.
<b>SQL</b>	Structured Query Language. A standard language for defining, controlling, and querying relational databases.

<b>static method or property</b>	A method or property of a built-in object that cannot be a property of instances of the object. For example, you can instantiate new instances of the <code>Date</code> object. Some methods of <code>Date</code> , such as <code>getHours</code> and <code>setDate</code> , are also methods of instances of the <code>Date</code> object. Other methods of <code>Date</code> , such as <code>parse</code> and <code>UTC</code> , are static, so instances of <code>Date</code> do not have these methods.
<b>stopped application</b>	An application that has been stopped with the Application Manager and is not accessible to clients.
<b>transaction</b>	A group of database actions that are performed together; all the actions succeed, or all fail.
<b>updatable cursor</b>	A database cursor in which you can update tables based on the contents of the virtual table.
<b>URL</b>	Universal Resource Locator. The addressing scheme used by the World Wide Web.
<b>web file</b>	The compiled form of a JavaScript application; contains bytecodes. Must be installed in a Netscape web server to run.
<b>WWW</b>	World Wide Web



# Index

## Symbols

- (bitwise NOT) operator 92
- (unary negation) operator 91
- (decrement) operator 91
- ! (logical NOT) operator 95
- != (not equal) operator 90
- \$NSHOME\js\samples  
and CLASSPATH 419
- % (modulus) operator 91
- %= operator 89
- && (logical AND) operator 95
- & (bitwise AND) operator 92
- &, in URLs 236
- &= operator 89
- \*/ comment 130
- \*= operator 89
- + (string concatenation) operator 96
- ++ (increment) operator 91
- += (string concatenation) operator 96
- += operator 89
- .htm file extension 58
- .html file extension 58
- .js file extension 58
- /\* comment 130
- // comment 130
- /= operator 89
- < (less than) operator 90
- << (left shift) operator 92, 94
- <= operator 89
- <= (less than or equal) operator 90
- == (equal) operator 90

- = operator 89
- > (greater than) operator 90
- >= (greater than or equal) operator 90
- >> (sign-propagating right shift) operator 92, 94
- >>= operator 89
- >>> (zero-fill right shift) operator 92, 94
- >>>= operator 89
- ?, in URLs 235
- ?: (conditional) operator 97
- \ 300
- ^ (bitwise XOR) operator 92
- ^= operator 89
- | (bitwise OR) operator 92
- |= operator 89
- || (logical OR) operator 95
- ' (backquote)  
See backquotes
- , (comma) operator 97

## A

- a compiler directive 61, 209
- ACTION attribute 233
  - in sample application 193
- active application
  - glossary entry 449
- addClient function 212, 237, 274, 277
- addResponseHeader function 213, 238, 241, 305
- agent property 250
- anchors, creating 219
- AND (&&) logical operator 95

- AND (&) bitwise operator 92
- application/x-www-form-urlencoded content type 302, 304

- Application Manager
  - capabilities of 56
  - and client object 263
  - configuring default settings 71
  - debugging applications 68
  - details 71
  - figure of 56
  - identifying library files 299
  - installing applications 61–64
  - modifying installation parameters 66
  - overview 56–58
  - protecting 49
  - removing applications 66
  - running applications 67
  - specifying database connections 315
  - starting, stopping, and restarting 66
  - using 68

- applications
  - architecture of 43–47
  - as CORBA clients 438, 447
  - as CORBA servers 447
  - bank sample application 188
  - bugbase sample application 188
  - building 54–55
  - cipher sample application 188
  - compiling 59–61
  - configuration 49–51
  - creating 54
  - creating source files 58–59
  - dbadmin sample application 188
  - debugging 68–70
  - deleting 66
  - deploying 47, 70–71
  - developing 53
  - and file upload 233
  - flexi sample application 188
  - hangman sample application 188
  - and HTML 46
  - installing 61–64
  - jsaccall sample application 189
  - metadata sample application 189
  - migrating 6

- applications (*continued*)
  - modifying installation parameters 66
  - name 64
    - changing 66
    - specifying 63
  - oldvideo sample application 188
  - partitioning tasks 205
  - publishing 62
  - removing 66
  - restarting 66
  - restricting access to 65, 70
  - running 46, 67
    - at runtime 47
  - sample 187–202
  - sendmail sample application 188
  - sharing data 260, 261
  - starting 66
  - starting, stopping, and restarting 66
  - statement types 46
  - stopping 66
  - system requirements for 47
  - upgrading 6
  - URLs for 64
  - videoapp sample application 188
  - viewer sample application 189
  - world sample application 188

- application status, defined 58

- application URLs 64, 206
  - glossary entry 449

- arenas, in garbage collection 242

- arguments array 133

- arithmetic operators 91
  - % (modulus) 91
  - (decrement) 91
  - (unary negation) 91
  - ++ (increment) 91

- Array object 214
  - creating 148
  - overview 147



## arrays

*See also the individual arrays*

- associative 140
- defined 147
- deleting elements 98
- indexing 149
- Java 420
- literals 81
- populating 148
- referring to elements 149
- regular expressions and 151
- two-dimensional 150
- undefined elements 79

## ASCII 296

- glossary entry 449

## assignment operators 89

- `%=` 89
- `&=` 89
- `*=` 89
- `+=` 89
- `/=` 89
- `<<=` 89
- `-=` 89
- `>>=` 89
- `>>>=` 89
- `^=` 89
- `|=` 89
- defined 87

## AUTH\_TYPE CGI variable 229, 250

`auth_type` property 229, 250

`auth_user` property 229, 250

authorization 221

## B

backquotes 216, 218–220

- enclosing JavaScript in 218
- when to use 220

backward compatibility 8

bank application 188

base application URL

- glossary entry 449

Bcc property 288

`beginTransaction` method 324, 349, 392

binary data type 390

binary format 296

binary large objects

- See* BLOBs

bit data type 390

bitwise operators 92

- `&` (AND) 92

- `-` (NOT) 92

- `<<` (left shift) 92, 94

- `>>` (sign-propagating right shift) 92, 94

- `>>>` (zero-fill right shift) 92, 94

- `^` (XOR) 92

- `|` (OR) 92

- logical 93

- shift 93

blob data type 376, 388

blob function 352

`blobImage` method 352

`blobLink` method 352

Blob object 210, 388, 389, 390

## BLOBs

- glossary entry 449

- overview 351

- working with 351–354

Body property 288

Boolean expressions 254

Boolean literals 82

Boolean object 151, 214

- conditional tests and 82

Boolean type conversions (LiveConnect) 431

`booleanValue` method 436

break statement 126

bugbase application 188

Built-in Maximum Database Connections 315

bytecodes 47

- building 54

- glossary entry 449

byte data type 389  
byteToString method 293, 296

## C

C++ libraries 297  
callC function 213, 298, 300  
case sensitivity 79  
    object names 140  
    property names 140  
    regular expressions and 115  
case statement  
    *See* switch statement  
-c compiler directive 60  
Cc property 288  
C functions  
    calling 213  
    registering 212  
CGI, glossary entry 449  
CGI variables  
    accessing 228–232  
    AUTH\_TYPE 229, 250  
    CONTENT\_LENGTH 231  
    CONTENT\_TYPE 231  
    GATEWAY\_INTERFACE 232  
    HTTP\_ACCEPT 231  
    HTTP\_IF\_MODIFIED\_SINCE 231  
    HTTPS 229  
    HTTPS\_KEYSIZE 229  
    HTTPS\_SECRETKEYSIZE 230  
    PATH\_TRANSLATED 230  
    QUERY\_STRING 229, 230, 250  
    REMOTE\_ADDR 230  
    REMOTE\_HOST 230  
    REMOTE\_USER 229, 230, 250  
    REQUEST\_METHOD 229, 230, 250  
    SCRIPT\_NAME 230  
    SERVER\_NAME 230  
    SERVER\_PORT 230  
    SERVER\_PROTOCOL 229, 230, 250  
    SERVER\_SOFTWARE 232  
    SERVER\_URL 230

char arguments 421  
char data type 388, 389, 390  
cipher application 188  
class-based languages, defined 162  
classes  
    defining 162  
    Java 420  
    LiveConnect 422, 423  
CLASSPATH 50, 419, 437  
clearError method 293, 296  
C libraries 297  
    calling 205  
client  
    characteristics of 204  
    communicating with server 232  
    glossary entry 449  
    maintaining client object on 268–271  
    preprocessing data on 204, 234, 238  
client cookies 268–269  
    glossary entry 449  
    lifetime of properties 275  
    maintaining client object with 226, 240, 266,  
        267, 268  
client-mode field, of jsa.conf 73  
client object 210, 215, 221, 225, 237, 252–259,  
    266, 268  
    adding properties to URLs 277–279  
    creating custom 255  
    custom 256–259  
    description of 252  
    destroying 276  
    getting identifier 213  
    id for maintaining 255  
    in sample application 193  
    lifetime 252  
    lifetime of 247, 275–276

- client object (*continued*)
  - maintaining 71, 212, 221, 223, 226, 232, 237, 240, 263–279
    - comparing techniques 264–267
    - on the client 268–271
    - on the server 271–274
    - with client cookies 268–269
    - with client-URL encoding 270–271
    - with IP address 272
    - with server cookie 273
    - with server-URL encoding 274
  - maintenance, specifying 63
  - overview 247
  - in page processing 221, 222
  - properties 253–255
  - properties, expiring 275–276
  - properties, overhead from 254
  - properties, restrictions on 253
  - properties of 253
  - restrictions 256
  - storing properties on project or server 255–256
  - uniquely referring to 255–256
- client properties
  - assigning 269
  - Boolean 254
  - changing 223
- client requests
  - See* requests
- client scripts
  - communicating with server 205
  - generating 239
  - sending values to 237
  - when to use 205
- client-server communication 232–242
  - using cookies for 239–241
- client-side JavaScript 30, 32, 206
  - glossary entry 449
  - illustrated 32
  - object lifetime 205
  - overview 32
- client URL encoding 223, 227, 232, 237, 268, 270–271
  - glossary entry 449
  - lifetime of properties 275
- clob data type 376, 388
- close method 292, 338, 364, 392, 393, 395
- colName property 338, 364
- columnName method 339, 345, 364, 395
- column names, displaying 345
- columns method 338, 345, 364
- comma (,) operator 97
- comments, types of 130
- comment statement 130
- commit, glossary entry 449
- commitTransaction method 324, 349, 392
- communication between client and server 232–242
- comparison operators 90
  - != (not equal) 90
  - < (less than) 90
  - <= (less than or equal) 90
  - == (equal) 90
  - > (greater than) 90
  - >= (greater than or equal) 90
  - on client and server 209
- compiler 54, 209
  - options 60
  - and PATH environment variable 51
  - using 59–61
- conditional (?:) operator 97
- conditional expressions 97
- conditional statements 120–122
  - if...else 120
  - switch 121
- conditional tests, Boolean objects and 82
- configuration styles 65, 71
- confirmation prompts, configuring 72
- connected method 324, 394

- connection method 393
  - of DbPool objects 323
- Connection objects 210, 214, 311, 316, 323, 392, 393, 394, 395, 396
  - creating 323
  - error methods of 396
  - methods 324
  - storedProc method 358
- connection pools
  - See also* DbPool objects
  - as property of project object 321
  - managing 318
  - sharing array of 321
  - sharing fixed set 320
  - storing with project object 318, 320
- connections
  - approaches to 311–324
  - DbPool objects 314
  - disconnecting 316
  - establishing 311–324
  - retrieving 316, 328
  - spanning multiple client requests 325
  - specifying number of 63, 71
  - specifying the number of 315
  - storing 255
  - waiting for 327
- connect method 392, 393
- constructor functions 142
  - global information in 181
  - initializing property values with 173
- containership
  - specifying default object 129
  - with statement and 129
- CONTENT\_LENGTH CGI variable 231
- CONTENT\_TYPE CGI variable 231
- content-length property 304
- content types, managing custom 302–305
- continue statement 127
- cookie.txt 239
- cookie protocol 266, 268, 273
  - See also* client cookies

- cookies 232, 237
  - client, glossary entry 449
  - defined 239
  - glossary entry 450
- CORBA 233, 238, 437–448
  - glossary entry 450
- core JavaScript 32, 208
  - differences on client and server 209
  - glossary entry 450
- critical section 279
  - glossary entry 450
- current row 338
  - glossary entry 450
- current transaction, glossary entry 450
- cursor method 324, 393
- Cursor objects 210, 214, 392, 394, 395
  - methods 338
  - overview 338
  - properties 338
- cursors
  - creating 339
  - customizing display functions 336
  - determining number of columns 345
  - displaying aggregate functions 343
  - displaying column names 345
  - displaying expressions 343
  - displaying record values 341
  - glossary entry 450
  - navigating with 344
  - overview 338
  - updatable 339, 346–348, 353
  - using 338–348

## D

- data
  - converting between formats 296
  - sharing between client and server 232
- database access 205
- database client libraries, configuring 369–385
- database clients, supported 372–375
- database configuration, verifying 370–371

- database connection pools
  - See* connection pools
- database connections
  - See* connections
- database name 314
- database object 210, 215, 392, 393, 394, 395
  - restrictions 318
  - using 309–368
- database pools
  - See* DbPool objects
- database queries
  - and flush function 226
- databases 307–411
  - See also* LiveWire Database Service
  - configuring 369–385
  - connecting to 316
  - converting data types 387–390
  - and dates 388
  - error handling 391–397
  - guidelines for managing connections and threads 317
  - multithreaded 316
  - single-threaded 316
  - typical interactions 310
  - verifying connection 316
- database server name 314
- database servers
  - in JavaScript application architecture 45
- database transactions
  - See* transactions
- database type 314
- data persistence 205
- data sharing 205, 215, 221, 247, 261, 263, 279–286, 292, 315, 318
- data types
  - Boolean conversions 431
  - converting 78
  - converting for LiveWire 387–390
  - converting with LiveConnect 429–436
  - and Date object 78
  - Informix 390
  - JSONArray conversions 433
  - data types (*continued*)
    - JavaClass conversions 434
    - JavaObject conversions 433
    - in JavaScript 36, 77
    - JavaScript to Java conversion 429
    - Java to JavaScript conversion 435
    - null conversions 432
    - number conversions 430
    - ODBC 389
    - Oracle 389
    - other conversions 434
    - string conversions 432
    - Sybase 388, 390
- date data type 388, 389, 390
- Date object 214, 388, 389, 390
  - converting dates to 387, 388
  - creating 151
  - overview 151
- dates
  - converting to Date objects 387
  - and databases 388
  - inserting in database 388
- datetime data type 389, 390
- DB2
  - configuring 376–377
  - data types 388
  - registering stored procedures in 357
  - stored procedure prototypes 358
- DB2COMM environment variable 377
- DB2INSTANCE environment variable 377
- DB2PATH environment variable 377
- dbadmin application 188, 370
- DbBuiltin object 210
- DbPool constructor 393
- DbPool objects 214
  - See also* connection pools, database pools
  - adding properties to 210
  - Boolean value returned by 394
  - connecting to a database with 311
  - connection method 323
  - connections with 314
  - creating 314

- DbPool objects (*continued*)
  - in connection pool arrays 321
  - no value returned by 395
  - numeric value returned by 393
  - object returned by 393
  - stored procedures and 358
  - string value returned by 395
  - using 309–368
- d compiler directive 60
- DDL, glossary entry 450
- deadlock 284–286
  - glossary entry 450
- debug function 70, 212
- Debugger 37
- debugging applications 68
- debugging functions 68
- debug URLs, using 69
- decimal data type 388, 389, 390
- decrement (--) operator 91
- default form values 237
- default objects, specifying 129
- default page
  - glossary entry 450
  - specifying 63, 71
- default settings, Application Manager 71
- delete operator 98, 147
- deleteResponseHeader function 213, 238, 305
- deleteRow method 339, 347, 348, 392
- DELETE SQL statement 348
- deleting
  - array elements 98
  - objects 98, 147
  - properties 98
- deploy, glossary entry 450
- deploying applications 70–71
- deployment server
  - defined 47
  - glossary entry 450
  - updating files to 67
- deprecate, glossary entry 451
- destroy method 269, 270, 276
- development environment, components of 47
- development platform, defined 47
- development server
  - defined 47
  - glossary entry 451
  - updating files from 67
- DHCP 272
- directories
  - conventions used 26
- disconnect method 392, 393, 395
- DML, glossary entry 451
- DNS 230
- do...while statement 124
- document conventions 26
- document root 64
- double data type 388
- double precision data type 389, 390
- Dynamic Host Configuration Protocol 272
- dynamic link libraries 297

## E

- ECMA, glossary entry 451
- ECMA-262 208
- ECMAScript, glossary entry 451
- ECMA specification 38
  - JavaScript documentation and 40
  - JavaScript versions and 39
  - terminology 40
- else statement
  - See* if...else statement
- email
  - See* mail
- environment variables, accessing 229
- eof method 293, 294
- equality on client and server 209
- errorCode method 287

- error handling for LiveWire 391–397
- errorMessage method 287
- error messages, retrieving 316
- error method 293, 296
- error status, for File object 296
- escape function 138, 212, 237
- escaping characters 86
- eval function 135
- event handlers 211
  - See also the individual event handlers*
  - direct substitution 237
  - onClick 234
- exceptions
  - handling in Java 425
- exec method 110
- execute method 324, 337, 392
- exists method 293, 296
- expiration method 269, 275
- expressions
  - See also* regular expressions
  - conditional 97
  - overview 87
  - that return no value 101
  - types of 88
- external functions
  - calling 298
  - defined 297
  - example of use 301
  - glossary entry 451
  - guidelines for writing 299
  - registering 300
  - using in JavaScript 300
  - when to use 298
- external libraries 297–301
  - calling 205
  - identifying files 299
  - security 297
  - specifying 63, 71

## F

- f compiler directive 61
- file access modes 292
- File class 226, 290–297
- file formats 296
- file I/O 297
- File object 210, 214
  - creating 291, 297
  - described 290
  - methods of 293
  - security considerations 290
- files
  - accessing with JavaScript 290–297
  - closing 291
  - getting information for 296
  - locking 292
  - opening 291
  - positioning within 294
  - reading from 294
  - writing to 295
- file upload 302
- fixed decimal notation 387
- flexi application 188, 439, 439–447
- FlexiServer 439, 441
- float data type 389, 390
- floating-point literals 83
- floatValue method 436
- flush function 212, 223, 225, 269, 305
  - described 226
- flush method 226, 295
  - example of use 227
- for...in statement 128, 140
- for loops
  - continuation of 127
  - sequence of execution 122
  - termination of 126
- form elements
  - hidden 234, 237, 238
  - using as request properties 233
- FORM HTML tag 233

- forms
  - and client maintenance 267
  - client scripts for 205
  - default values 237
  - GET method 303
  - hidden elements 237
  - POST method 303
  - processing 192
  - and the request object 246
  - statements 193
  - variables 251

- for statement 122

- From property 288

- function keyword 131

- Function object 155, 214

- functions 131–138, 211

- arguments array 133

- calling 132

- debugging 68

- defining 131

- Function object 155

- predefined 134–138

- recursive 133

- redirect 227

- using built-in 134–138

- write 216, 218

## G

- garbage collection, in JavaScript 242–243

- GATEWAY\_INTERFACE CGI variable 232

- getDay method 153

- getHours method 154

- getLength method 293, 296

- getMember method 424

- getMinutes method 154

- getOptionValueCount function 212

- getOptionValue function 212, 234

- getPosition method 293, 294

- getPostData method 304

- getSeconds method 154

- getTime method 153

- get value of method attribute 233

- global object 40

## H

- hangman application 188

- h compiler directive 60

- headers 238, 241

- request 231

- Hello World application 188

- hidden form elements 237

- history method 249

- home field, of jsa.conf 73

- hostname 230

- hostname property 262

- host property 262

- HREF attribute 219

- HTML 204, 206

- attributes 211, 218

- conditionalizing 205

- embedding JavaScript in 216–220

- generating 212

- glossary entry 451

- and JavaScript 203, 216–220

- sample source code 192

- HTML page

- constructing 206, 221, 222, 225–228

- sending to client 225, 267

- HTML tags

- FORM 233

- IMG 351, 352

- INPUT 238

- HTTP 215, 221

- applets 417

- glossary entry 451

- protocol level 229, 230

- request, *See* requests

- request information 426

- response buffer 426

- user 229, 230



- HTTP\_ACCEPT CGI variable 231
- HTTP\_IF\_MODIFIED\_SINCE CGI variable 231
- HTTPD processes, objects for 261
- HTTPHeader method 231, 233, 241, 303
- HTTP method 229, 230
- HTTPS\_KEYSIZE CGI variable 229
- HTTPS\_SECRETKEYSIZE CGI variable 230
- HTTPS CGI variable 229

## I

- i compiler directive 60
- if...else statement 120
  - in sample application 194
- IIOF 437–448
- image data type 390
- image maps 252
  - using 252
- imageX property 251, 252
- imageY property 251, 252
- IMG HTML tag 351, 352
- increment (++) operator 91
- index.html and default page 63
- Informix 310
  - configuring 378–379
  - data types 389, 390
  - stored procedure parameters 359
- INFORMIXDIR environment variable 378
- INFORMIXSERVER environment variable 378
- INFORMIXSQLHOSTS environment variable 378
- inheritance
  - class-based languages and 163
  - multiple 183
  - property 178
- initializers for objects 141

- initial page 252, 318
  - and request object 249
  - glossary entry 451
  - specifying 63, 71
- INPUT HTML tag 238
- input validation 205
- insertRow method 339, 347, 348, 392
- INSERT SQL statement 348
- installation parameters
  - configuring 71
  - modifying 66
- Installing an application 61
- int data type 390
- integer data type 388, 389
- integers, in JavaScript 83
- Internet InterORB Protocol 437
- interval data type 389
- IP address 267, 271, 272
  - glossary entry 451
  - lifetime of properties 275
- IP address technique, glossary entry 451
- ip property 250
- ISB for Java 416, 437
- isFinite function 135
- ISMAP attribute, of IMG tag 252
- isNaN function 136, 212

## J

- Java
  - See also* LiveConnect
  - accessing JavaScript 422
  - accessing with LiveConnect 418
  - arrays in JavaScript 420
  - calling from JavaScript 418
  - classes 420
  - communication with JavaScript 415–436
  - compared to JavaScript 36, 161–184
  - example of calling from JavaScript 421
  - to JavaScript communication 422
  - JavaScript exceptions and 425

## Java (*continued*)

- methods requiring char arguments 421
  - objects, naming in JavaScript 419
  - object wrappers 417
  - packages 420
- Java applets, server scripts for 205
- JavaArray object 418, 420
- JavaArray type conversions 433
- JavaClass object 418, 420
- JavaClass type conversions (LiveConnect) 434
- JavaObject object 418
- JavaObject type conversions 433
- java package 419
- JavaPackage object 418, 420
- JavaScript
- accessing from Java 422
  - application executable files 47
  - background for using 23
  - basics 203–243
  - client-side 32
  - communication with Java 415–436
  - compared to Java 36, 161–184
  - components illustrated 31
  - core 32
  - debugging 212
  - differences between server and client 30
  - ECMA specification and 38
  - enabling 49
  - example of calling from Java 427
  - files 46, 211
    - compiling 47
  - garbage collection 242–243
  - and HTML 216–220
  - to Java Communication 418
  - Navigator 32–33
  - object wrappers 436
  - overview 30
  - runtime processing 206–208, 220–225
  - server-side 34–36
  - server-side overview 208–215
  - special characters 85
  - tasks on client 204–205
  - tasks on server 204–205

## JavaScript (*continued*)

- variables, and request properties 251
  - versions and Navigator 24
  - where it can occur 211
- JavaScript files 211
- Java virtual machine 46
- jsa.conf file 73
- jsac
  - See* compiler
- jsaccall.c 299
- jsaccall.h 299
- jsaccall application 189, 298
- JSExcption class 422, 425
- js files 46
- JSObject, accessing JavaScript with 424
- JSObject class 422
- jsVersion property 262

## L

- labeled statements
- with break 126
  - with continue 127
- label statement 125
- l compiler directive 60
- LD\_LIBRARY\_PATH environment variable 377, 381, 384
- left shift (<<) operator 92, 94
- length property 159
- LIBPATH environment variable 377, 381
- libraries, external 297–301
- library field, of jsa.conf 73
- links
- for BLOb data 352
  - creating 219
  - with no destination 101

- literals 81
  - Array 81
  - Boolean 82
  - floating point 83
  - integers 83
  - object 83
  - string 84
- LiveConnect 211, 214, 233, 238, 415–436
  - accessing Java directly 418
  - capabilities 417
  - configuration for 50
  - converting data types 429–436
  - glossary entry 451
  - and HTTP applets 417
  - Java to JavaScript communication 422
  - and NSAPI applications 417
  - objects 418
  - restrictions 417
  - and WAI plug-ins 417
- LiveWire database access library 46
- LiveWire Database Service 307–411
  - See also* databases
  - system requirements for 48
- locking 279–286
- lock method 261, 279–286
  - in sample application 194
- Lock object 210, 215, 279–286
- logical operators 95
  - ! (NOT) 95
  - && (AND) 95
  - || (OR) 95
  - short-circuit evaluation 96
- longdatacompat 376
- long data type 390
- long raw data type 390
- loops
  - continuation of 127
  - for...in 128
  - termination of 126

- loop statements 122–128
  - break 126
  - continue 127
  - do...while 124
  - for 122
  - label 125
  - while 124
- lowercase 79

## M

- mail
  - MIME-compliant 288
  - sending with JavaScript 215, 287–290
- majorErrorCode method 316, 324, 392, 393, 396
- majorErrorMessage method 316, 324, 395, 396, 397
- mark and sweep 243
- matching patterns
  - See* regular expressions
- match method 110
- mathematical constants and functions 387
- Math object 156, 214
- maxdbconnect field, of jsa.conf 73
- metadata application 189
- METHOD attribute 233
- method property 229, 250
- methods
  - close 292
  - defined 132
  - defining 145
  - destroy 276
  - expiration 275
  - flush 226
  - history 249
  - open 291
  - setPosition 294
  - static 453
- migrating applications 6
- MIME, glossary entry 451
- MIME-compliant mail 288

- MIME types 290
- minorErrorCode method 316, 324, 392, 393, 396
- minorErrorMessage method 316, 324, 395, 396
- modulus (%) operator 91
- money data type 389, 390
- multimedia
  - using BLOBs 351
- MULTIPLE attribute
  - of SELECT tag 234
- multithreaded databases 316
- multi-threading
  - and Sybase 385

## N

- NAME attribute 219, 233
  - in sample application 193
  - of SELECT tag 234
- native functions 297–301
- Navigator
  - in JavaScript application architecture 44
  - and JavaScript 32, 34
  - JavaScript versions supported 24
- Navigator JavaScript
  - See* client-side JavaScript
- nchar data type 389, 390
- NETSCAPE\_LIVEWIRE 240, 268
- Netscape cookie protocol 268, 273
  - glossary entry 452
- Netscape Internet Service Broker for Java 416, 437
- netscape package 419
- Netscape packages
  - See* packages
- Netscape web servers
  - configuration style support 65
  - in JavaScript application architecture 44
  - sample applications installed with 187
- new operator 99, 142
- next method 339, 341, 344, 362, 364, 394

- NOT (!) logical operator 95
- NOT (-) bitwise operator 92
- NSAPI applications 417
- NSHOME 50
- null keyword 77
- null value conversions (LiveConnect) 432
- number data type 389, 390
- Number function 137
- Number object 158, 214
- number property 194
- numbers
  - converting to characters 293, 296
  - identifying 212
  - Number object 158
  - parsing from strings 136
  - storing 387
  - type conversions (LiveConnect) 430
- numeric data type 390
- nvarchar data type 389, 390

## O

- object field, of jsa.conf 73
- object manipulation statements
  - for...in 128
  - this keyword 99
  - with statement 129
- object model 161–184
- Object object 214
- object prototypes 210
- object request brokers 437
- objects 139–160
  - adding properties 143, 144
  - adding properties to 210
  - constructor function for 142
  - creating 141–143
  - creating new types 99
  - deleting 98, 147
  - establishing default 129
  - getting list of properties for 140
  - indexing properties 144

- objects (*continued*)
  - inheritance 169
  - initializers for 141
  - iterating properties 140
  - JavaScript in Java 423
  - lifetimes of 246
  - literals 83
  - LiveConnect 418
  - model of 161–184
  - overview 140
  - predefined 147
  - single instances of 141
- o compiler directive 60
- ODBC
  - configuring 379–381
  - data types 389
  - drivers supported 372–375
  - glossary entry 452
  - stored procedure prototypes 358
- oldvideo application 188, 399–411
  - and Informix 399
- onClick event handler 234
- Open DataBase Connectivity standard
  - See* ODBC
- OpenLink
  - configuring 380–381
- open method 291
- operators
  - arithmetic 91
  - assignment 89
  - bitwise 92
  - comparison 90
  - defined 87
  - logical 95
  - order of 102
  - overview 88
  - precedence 102
  - special 97
  - string 96
- OPTION tag 235
- OR (|) bitwise operator 92
- OR (| |) logical operator 95

- Oracle 310
  - configuring 381–383
  - data types 389, 390
  - stored procedure parameters 359
- ORACLE\_HOME environment variable 382, 383
- ORACLE\_SID environment variable 383
- ORBs 437
- outParamCount method 360, 367
- outParameters method 356, 360, 365, 367
- output buffer 222
  - flushing 212, 226–227
- output parameters
  - of stored procedures 367

**P**

- packages, Java 420
- Packages object 214, 419
- packed decimal notation 387
- parameters for stored procedures 359
- parentheses in regular expressions 110, 113
- parseFloat function 136, 212
- parseInt function 136, 212
- parse method 154
- Pascal functions 297
- passthrough SQL, executing 337
- PATH\_INFO CGI variable 230
- PATH\_TRANSLATED CGI variable 230
- PATH environment variable 377
  - for the compiler 51
- pattern matching
  - See* regular expressions
- p compiler directive 61
- PI property 156
- pointers 294
- pools of database connections
  - See* connection pools
- popups, client scripts for 205

- port property 262
- post value of method attribute 233
- predefined objects 147
- primitive value, glossary entry 452
- project object 210, 215, 221, 260–261
  - description of 260
  - in sample application 194
  - lifetime 260
  - lifetime of 248, 260
  - locking 261, 279, 283, 292
  - overview 247–248
  - properties 260–261
  - properties of 260
  - sharing 261
  - storing connection pools on 318, 320, 321
- properties
  - See also the individual properties*
  - adding 144, 171
  - class-based languages and 163
  - creating 171
  - getting list of for an object 140
  - indexing 144
  - inheritance 169, 178
  - initializing with constructors 173
  - iterating for an object 140
  - overview 140
  - static 453
- protocol property 229, 250, 262
- prototype-based languages, defined 162
- prototypes 169, 210
  - stored procedures and 358

## Q

- queries
  - customizing output 336
  - displaying 336–337
- QUERY\_STRING CGI variable 229, 230, 250
- query property 229, 250

- quotation marks
  - with backslash 220
  - order of 218
  - for string literals 84

## R

- raw data type 390
- r compiler directive 61
- readByte method 293, 295
- readLn method 293, 295
- read method 293, 294
- real data type 390
- record values, displaying 341
- redirect function 212, 223, 225, 227–228, 249, 267, 269, 270, 274, 277, 305
  - described 227
- RegExp object 103–118
- registerCFunction function 212, 298, 300
- regular expressions 103–118
  - arrays and 151
  - creating 104
  - defined 103
  - examples of 116
  - global search with 115
  - ignoring case 115
  - parentheses in 110, 113
  - remembering substrings 110, 113
  - special characters in 105, 117
  - using 110
  - writing patterns 104
- release method 323, 324, 392, 395
- REMOTE\_ADDR CGI variable 230
- REMOTE\_HOST CGI variable 230
- REMOTE\_USER CGI variable 229, 230, 250
- replace method 110
- REQUEST\_METHOD CGI variable 229, 230, 250
- request bodies, manipulating 304–305
- request headers 231, 233, 241
  - manipulating 303

- request object 210, 215, 221, 225, 232, 249–252, 303, 304
  - creation 249
  - description of 249
  - example of property creation 233
  - and forms 251
  - in sample application 193
  - lifetime of 246, 249
  - overview 246–247
  - in page processing 221, 222
  - properties 229, 250–251
  - properties, and JavaScript variables 251
  - saving properties 228
  - setting properties with form elements 233
- request properties encoding in URLs 236
- requests
  - changing 212, 227–228
  - header 238
  - manipulating raw data 302–305
  - redirecting 236
  - sharing a connection 325
  - terminating 227
- request thread 426
- response headers, manipulating 213, 305
- responses, manipulating raw data 302–305
- resultSet method 355, 360, 361, 393
- ResultSet object 210, 214, 354, 393, 394, 395
  - See also* result sets
  - methods of 364
- result sets 360
  - See also* ResultSet object
  - creating 361
  - ResultSet object 360
- return statement 132
- returnValue method 356, 360, 365, 366
- return values of stored procedures 366
- rollback, glossary entry 452
- rollbackTransaction method 324, 349, 392
- rowid data type 390
- runtime environment components 46
- runtime library 46

- runtime processing 216, 220–225
  - example 207

## S

- sample applications 187–202
  - Hangman 196–202
  - Hello World 190–195
- SCRIPT\_NAME CGI variable 230
- scripts, changing client properties 223
- SCRIPT tag 46, 211
  - See also* client scripts
  - direct substitution in 237
  - when to use
- search method 110
- security
  - external libraries and 297
  - File object and 290
- select lists 234
- SELECT SQL statement 339, 341, 343, 346
- SELECT tag 212, 234
- sendmail application 188
- SendMail class 287–290
- SendMail object 210, 215
- send method 287
- serial data type 389
- server
  - administration functions 221
  - authorization 221
  - characteristics of 204
  - communicating with client 237
  - maintaining client object on 271–274
  - processes, objects for 261
  - restarting 299
  - routing 64
- SERVER\_NAME CGI variable 230
- SERVER\_PORT CGI variable 230
- SERVER\_PROTOCOL CGI variable 229, 230, 250
- SERVER\_SOFTWARE CGI variable 232

- SERVER\_URL CGI variable 230
- server applications
  - developing 53
- server-client communication 232–242
- server cookies 266, 271, 273
  - glossary entry 452
  - lifetime of properties 275
- server object 210, 215, 221, 261–263
  - description of 261
  - lifetime 262
  - lifetime of 248
  - locking 279, 283, 292
  - overview 248
  - properties 262
  - properties of 262
  - sharing data 263
- server scripts
  - communicating with client 205
  - when to use 205
- server-side JavaScript 30, 34–36
  - enabling 49
  - executing 225
  - glossary entry 452
  - illustrated 34, 35
- server-side objects, lifetime of 205
- SERVER tag 46, 205, 206, 211, 216, 217, 220
  - See also* server scripts
  - in sample application 192, 193
  - when to use 220
- server URL encoding 227, 232, 237, 271, 274
  - glossary entry 452
  - lifetime of properties 275
- session key 229
- Session Management Service 205, 221, 245–286
  - glossary entry 452
  - object overview 246–249
- setDay method 153
- setPosition method 293, 294
- setTime method 153, 154
- SHLIB\_PATH environment variable 377, 378, 381
- short-circuit evaluation 96
- sign-propagating right shift (>>) operator 92, 94
- single-threaded databases 316
- smalldatetime data type 390
- smallfloat data type 389
- smallint data type 388, 389, 390
- smallmoney data type 390
- Smtplib property 288
- source files 70
  - components of 58
  - creating 58–59
- source script, example of 192
- special characters in regular expressions 105, 117
- special characters in URLs 237
- special operators 97
- split method 110
- SQL 310
  - See also the individual statements*
  - error handling 391–397
  - executing 337
  - glossary entry 452
- SQL\_BIGINT data type 389
- SQL\_BINARY data type 389
- SQL\_CHAR data type 389
- SQL\_DATE data type 389
- SQL\_DECIMAL data type 389
- SQL\_DOUBLE data type 389
- SQL\_FLOAT data type 389
- SQL\_INTEGER data type 389
- SQL\_LONGBINARY data type 389
- SQL\_LONGVARCHAR data type 389
- SQL\_NUMERIC data type 389
- SQL\_REAL data type 389
- SQL\_SMALLINT data type 389
- SQL\_TIME data type 389
- SQL\_TIMESTAMP data type 389



- SQL\_VARBINARY data type 389
- SQL\_VARCHAR data type 389
- SQLTable method 324, 336, 392
- ssjs\_generateClientID function 213, 255, 327
- ssjs\_getCGIVariable function 213, 228, 229, 250
- ssjs\_getClientID function 213, 255, 273, 274
- start field, of jsa.conf 73
- statements
  - break 126
  - conditional 120–122
  - continue 127
  - do...while 124
  - for 122
  - for...in 128
  - if...else 120
  - label 125
  - loop 122–128
  - object manipulation 128–129
  - overview 119–130
  - switch 121
  - while 124
- static, glossary entry 453
- stopped transaction, glossary entry 453
- storedProcArgs method 358, 392, 393
- stored procedures 354–368
  - See also* Stproc object
  - arguments to 355
  - defining prototypes for 358
  - executing 358
  - in DB2 357
  - output parameters 367
  - parameters for 359
  - registering 357
  - result sets 355
  - return values 356, 366
  - steps for using 356
  - Stproc object 354, 356
- storedProc method 324, 358, 393
- Stproc object 210, 214, 354, 356, 393
  - See also* stored procedures
  - creating 358
  - methods of 360
- string data type 388, 389, 390
- String function 137
- string literals 84
- String object 214
  - overview 159
  - regular expressions and 110
- strings
  - changing order using regular expressions 116
  - concatenating 96
  - operators for 96
  - parsing 212
  - regular expressions and 103
  - searching for patterns 103
  - type conversions (LiveConnect) 432
- stringToByte method 293, 296
- stubs 438
- styles, configuration 65, 71
- subclasses 163
- Subject property 288
- sun package 419
- switch statement 121
- Sybase 310
  - configuring 383–385
  - data types 388, 390
  - stored procedure prototypes 358
- SYBASE environment variable 384

## T

- targets, creating 219
- TCP port 230
- test method 110
- text/html content type 302
- text data type 389, 390
- this keyword 142, 145
  - described 99
  - for object references 146
- threads
  - and databases 316
  - and Java 426

- ticks
  - See* backquotes
- time data type 388
- timestamp data type 388
- tinyint data type 390
- TNS\_ADMIN environment variable 382
- To property 288
- toString method 436
- trace utility 68, 69, 70
  - configuring 72
- transactions
  - committing 315
  - controlling 349
  - glossary entry 453
  - managing 348–350
  - overview 348
  - rolling back 315
  - scope of 350
- typeof operator 100

## U

- unary negation (-) operator 91
- undefined property 77
- undefined value 79
- unescape function 138, 212
- unique identifier 213
- unlock method 261, 279–286
  - in sample application 194
- updatable cursor
  - glossary entry 453
- updateRow method 339, 347, 348, 392
- UPDATE SQL statement 348
- upgrading applications 6
- uppercase 79
- uri field, of jsa.conf 73
- uri property 251

- URL-encoded variables
  - and request object 251
  - resetting 237
- URL encoding
  - See also* client URL encoding, server URL encoding
  - maintaining client object with 268, 270, 271
- URLs 64, 230
  - adding client properties to 277–279
  - adding information to 212
  - application 64
  - changing 227–228
  - and client maintenance 270–271, 274
  - conventions used 26
  - creating 267
  - debug 69
  - dynamically generating 235
  - encoding information in 235–242
  - escaping characters in 212
  - glossary entry 453
  - including special characters 237
  - modifying 232
  - and redirect function 227
  - redirecting to 212
  - and reloading a page 237
  - and Session Management objects 248
  - to start and stop applications 66

## V

- VALUE attribute 238
- varbinary data type 390
- varchar2 data type 390
- varchar data type 388, 389, 390
- variables
  - declaring 79
  - in JavaScript 79
  - naming 79
  - scope of 80
  - undefined 79
- var statement 79
- v compiler directive 60
- VDBCINI environment variable 381

- versions of JavaScript 24
- videoapp application 188, 399–411
  - and Informix 399
  - and ODBC 399
  - and SQL Server 399
- video application
  - See* videoapp application
- viewer application 189
- Visigenic, configuring 381
- Visual JavaScript 38
- void operator 101

## W

- WAI plug-ins 417
- web files 47, 67
  - building 54
  - defined 59
  - glossary entry 453
  - moving 67
  - specifying path 63, 71
- while loops
  - continuation of 127
  - termination of 126

- while statement 124
- with statement 157
  - described 129
- wrappers
  - for Java objects 417
  - for JavaScript objects 436
- writeByte method 293, 295
- write function 192, 212, 217, 225, 277, 291
  - with backquotes 218
  - and client maintenance 270
  - described 226
  - and flush 226
  - with SERVER tag 216
- writeln method 293, 295
- write method 293, 295
- WWW, glossary entry 453

## X

- XOR (^) operator 92

## Z

- zero-fill right shift (>>>) operator 92, 94