Server-Side ActionScript Language Reference
# Contents

**Chapter 1: Server-Side ActionScript Language Reference**

- Using server-side ActionScript ........................................ 5
- Using naming conventions .............................................. 7
- ActionScript classes ...................................................... 8
- Global functions .......................................................... 9
- ActionScript elements ................................................... 10
- Application class .......................................................... 15
- clearInterval() .............................................................. 46
- Client class ................................................................. 47
- File class ................................................................... 63
- getGlobal() ................................................................. 86
- load() ......................................................................... 87
- LoadVars class ............................................................... 87
- Log class .................................................................... 101
- NetConnection class ...................................................... 102
- protectObject() .............................................................. 112
- setAttributes() ............................................................... 113
- setInterval() ................................................................ 115
- SharedObject class ......................................................... 116
- SOAPCall class ............................................................... 140
- SOAPFault class ............................................................. 142
- Stream class ................................................................. 144
- trace() ....................................................................... 161
- WebService class ............................................................. 162
- XML class .................................................................. 166
- XMLSocket class ............................................................. 218
- XMLStreams class ........................................................... 227

**Appendix A: Server-Side Information Objects**

- Application information objects ...................................... 230
- NetConnection information objects ................................... 231
- Stream information objects .............................................. 232
Server-side ActionScript is a scripting language on the server that lets you develop efficient and flexible client-server Macromedia Flash Media Server applications. For example, you can use server-side ActionScript to control log-in procedures, control events in connected Macromedia Flash applications, determine what users see in their Flash applications, and communicate with other servers. You can also use server-side scripting to allow and disallow users access to various server-side application resources and to allow users to update and share information.

Server-side ActionScript is based on the ECMA-262 specification (ECMAScript 1.5), which is derived from JavaScript and lets you access the core JavaScript server object model. Server-side ActionScript provides global methods and classes and exposes a rich object model for developing media applications. You can also create classes, properties, and methods. This dictionary provides detailed information about the classes and their properties, methods, and events.

Client-side ActionScript is based on the ECMA-262 specification but deviates from the specification in several places. Server-side ActionScript does not deviate from the ECMA-262 specification. For information about the relationship between server-side ActionScript and client-side ActionScript, see Developing Media Applications.

Using server-side ActionScript

To use server-side ActionScript with a Flash Media Server application, you write the code, add the script file to the appropriate server directory, and run the SWF file that connects to the server. To understand the architecture of Flash Media Server and its applications, see Developing Media Applications.
Create the server-side ActionScript file and name it main.asc. All ActionScript code that is embedded in the script file and not in a function body executes once when the application is loaded but before the `application.onAppStart` event handler is called.

You can name your server-side ActionScript file app_name, where app_name is the name of your application's directory, and save it with a file extension of .asc or .js. Also, any double-byte characters (including characters of all Asian languages) in the server-side ActionScript file must be UTF-8-encoded. For more information on server-side script files, see Developing Media Applications.

For large-scale applications, you may need to use multiple server-side script files. You can use the Flash Media Server archive compiler utility to deploy those files as a package from a single location. For more information, see "Archiving and compiling server-side script files" in Developing Media Applications.

Server-side scripts also have a secure loading phase, during which critical code can be loaded before the main application loading phase. Server-side scripts also let you create protected objects; these are objects whose methods and data cannot be inspected or manipulated. These two features allow you to implement system calls that protect critical data and functions. For more information, see "Implementing secure system objects" in Developing Media Applications.

To install and test the server-side ActionScript file:

1. Locate the Macromedia Flash Media Server /applications directory.

   The default location of the /applications directory is in the Macromedia Flash Media Server product installation directory.

   If you did not accept the default installation settings and you aren’t sure where the /application directory is located, the location is specified in the `AppsDir` tag of the `Vhost.xml` file, which is located at `/Flash Media Server/conf/defaultRoot/defaultVhost`. For information about configuring a different application directory, see Managing Flash Media Server. Although your SWF and HTML files should be published under a web server directory, your server-side ASC files, your audio/video FLV files, and your ActionScript FLA source files should not be accessible to a user browsing your website.

2. Your server-side script file must be named main.asc, main.js, registered_app_name.asc, or registered_app_name.js.

3. Create a subdirectory in the /applications directory called `appName`, where `appName` is a name you choose as the filename of your Flash Media Server application. You must pass this name as a parameter to the `NetConnection.connect()` method in the client-side ActionScript.
4. Place the main.asc file in the appName directory or in a subdirectory called scripts in the appName directory.

5. Open the Flash application (the SWF file) in a browser or in the stand-alone Flash Player. The SWF file must contain ActionScript code that passes appName to the connect method of the NetConnection class, as shown in the following example:

   ```javascript
   nc = new NetConnection();
   nc.connect("rtmp://flashcomsvr.mydomain.com/myFlashComAppName");
   ```

### Using naming conventions

When you write server-side ActionScript code, there are certain naming conventions that you must use to name your applications, methods, properties, and variables. These rules let you logically identify objects so your code executes properly.

#### Naming applications

Flash Media Server application names must follow the Uniform Resource Identifier (URI) RFC 2396 convention. This convention supports a hierarchical naming system where a forward slash (/) separates the elements in the hierarchy. The first element specifies the application name. The element following the application name specifies the application instance name. Each instance of the application has its own script environment.

#### Specifying instances

By specifying a unique application instance name after an application name, you can run multiple instances of a single application. For example, rtmp://support/session215 specifies a customer support application named “support” and refers to a specific session of that application named “session215”. All users who connect to the same instance name can communicate with each other by referencing the same streams or shared objects.

#### Using JavaScript syntax

You must follow all syntax rules of JavaScript. For example, JavaScript is case-sensitive and does not allow punctuation other than underscores (_) and dollar signs ($) in names. You can use numbers in names, but names cannot begin with a number.

---

You can use the management console to check if the application loaded successfully.
Avoiding reserved commands

Flash Media Server has reserved commands that you cannot use in a script. These commands are either methods that belong to the client-side NetConnection class or methods that belong to the server-side Client class. This means that if you have a NetConnection object on the client (player), you cannot make the following call:

```-actionscript
nc.call("reservedCmd", ...);
```

In this call, "reservedCmd" is any of the following commands: `closeStream`, `connect`, `createStream`, `deleteStream`, `onStatus`, `pause`, `play`, `publish`, `receiveAudio`, `receiveVideo`, or `seek`. It also cannot be any of the server-side Client class methods: `getBandwidthLimit`, `setBandwidthLimit`, `getStats`, and `ping`.

ActionScript classes

The following table lists all the classes in the Server-Side ActionScript Language Reference.

<table>
<thead>
<tr>
<th>ActionScript class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application class</td>
<td>The Application class contains information about a Flash Media Server application instance that lasts until the application instance is unloaded.</td>
</tr>
<tr>
<td>Client class</td>
<td>The Client class lets you handle each user, or client, connection to a Flash Media Server application instance.</td>
</tr>
<tr>
<td>File class</td>
<td>The File class lets applications write to the server’s file system.</td>
</tr>
<tr>
<td>LoadVars class</td>
<td>The LoadVars class lets you load variables into a server-side script from a remote or local location.</td>
</tr>
<tr>
<td>Log class</td>
<td>The Log class lets you create a Log object that can be passed as an optional argument to the Constructor for the WebService class.</td>
</tr>
<tr>
<td>NetConnection class</td>
<td>The server-side NetConnection class lets you create a two-way connection between a Flash Media Server application instance and an application server, another Flash Media Server, or another Flash Media Server application instance on the same server.</td>
</tr>
<tr>
<td>SharedObject class</td>
<td>The Shared Object class lets you share data between multiple client applications in real time.</td>
</tr>
<tr>
<td>SOAPCall class</td>
<td>The SOAPCall class is the object type that is returned from all web service calls.</td>
</tr>
<tr>
<td>SOAPFault class</td>
<td>The SOAPFault class is the object type of the error object returned to <code>WebService.onFault</code> and <code>SOAPCall.onFault</code> functions.</td>
</tr>
</tbody>
</table>


The following table lists all the global functions in the *Server-Side ActionScript Language Reference*.

<table>
<thead>
<tr>
<th>Global function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clearInterval()</td>
<td>Cancels a time-out that was set with a call to the <code>setInterval()</code> method.</td>
</tr>
<tr>
<td>getGlobal()</td>
<td>Provides access to the global object while the secure.asc file is loading.</td>
</tr>
<tr>
<td>protectObject()</td>
<td>Protects user-defined or built-in objects behind a C wrapper object.</td>
</tr>
<tr>
<td>setAttributes()</td>
<td>Lets you prevent certain methods and properties from being enumerated, written, and deleted.</td>
</tr>
<tr>
<td>setInterval()</td>
<td>Continually calls a function or method at a specified time interval until the <code>clearInterval()</code> method is called.</td>
</tr>
<tr>
<td>trace()</td>
<td>Evaluates an expression and displays the value.</td>
</tr>
</tbody>
</table>
### ActionScript elements

Entries in this document are alphabetical by class name and then by method, property, or event handler name. The following table lists all classes, methods, properties, and event handlers individually in alphabetical order.

<table>
<thead>
<tr>
<th>ActionScript element</th>
<th>See entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>acceptConnection</td>
<td>Application.acceptConnection()</td>
</tr>
<tr>
<td>addHeader</td>
<td>NetConnection.addHeader()</td>
</tr>
<tr>
<td>addRequestHeader</td>
<td>LoadVars.addRequestHeader(), XML.addRequestHeader()</td>
</tr>
<tr>
<td>agent</td>
<td>Client.agent</td>
</tr>
<tr>
<td>allowDebug</td>
<td>Application.allowDebug</td>
</tr>
<tr>
<td>appendChild</td>
<td>XML.appendChild()</td>
</tr>
<tr>
<td>attributes</td>
<td>XML.attributes</td>
</tr>
<tr>
<td>autoCommit</td>
<td>SharedObject.autoCommit</td>
</tr>
<tr>
<td>broadcastMsg</td>
<td>Application.broadcastMsg()</td>
</tr>
<tr>
<td>bufferTime</td>
<td>Stream.bufferTime</td>
</tr>
<tr>
<td>call</td>
<td>Client.call(), NetConnection.call()</td>
</tr>
<tr>
<td>canAppend</td>
<td>File.canAppend</td>
</tr>
<tr>
<td>canRead</td>
<td>File.canRead</td>
</tr>
<tr>
<td>canReplace</td>
<td>File.canReplace</td>
</tr>
<tr>
<td>canWrite</td>
<td>File.canWrite</td>
</tr>
<tr>
<td>childNodes</td>
<td>XML.childNodes</td>
</tr>
<tr>
<td>clear</td>
<td>SharedObject.clear(), Stream.clear()</td>
</tr>
<tr>
<td>clearInterval</td>
<td>clearInterval()</td>
</tr>
<tr>
<td>clearSharedObject</td>
<td>Application.clearSharedObjects()</td>
</tr>
<tr>
<td>clearStreams</td>
<td>Application.clearStreams()</td>
</tr>
<tr>
<td>clients</td>
<td>Application.clients</td>
</tr>
<tr>
<td>cloneNode</td>
<td>XML.cloneNode()</td>
</tr>
<tr>
<td>close</td>
<td>NetConnection.close(), File.close(), SharedObject.close(), XMLSocket.close()</td>
</tr>
<tr>
<td><em>commandName</em></td>
<td>Client.&quot;commandName&quot;</td>
</tr>
<tr>
<td>commit</td>
<td>SharedObject.commit()</td>
</tr>
<tr>
<td>ActionScript element</td>
<td>See entry</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>config</td>
<td>Application.config</td>
</tr>
<tr>
<td>connect</td>
<td>NetConnection.connect(), XMLSocket.connect()</td>
</tr>
<tr>
<td>contentType</td>
<td>LoadVars.contentType, XML.contentType</td>
</tr>
<tr>
<td>copyTo</td>
<td>File.copyTo()</td>
</tr>
<tr>
<td>createElement</td>
<td>XML.createElement()</td>
</tr>
<tr>
<td>createTextNode</td>
<td>XML.createTextNode()</td>
</tr>
<tr>
<td>creationTime</td>
<td>File.creationTime</td>
</tr>
<tr>
<td>decode</td>
<td>LoadVars.decode()</td>
</tr>
<tr>
<td>detail</td>
<td>SOAPFault.detail</td>
</tr>
<tr>
<td>disconnect</td>
<td>Application.disconnect()</td>
</tr>
<tr>
<td>docTypeDecl</td>
<td>XML.docTypeDecl</td>
</tr>
<tr>
<td>eof</td>
<td>File.eof()</td>
</tr>
<tr>
<td>exists</td>
<td>File.exists</td>
</tr>
<tr>
<td>faultactor</td>
<td>SOAPFault.faultactor</td>
</tr>
<tr>
<td>faultcode</td>
<td>SOAPFault.faultcode</td>
</tr>
<tr>
<td>faultstring</td>
<td>SOAPFault.faultstring</td>
</tr>
<tr>
<td>firstChild</td>
<td>XML.firstChild</td>
</tr>
<tr>
<td>flush</td>
<td>File.flush(), SharedObject.flush(), Stream.flush()</td>
</tr>
<tr>
<td>gc</td>
<td>Application.gc()</td>
</tr>
<tr>
<td>get</td>
<td>SharedObject.get(), Stream.get()</td>
</tr>
<tr>
<td>getBandwidthLimit</td>
<td>Client.getBandwidthLimit()</td>
</tr>
<tr>
<td>getBytesLoaded</td>
<td>LoadVars.getBytesLoaded(), XML.getBytesLoaded()</td>
</tr>
<tr>
<td>getBytesTotal</td>
<td>LoadVars.getBytesTotal(), XML.getBytesTotal()</td>
</tr>
<tr>
<td>getNamespaceForPrefix</td>
<td>XML.getNamespaceForPrefix()</td>
</tr>
<tr>
<td>getPrefixForNamespace</td>
<td>XML.getPrefixForNamespace()</td>
</tr>
<tr>
<td>getProperty</td>
<td>SharedObject.getProperty()</td>
</tr>
<tr>
<td>getPropertyNames</td>
<td>SharedObject.getPropertyNames()</td>
</tr>
<tr>
<td>getStats</td>
<td>Application.getStats(), Client.getStats()</td>
</tr>
<tr>
<td>handlerName</td>
<td>SharedObject.handlerName</td>
</tr>
<tr>
<td>hasChildNodes</td>
<td>XML.hasChildNodes()</td>
</tr>
<tr>
<td><strong>ActionScript element</strong></td>
<td><strong>See entry</strong></td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td><code>hostName</code></td>
<td><code>Application.hostname</code></td>
</tr>
<tr>
<td><code>ignoreWhite</code></td>
<td><code>XML.ignoreWhite</code></td>
</tr>
<tr>
<td><code>insertBefore</code></td>
<td><code>XML.insertBefore()</code></td>
</tr>
<tr>
<td><code>ip</code></td>
<td><code>Client.ip</code></td>
</tr>
<tr>
<td><code>isConnected</code></td>
<td><code>NetConnection.isConnected</code></td>
</tr>
<tr>
<td><code>isDirectory</code></td>
<td><code>File.isDirectory</code></td>
</tr>
<tr>
<td><code>isDirty</code></td>
<td><code>SharedObject.isDirty</code></td>
</tr>
<tr>
<td><code>isFile</code></td>
<td><code>File.isFile</code></td>
</tr>
<tr>
<td><code>isOpen</code></td>
<td><code>File.isOpen</code></td>
</tr>
<tr>
<td><code>lastChild</code></td>
<td><code>XML.lastChild</code></td>
</tr>
<tr>
<td><code>lastModified</code></td>
<td><code>File.lastModified</code></td>
</tr>
<tr>
<td><code>length</code></td>
<td><code>File.length</code>, <code>Stream.length()</code></td>
</tr>
<tr>
<td><code>list</code></td>
<td><code>File.list()</code></td>
</tr>
<tr>
<td><code>load</code></td>
<td><code>load()</code>, <code>LoadVars.load()</code>, <code>XML.load()</code></td>
</tr>
<tr>
<td><code>loaded</code></td>
<td><code>LoadVars.loaded</code>, <code>XML.loaded</code></td>
</tr>
<tr>
<td><code>localName</code></td>
<td><code>XML.localName</code></td>
</tr>
<tr>
<td><code>lock</code></td>
<td><code>SharedObject.lock()</code></td>
</tr>
<tr>
<td><code>mark</code></td>
<td><code>SharedObject.mark()</code></td>
</tr>
<tr>
<td><code>mkdir</code></td>
<td><code>File.mkdir()</code></td>
</tr>
<tr>
<td><code>mode</code></td>
<td><code>File.mode</code></td>
</tr>
<tr>
<td><code>name</code></td>
<td><code>Application.name</code>, <code>File.name</code>, <code>SharedObject.name</code>, <code>Stream.name</code></td>
</tr>
<tr>
<td><code>namespaceURI</code></td>
<td><code>XML.namespaceURI</code></td>
</tr>
<tr>
<td><code>nextSibling</code></td>
<td><code>XML.nextSibling</code></td>
</tr>
<tr>
<td><code>nodeName</code></td>
<td><code>XML.nodeName</code></td>
</tr>
<tr>
<td><code>nodeType</code></td>
<td><code>XML.nodeType</code></td>
</tr>
<tr>
<td><code>nodeValue</code></td>
<td><code>XML.nodeValue</code></td>
</tr>
<tr>
<td><code>onAppStart</code></td>
<td><code>Application.onAppStart</code></td>
</tr>
<tr>
<td><code>onAppStop</code></td>
<td><code>Application.onAppStop</code></td>
</tr>
<tr>
<td><code>onClose</code></td>
<td><code>XMLSocket.onClose</code></td>
</tr>
<tr>
<td><strong>ActionScript element</strong></td>
<td><strong>See entry</strong></td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>onConnect</td>
<td>Application.onConnect, XMLSocket.onConnect</td>
</tr>
<tr>
<td>onConnectAccept</td>
<td>Application.onConnectAccept</td>
</tr>
<tr>
<td>onConnectReject</td>
<td>Application.onConnectReject</td>
</tr>
<tr>
<td>onData</td>
<td>LoadVars.onData, XML.onData, XMLSocket.onData</td>
</tr>
<tr>
<td>onDisconnect</td>
<td>Application.onDisconnect</td>
</tr>
<tr>
<td>onFault</td>
<td>SOAPCall.onFault, WebService.onFault</td>
</tr>
<tr>
<td>onHTTPStatus</td>
<td>LoadVars.onHTTPStatus, XML.onHTTPStatus</td>
</tr>
<tr>
<td>onLoad</td>
<td>LoadVars.onLoad, WebService.onLoad, XML.onLoad</td>
</tr>
<tr>
<td>onLog</td>
<td>Log.onLog</td>
</tr>
<tr>
<td>onResult</td>
<td>SOAPCall.onResult</td>
</tr>
<tr>
<td>onStatus</td>
<td>Application.onStatus, NetConnection.onStatus, SharedObject.onStatus, Stream.onStatus</td>
</tr>
<tr>
<td>onSync</td>
<td>SharedObject.onSync</td>
</tr>
<tr>
<td>onXML</td>
<td>XMLSocket.onXML</td>
</tr>
<tr>
<td>open</td>
<td>File.open()</td>
</tr>
<tr>
<td>parentNode</td>
<td>XML.parentNode</td>
</tr>
<tr>
<td>parseXML</td>
<td>XML.parseXML()</td>
</tr>
<tr>
<td>ping</td>
<td>Client.readAccess</td>
</tr>
<tr>
<td>play</td>
<td>Stream.play()</td>
</tr>
<tr>
<td>position</td>
<td>File.position</td>
</tr>
<tr>
<td>prefix</td>
<td>XML.prefix</td>
</tr>
<tr>
<td>previousSibling</td>
<td>XML.previousSibling</td>
</tr>
<tr>
<td>protocol</td>
<td>Client.protocol</td>
</tr>
<tr>
<td>purge</td>
<td>SharedObject.purge()</td>
</tr>
<tr>
<td>read</td>
<td>File.read()</td>
</tr>
<tr>
<td>readAccess</td>
<td>Client.readAccess</td>
</tr>
<tr>
<td>readAll</td>
<td>File.readAll()</td>
</tr>
<tr>
<td>readByte</td>
<td>File.readByte()</td>
</tr>
<tr>
<td>readln</td>
<td>File.readln()</td>
</tr>
<tr>
<td>record</td>
<td>Stream.record()</td>
</tr>
<tr>
<td>ActionScript element</td>
<td>See entry</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>referrer</td>
<td>Client.referrer</td>
</tr>
<tr>
<td>registerClass</td>
<td>Application.registerClass()</td>
</tr>
<tr>
<td>registerProxy</td>
<td>Application.registerProxy()</td>
</tr>
<tr>
<td>rejectConnection</td>
<td>Application.rejectConnection()</td>
</tr>
<tr>
<td>remove</td>
<td>File.remove()</td>
</tr>
<tr>
<td>removeNode</td>
<td>XML.removeNode()</td>
</tr>
<tr>
<td>renameTo</td>
<td>File.renameTo()</td>
</tr>
<tr>
<td>request</td>
<td>SOAPCall.request</td>
</tr>
<tr>
<td>__resolve</td>
<td>Client.__resolve</td>
</tr>
<tr>
<td>response</td>
<td>SOAPCall.response</td>
</tr>
<tr>
<td>resyncDepth</td>
<td>SharedObject.resyncDepth</td>
</tr>
<tr>
<td>secure</td>
<td>Client.secure</td>
</tr>
<tr>
<td>seek</td>
<td>getGlobal()</td>
</tr>
<tr>
<td>send</td>
<td>LoadVars.send(), SharedObject.send(), Stream.send(), XML.send(), XMLSocket.send()</td>
</tr>
<tr>
<td>sendAndLoad</td>
<td>LoadVars.sendAndLoad(), XML.sendAndLoad()</td>
</tr>
<tr>
<td>server</td>
<td>Application.server</td>
</tr>
<tr>
<td>setAttributes</td>
<td>setAttributes()</td>
</tr>
<tr>
<td>setBandwidthLimit</td>
<td>Client.setBandwidthLimit()</td>
</tr>
<tr>
<td>setBufferTime</td>
<td>Stream.setBufferTime()</td>
</tr>
<tr>
<td>setInterval</td>
<td>setInterval()</td>
</tr>
<tr>
<td>setProperty</td>
<td>SharedObject.setProperty()</td>
</tr>
<tr>
<td>setVirtualPath</td>
<td>Stream.setVirtualPath()</td>
</tr>
<tr>
<td>shutdown</td>
<td>Application.shutdown()</td>
</tr>
<tr>
<td>size</td>
<td>SharedObject.size(), Stream.size()</td>
</tr>
<tr>
<td>server</td>
<td>Application.server</td>
</tr>
<tr>
<td>status</td>
<td>XML.status</td>
</tr>
<tr>
<td>syncWrite</td>
<td>Stream.syncWrite</td>
</tr>
<tr>
<td>toString</td>
<td>File.toString(), LoadVars.toString(), XML.toString()</td>
</tr>
<tr>
<td>trace</td>
<td>trace()</td>
</tr>
</tbody>
</table>
Application class

**Availability**
Flash Communication Server 1.

The Application class contains information about a Flash Media Server application instance that lasts until the application instance is unloaded. A Flash Media Server application is a collection of stream objects, shared objects, and clients (connected users). Each application has a unique name, and you can use a naming scheme to create multiple instances of an application. For more information about the naming scheme, see “Using naming conventions” on page 7.

Every instance of a Flash Media Server application has an application object, which is a single instance of the Application class. You don't need to use a constructor function to create the application object; it is created automatically when an application is instantiated by the server. Use the following syntax to call the methods, properties, and event handlers of the Application class:

```ActionScript
class Application {
  // Code...
}
```

Use the application object to accept and reject client connection attempts, register and unregister classes and proxies, and create functions that are invoked when an application starts or stops or when a client connects or disconnects.

For more information about the life cycle of an application, see “Application flow” in Developing Media Applications.

### ActionScript element | See entry
---|---
type | File.type
unlock | SharedObject.unlock()
uri | NetConnection.uri, Client.uri
version | SharedObject.version
virtualKey | Client.virtualKey
write | File.write()
writeAccess | Client.writeAccess
writeAll | File.writeAll()
writeByte | File.writeByte()
writeln | File.writeln()
xmlDecl | XML.xmlDecl
Besides the built-in properties of the Application class, you can create other properties of any legal ActionScript type, including references to other ActionScript objects. For example, the following code creates a new property of type `Array` and a new property of type `Number`:

```actionscript
application.myarray = new Array();
application.num_requests = 1;
```

### Method summary for the Application class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Application.acceptConnection()</code></td>
<td>Accepts a connection to an application from a client.</td>
</tr>
<tr>
<td><code>Application.broadcastMsg()</code></td>
<td>Broadcasts a message to all connected clients.</td>
</tr>
<tr>
<td><code>Application.clearSharedObjects()</code></td>
<td>Clears all shared objects associated with the current instance.</td>
</tr>
<tr>
<td><code>Application.clearStreams()</code></td>
<td>Clears all stream objects associated with the current instance.</td>
</tr>
<tr>
<td><code>Application.disconnect()</code></td>
<td>Disconnects a client from the server.</td>
</tr>
<tr>
<td><code>Application.gc()</code></td>
<td>Invokes the garbage collector to reclaim any unused resources for this application instance.</td>
</tr>
<tr>
<td><code>Application.getStats()</code></td>
<td>Returns network statistics for the application instance.</td>
</tr>
<tr>
<td><code>Application.registerClass()</code></td>
<td>Registers or unregisters a constructor that is called during object deserialization.</td>
</tr>
<tr>
<td><code>Application.registerProxy()</code></td>
<td>Registers a NetConnection or Client object to fulfill a method request.</td>
</tr>
<tr>
<td><code>Application.rejectConnection()</code></td>
<td>Rejects a connection to an application.</td>
</tr>
<tr>
<td><code>Application.shutdown()</code></td>
<td>Unloads the application instance.</td>
</tr>
</tbody>
</table>

### Property summary for the Application class

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Application.allowDebug</code></td>
<td>A Boolean value that lets administrators use the <code>approveDebugSession()</code> <code>Server Management</code> <code>ActionScript</code> method to access your Flash Media Server application (<code>true</code>) or not (<code>false</code>).</td>
</tr>
<tr>
<td><code>Application.clients</code></td>
<td>Read-only; an object containing a list of all clients currently connected to the application.</td>
</tr>
</tbody>
</table>
Event handler summary for the Application class

<table>
<thead>
<tr>
<th>Event handler</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application.onAppStart</td>
<td>Invoked when the application is loaded by the server.</td>
</tr>
<tr>
<td>Application.onAppStop</td>
<td>Invoked when the application is unloaded by the server.</td>
</tr>
<tr>
<td>Application.onConnect</td>
<td>Invoked when a client connects to the application.</td>
</tr>
<tr>
<td>Application.onConnectAccept</td>
<td>Invoked when a client successfully connects to the application; for use with media components only.</td>
</tr>
<tr>
<td>Application.onConnectReject</td>
<td>Invoked when a client fails to connect to the application; for use with communication components only.</td>
</tr>
<tr>
<td>Application.onDisconnect</td>
<td>Invoked when a client disconnects from the application.</td>
</tr>
<tr>
<td>Application.onStatus</td>
<td>Invoked when a script generates an error.</td>
</tr>
</tbody>
</table>

Application.acceptConnection()

**Availability**
Flash Communication Server MX 1.0.

**Usage**
application.acceptConnection(clientObj)

**Parameters**

- **clientObj** A Client object; a client to accept.

**Returns**
Nothing.
Description
Method; accepts the connection call from a client to the server. The application.onConnect event handler is invoked on the server side to notify a script when NetConnection.connect() is called from the client side. You can use the application.acceptConnection() method in an application.onConnect event handler to accept a connection from a client. You can use the application.acceptConnection() method outside an application.onConnect event handler to accept a client connection that had been placed in a pending state (for example, to verify a user name and password).

When you use components and your code includes an explicit call to application.acceptConnection() or application.rejectConnection(), the last line (in order of execution) of the onConnect event handler should be either application.acceptConnection() or application.rejectConnection(). Also, any logic that follows the explicit acceptConnection or rejectConnection statement must be placed in application.onConnectAccept and application.onConnectReject statements, or it will be ignored. This requirement exists only when you use components.

Example
The following example uses the application.acceptConnection() method to accept the connection from client1:

application.onConnect = function (client1){
  // Insert code here.
  application.acceptConnection(client1);
  client1.call("welcome");
};

This example shows code from an application that does not use components.

See also
Application.onConnect, Application.rejectConnection()

Application.allowDebug

Availability
Flash Media Server 2.

Usage
application.allowDebug
Description
Property; a Boolean value that allows Administrators to access your application with the approveDebugSession() Server Management API (true) or not (false). A debug connection displays information about shared objects and streams.
For information about creating a debug connection, see NetConnection.connect() and approveDebugSession().

Application.broadcasterMsg()

Availability
Flash Media Server 2.

Usage
application.broadcastMsg(cmd [, p1, p2, ...., pN])

Parameters

- cmd A string; a message to broadcast.
- p1 A string; additional messages.

Returns
Nothing.

Description
Method; broadcasts a message to all connected clients.
This method is equivalent to looping through the Application.clients array and calling Client.call() on each individual client, but is more efficient (especially for large number of connected clients). The only difference is that you can’t specify a response object when you call broadcastMsg(), otherwise, the syntax is the same.

Shared objects can handle broadcast messages with the SharedObject.handlerName property.

Example
The following server-side code sends a message to the client:
application.broadcastMsg("handlerName", "Hello World");
The following client-side code catches the message and prints it in the Output panel:
nc = new NetConnection();
nc.handlerName = function(msg) { trace(msg); }
// traces out "Hello World"
See also

**SharedObject.handlerName**

**Application.clearSharedObjects()**

**Availability**

Flash Communication Server MX 1.0.

**Usage**

```javascript
application.clearSharedObjects(soPath)
```

**Parameters**

- **soPath** A string that indicates the URI of a shared object.

**Returns**

A Boolean value of `true` if the shared object at the specified path was deleted; otherwise, `false`. If using wildcard characters to delete multiple files, the method returns `true` only if all the shared objects matching the wildcard pattern were successfully deleted; otherwise, it will return `false`.

**Description**

Method; deletes persistent shared objects (FSO files) specified by the `soPath` parameter and clears all properties from active shared objects (persistent and nonpersistent).

The `soPath` parameter specifies the name of a shared object, which can include a slash (`/`) as a delimiter between directories in the path. The last element in the path can contain wildcard patterns (for example, a question mark (`?`) and an asterisk (`*`)) or a shared object name. The `application.clearSharedObjects()` method traverses the shared object hierarchy along the specified path and clears all the shared objects. Specifying a slash (`/`) clears all the shared objects associated with an application instance.

The following values are possible for the `soPath` parameter:

- `/` clears all local and persistent shared objects associated with the instance.
- `/foo/bar` clears the shared object `/foo/bar`; if bar is a directory name, no shared objects are deleted.
- `/foo/bar/*` clears all shared objects stored under the instance directory `/foo/bar`. The bar directory is also deleted if no persistent shared objects are in use within this namespace.
- `/foo/bar/XX??` clears all shared objects that begin with `XX`, followed by any two characters. If a directory name matches this specification, all the shared objects within this directory are cleared.
If you call the `clearSharedObjects()` method and the specified path matches a shared object that is currently active, all its properties are deleted, and a “clear” event is sent to all subscribers of the shared object. If it is a persistent shared object, the persistent store is also cleared.

**Example**
The following example clears all the shared objects for an instance:
```javascript
function onApplicationStop()
  application.clearSharedObjects("/");
}
```

**See also**
`SharedObject.clear()`

### Application.clearStreams()

**Availability**
Flash Communication Server MX 1.0.

**Usage**
`application.clearStreams(streamPath)`

**Parameters**
- `streamPath` A string that indicates the URI of a stream.

**Returns**
A Boolean value of `true` if the stream at the specified path was deleted; otherwise, `false`. If using wildcard characters to clear multiple stream files, the method returns `true` only if all the streams matching the wildcard pattern were successfully deleted; otherwise it returns `false`.

**Description**
Method; clears recorded streams (FLV) files and MP3 files associated with the application instance. You can use this method to clear a single stream, all streams associated with the application instance, just those streams in a specific subdirectory of the application instance, or just those streams whose names match a specified wildcard pattern.
The `streamPath` parameter specifies the location and name of a stream, relative to the application's instance directory. You can include a slash (/) as a delimiter between directories in the path. The last element in the path can contain wildcard patterns (for example, a question mark (?) and an asterisk (*) or a stream name. The `application.clearStreams()` method traverses the stream hierarchy along the specified path and clears all the recorded streams that match the given wildcard pattern. Specifying a slash clears all the streams associated with an application instance.

To clear MP3 files associated with the application instance, precede the path to the stream with `mp3:` (for example, `mp3:/streamPath`). By default, the `application.clearStreams()` method clears only recorded FLV streams. You can also explicitly clear only FLV streams by placing `flv:` before the stream path (for example, `flv:/streamPath`). See the following Example section for examples of clearing FLV and MP3 files.

The following examples show some possible values for the `streamPath` parameter:

- `/` or `flv:/` clears all recorded (FLV) streams associated with the instance.
- `/report` clears the stream file `report.flv` from the application instance directory.
- `/presentations/intro` clears the recorded stream file `intro.flv` from the application instance's `/presentations` subdirectory; if `intro` is a directory name, no streams are deleted.
- `/presentations/*` clears all recorded stream files stored from the application instance's `/presentations` subdirectory. The `/presentation` subdirectory is also deleted if no streams are used in this namespace.
- `mp3:/` clears all MP3 files associated with the instance.
- `mp3:/mozart/requiem` clears the MP3 file named `requiem.mp3` from the application instance's `/mozart` subdirectory.
- `mp3:/mozart/*` clears all MP3 file from the application instance's `/mozart` subdirectory.
- `/presentations/report??` clears all recorded (FLV) streams that begin with “report”, followed by any two characters. If there are directories within the given directory listing, the directories are cleared of any streams that match `report??`.

If an `application.clearStreams()` method is invoked on a stream that is currently recording, the recorded file is set to length 0 (cleared), and the internal cached data is also cleared.

**NOTE** You can also use the Server Management ActionScript API `removeApp` method to delete all the resources for a single instance.
Example
The following example clears all recorded streams:

```javascript
function onApplicationStop(){
    application.clearStreams("/");
}
```

The following example clears all MP3 files from the application instance’s /disco subdirectory:

```javascript
function onApplicationStop(){
    application.clearStreams("mp3:/disco/*");
}
```

Application.clients

Availability
Flash Communication Server MX 1.0.

Usage
application.clients

Description
Property (read-only); an object containing all the Flash clients or other Flash Media Servers currently connected to the application. The object is a custom object like an array, but with only one property, length. Each element in the object is a reference to a Client object instance, and you can use the length property to determine the number of users connected to the application. You can use the array access operator ([ ]) with the application.clients property to access elements in the object.

The object used for the clients property is not an array, but it acts the same except for one difference: you can’t use the following syntax to iterate through the object:

```javascript
for(var i in application.clients) {
    // Insert code here.
}
```

Instead, use the following code to loop through each element in a clients object:

```javascript
for (var i = 0; i < application.clients.length; i++) {
    // Insert code here.
}
```
Example
The following example uses a for loop to iterate through each member of the application.clients array and calls the method serverUpdate on each client:

```javascript
for (i = 0; i < application.clients.length; i++)
  application.clients[i].call("serverUpdate");
```

Application.config

Availability
Flash Media Server 2.

Usage
application.config

Description
Property; lets you access properties of the ApplicationObject tag in the Application.xml configuration file. For more information, see “Configurable application object properties for server-side scripting” in Managing Flash Media Server.

Example
Use the following sample <ApplicationObject> tag in the Application.xml file for this Application.config example:

```xml
<Application>
  <JSEngine>
    <ApplicationObject>
      <config>
        <user_name>jdoe</user_name>
        <dept_name>engineering</dept_name>
      </config>
    </ApplicationObject>
  </JSEngine>
</Application>
```

Either of the following lines of code access the same Application properties defined in the previous Application.xml file example:

```javascript
trace("I am " + application.config.user + " and I work in the " + application.config.dept_name + " department.");

trace("I am " + application.config["user"] + " and I work in the " + application.config["dept_name"] + " department.");
```

The following code is sent to the application log file and Application inspector:
I am jdoe and I work in the engineering department.
Application.disconnect()  

**Availability**  
Flash Communication Server MX 1.0.

**Usage**  
`application.disconnect(clientObj)`

**Parameters**  
`clientObj` The client to disconnect. The object must be a Client object from the `application.clients` array.

**Returns**  
A Boolean value of `true` if the disconnect was successful; otherwise, `false`.

**Description**  
Method; causes the server to terminate a client connection to the application. When this method is called, `NetConnection.onStatus` is invoked on the client side with a status message of `NetConnection.Connection.Closed`. The `application.onDisconnect` handler is also invoked.

**Example**  
The following example calls the `application.disconnect()` method to disconnect all users from an application instance:
```javascript
function disconnectAll(){
    for (i=0; i < application.clients.length; i++){
        application.disconnect(application.clients[i]);
    }
}
```

Application.gc()  

**Availability**  
Flash Media Server 2.

**Usage**  
`application.gc()`

**Parameters**  
None.
Returns
Nothing.

Description
Method; invokes the garbage collector to reclaim any unused resources for this application instance.

Application.getStats()

Availability
Flash Communication Server MX 1.0.

Usage
application.getStats()

Returns
An ActionScript object with various properties for each statistic returned.

Description
Method; returns statistics for the application instance including the total number of bytes sent and received, the number of RTMP messages sent and received, the number of dropped messages, the number of clients connected to the application instance, and the number of clients who have disconnected from the application instance.

Example
The following example uses Application.getStats() to output the application instance's statistics:

```actionscript
stats = application.getStats();
trace("Total bytes received: " + stats.bytes_in);
trace("Total bytes sent: " + stats.bytes_out);
trace("RTMP messages received: " + stats.msg_in);
trace("RTMP messages sent: " + stats.msg_out);
trace("RTMP messages dropped: " + stats.msg_dropped);
trace("Total clients connected: " + stats.total_connects);
trace("Total clients disconnected: " + stats.total_disconnects);
```

Application.hostname

Availability
Flash Media Server MX 1.5
**Usage**  
`application.hostname`

**Description**  
Property (read-only); contains the host name of the server for default virtual hosts and the virtual host name for non-default virtual hosts.

**Example**  
The following example traces the name of the host running the current application:  
```
trace(application.hostname)
```

---

**Application.name**

**Availability**  
Flash Communication Server MX 1.0.

**Usage**  
`application.name`

**Description**  
Property (read-only); contains the name of the Flash Media Server application instance.

**Example**  
The following example checks the `name` property against a specific string before it executes some code:  
```
if (application.name == "videomail/work"){
    // Insert code here.
}
```

---

**Application.onAppStart**

**Availability**  
Flash Communication Server MX 1.0.

**Usage**  
`application.onAppStart = function (){}`

**Parameters**  
None.
Returns
Nothing.

Description
Event handler; invoked when the server first loads the application instance. You use this handler to initialize an application state. You can use `application.onAppStart` and `application.onAppStop` to initialize and clean up global variables in an application because each of these events is invoked only once during the lifetime of an application instance.

Example
The following example defines an anonymous function for the `application.onAppStart` event handler that sends a `trace` message:

```javascript
application.onAppStart = function () {
  trace("onAppStart called");
};
```

Application.onAppStop

Availability
Flash Communication Server MX 1.0.

Usage
`application.onAppStop = function (info){}`

Parameters
- `info` An information object that explains why the application stopped running. See “Server-Side Information Objects” on page 229.

Returns
The value returned by the function you define, if any, or `null`. To refuse to unload the application, return `false`. To unload the application, return `true` or any non-`false` value.

Description
Event handler; invoked when the application is about to be unloaded by the server. You can define a function that executes when the event handler is invoked. If the function returns `true`, the application unloads. If the function returns `false`, the application doesn’t unload. If you don’t define a function for this event handler, or if the return value is not a Boolean value, the application is unloaded when the event is invoked.
The Flash Media Server application passes an information object to the `application.onAppStop` event. You can use server-side ActionScript to look at this information object to decide what to do in the function you define. You could also define the `application.onAppStop` event to notify users before shutdown.

If you use the management console or the Server Management ActionScript API to unload a Flash Media Server application, `application.onAppStop` is not invoked. Therefore you cannot use the `application.onAppStop` event, for example, to tell users that the application is exiting.

**Example**
The following example defines a function to perform the shutdown operations on the application. The function is then assigned to the event handler so that it executes when the handler is invoked.

```actionscript
function onMyApplicationEnd(info):
    // Do all the application-specific shutdown logic here.

application.onAppStop = onMyApplicationEnd;
```

**Application.onConnect**

**Availability**
Flash Communication Server MX 1.0.

**Usage**
```
application.onConnect = function (clientObj [, p1, ...., pN]){}
```

**Parameters**
- `clientObj` A Client object. The client connecting to the application.
- `p1, ...., pN` Optional parameters passed to the `application.onConnect` handler. These parameters are passed from the client-side `NetConnection.connect` handler when a client connects to the application.
Returns
The value you provide. If you return a Boolean value of `true`, the server accepts the connection; if the value is `false`, the server rejects the connection. If you return `null` or no return value, the server puts the client in a pending state and the client can’t receive or send messages. If the client is put in a pending state, you must call `application.acceptConnection()` or `application.rejectConnection()` at a later time to accept or reject the connection. For example, you can perform external authentication by making a NetConnection call in your `application.onConnect` event handler to an application server and having the reply handler call `application.acceptConnection()` or `application.rejectConnection()`, depending on the information received by the reply handler.

You can also call `application.acceptConnection` or `application.rejectConnection` in the `application.onConnect` event handler. If you do, any value returned by the function is ignored.

| NOTE | Returning 1 or 0 is not the same as returning `true` or `false`. The values 1 and 0 are treated the same as any other integers and do not accept or reject a connection. |
How to use `application.onConnect` to accept, reject, or put a client in a pending state.

Description

Event handler; invoked on the server side when `NetConnection.connect` is called from the client side and a client attempts to connect to an application instance. You can define a function for the `application.onConnect` event handler. If you don’t define a function, connections are accepted by default. If the server accepts the new connection, the `application.clients` object is updated.

You can use the `application.onConnect` event in server-side scripts to perform authentication. All the information required for authentication should be sent to the server by the client as parameters (`p1`, ..., `pN`) that you define. In addition to authentication, the script can set the access rights to all server-side objects that this client can modify by setting the `Client.readAccess` and `Client.writeAccess` properties.
If there are several simultaneous connection requests for an application, the server serializes the requests so there is only one `application.onConnect` handler executing at a time. It is a good idea to write code for the `application.onConnect` function that executes quickly to prevent a long connection time for clients.

**Example**

The following examples show the three basic uses of the `onConnect` handler:

**(Usage 1)**

```javascript
application.onConnect = function (clientObj [, p1, ..., pN]){
    // Insert code here to call methods that do authentication.
    // Returning null puts the client in a pending state.
    return null;
};
```

**(Usage 2)**

```javascript
application.onConnect = function (clientObj [, p1, ..., pN]){
    // Insert code here to call methods that do authentication.
    // The following code accepts the connection:
    application.acceptConnection(clientObj);
};
```

**(Usage 3)**

```javascript
application.onConnect = function (clientObj [, p1, ..., pN])
{
    // Insert code here to call methods that do authentication.
    // The following code accepts the connection by returning true:
    return true;
};
```

The following example verifies that the user has sent the password “XXXX”. If the password is sent, the user’s access rights are modified, and the user can complete the connection. In this case, the user can create or write to streams and shared objects in the user’s own directory and can read or view any shared object or stream in this application instance.

```javascript
// This code should be placed in the global scope.

application.onConnect = function (newClient, userName, password){
    // Do all the application-specific connect logic.
    if (password == "XXXX"){
        newClient.writeAccess = "/" + userName;
        this.acceptConnection(newClient);
    } else {
        var err = new Object();
        err.message = "Invalid password";
        this.rejectConnection(newClient, err);
    }
};
```

**NOTE**

If you are using the Component framework (that is, you are loading the `components.asc` file in your server-side script file) you must use the `Application.onConnectAccept` method to accept client connections. For more information see `Application.onConnectAccept`. 

---

32 Server-Side ActionScript Language Reference
If the password is incorrect, the user is rejected and an information object with a `message` property set to “Invalid password” is returned to the client side. The object is assigned to `infoObject.application`. To access the `message` property, use the following code on the client side:

```javascript
ClientCom.onStatus = function (info){
    trace(info.application.message);
    // Prints "Invalid password"
    // in the Output panel on the client-side.
};
```

See also

`Application.acceptConnection()`, `Application.onConnectAccept`, `Application.onConnectReject`, `Application.rejectConnection()`

**Application.onConnectAccept**

**Availability**

Flash Media Server (with communication components only).

**Usage**

```javascript
application.onConnectAccept = function (clientObj [,p1, .... pN]){
```

**Parameters**

- `clientObj` A Client object; the client connecting to the application.
- `p1...pN` Optional parameters passed to the `application.onConnectAccept()` method. These parameters are passed from the client-side `NetConnection.connect()` method when a client connects to the application.

**Returns**

Nothing.

**Description**

Event handler; invoked only when components are used (that is, when the components.asc script is loaded into your server-side script) when `NetConnection.connect()` is called from the client-side and a client successfully connects to an application instance.

Use `onConnectAccept` to handle the result of an accepted connection in an application that contains components.
If you don’t use the Flash Media Server components framework, you can execute code in the application.onConnect handler after accepting or rejecting the connection. When you use the components framework, however, any code you want to execute after the connection is accepted or rejected must be placed in the event handlers application.onConnectAccept and application.onConnectReject. This architecture allows all the components to decide whether a connection is accepted or rejected.

Example
The following example is the client-side code you can use for an application:
```javascript
nc = new NetConnection();
nc.connect("rtmp://test","jlopes");

nc.onStatus = function(info) {
    trace(info.code);
};

nc.doSomething = function(){
    trace("doSomething called!");
}

The following example is server-side code you can include in the main.asc file:
```javascript
// When using components, always load components.asc.
load("components.asc");

application.onConnect = function(client, username){
    trace("onConnect called");
gFrameworkFC.getClientGlobals(client).username = username;
if (username == "hacker") {
    application.rejectConnection(client);
} else {
    application.acceptConnection(client);
}

// Code is in onConnectAccept and onConnectReject statements
// because components are used.
application.onConnectAccept = function(client, username){
    trace("Connection accepted for "+username);
    client.call("doSomething",null);
}

application.onConnectReject = function(client, username){
    trace("Connection rejected for "+username);
}
```
See also
Application.acceptConnection(), Application.onConnect,
Application.onConnectReject, Application.rejectConnection()

Application.onConnectReject

Availability
Flash Media Server (with communication components only).

Usage
application.onConnectReject = function (clientObj [,p1, ..., pN]){}

Parameters
clientObj  A Client object; the client connecting to the application.
p1...pN   Optional parameters passed to the application.onConnectReject handler.
These parameters are passed from the client-side NetConnection.connect() method when a
client connects to the application.

Returns
Nothing.

Description
Event handler; invoked only when communication components are used (that is, when the
components.asc script is loaded into your server-side script) when
NetConnection.connect() is called from the client side and a client fails to connect to an
application instance.

Use onConnectReject to handler the result of an rejected connection in an application that
contains components.

If you don't use the Flash Media Server components framework, you can execute code in the
application.onConnect handler after accepting or rejecting a connection. When you use
the components framework, however, any code you want to execute after the connection is
accepted or rejected must be placed in the framework event handlers
application.onConnectAccept and application.onConnectReject. This architecture
allows all the components to decide whether a connection is accepted or rejected.
Example

The following example is the client-side code you can use for an application:

```javascript
nc = new NetConnection();
nc.connect("rtmp://test","jlopes");
nc.onStatus = function(info) {
   trace(info.code);
};
nc.doSomething = function(){
   trace("doSomething called!");
}
```

The following example is the server-side code you can include in the main.asc file:

```javascript
// When using components, always load components.asc.
load("components.asc");

application.onConnect = function(client, username){
   trace("onConnect called");
   gFrameworkFC.getClientGlobals(client).username = username;
   if (username == "hacker") {
      application.rejectConnection(client);
   } else {
      application.acceptConnection(client);
   }
}

application.onConnectAccept = function(client, username){
   trace("Connection accepted for "+username);
   client.call("doSomething",null);
}

application.onConnectReject = function(client, username){
   trace("Connection rejected for "+username);
}
```

See also

(Application.acceptConnection(), Application.onConnect,
Application.onConnectAccept, Application.rejectConnection())
Application.onDisconnect

Availability
Flash Communication Server MX 1.0.

Usage
application.onDisconnect = function (clientObj){}

Parameters
clientObj  A Client object; a client disconnecting from the application.

Returns
The server ignores any return value.

Description
Event handler; invoked when a client disconnects from an application. You can use this event handler to flush any client state information or to notify other users of this event. This event handler is optional.

Example
The following example uses an anonymous function and assigns it to the application.onDisconnect event handler:

```
// This code should be placed in the global scope.
application.onDisconnect = function (client){
  // Do all the client-specific disconnect logic.
  // Insert code here.
  trace("user disconnected");
};
```

Application.onStatus

Availability
Flash Communication Server MX 1.0.

Usage
application.onStatus = function (infoObj){}

Parameters
infoObject  An object that contains the error level, code, and sometimes a description. For more information, see “Server-Side Information Objects” on page 229.
Returns
Any value that the callback function returns.

Description
Event handler; invoked when the server encounters an error while processing a message that was targeted at this application instance. The `application.onStatus` event handler is the root for any `Stream.onStatus` or `NetConnection.onStatus` messages that don’t find handlers. Also, there are a few status calls that come only to `application.onStatus`. This event handler can be used for debugging messages that generate errors.

Example
The following example defines a function that sends a trace statement whenever the `application.onStatus` method is invoked. You can also define a function that gives users specific feedback about the type of error that occurred.
```ActionScript
appInstance.onStatus = function(infoObject){
  trace("An application error occurred");
};
```

Application.registerClass()

Availability
Flash Communication Server MX 1.0.

Usage
```ActionScript
application.registerClass(className, constructor)
```

Parameters
- `className` The name of an ActionScript class.
- `constructor` A constructor function used to create an object of a specific class type during object deserialization. The name of the constructor function must be the same as `className`. During object serialization, the name of the constructor function is serialized as the object’s type. To unregister the class, pass the value `null` as the `constructor` parameter. Serialization is the process of turning an object into something you can send to another computer over the network.
Returns
Nothing.

Description
Method; registers a constructor function that is used when deserializing an object of a certain class type. If the constructor for a class is not registered, you cannot call the deserialized object's methods. This method is also used to unregister the constructor for a class. This is an advanced use of the server and is necessary only when sending ActionScript objects between a client and a server.

The client and the server communicate over a network connection. Therefore, if you use typed objects, each side must have the prototype of the same objects they both use. In other words, both the client-side and server-side ActionScript must define and declare the types of data they share so that there is a clear, reciprocal relationship between an object, method, or property on the client and the corresponding element on the server. You can use `application.registerClass` to register the object's class type on the server side so that you can use the methods defined in the class.

Constructor functions should be used to initialize properties and methods; they should not be used for executing server code. Constructor functions are called automatically when messages are received from the client and need to be “safe” in case they are executed by a malicious client. You shouldn't define procedures that could result in negative situations such as filling up the hard disk or consuming the processor.

The constructor function is called before the object's properties are set. A class can define an `onInitialize` method, which is called after the object has been initialized with all its properties. You can use this method to process data after an object is deserialized.

If you register a class that has its prototype set to another class, you must set the prototype constructor back to the original class after setting the prototype. The second example below illustrates this point.

NOTE
Client-side classes must be defined as `function function_name(){}`, as shown in the following examples. If not defined in the correct way, `application.registerClass` will not properly identify the class when its instance passes from the client to the server, and an error will be returned.
Example

The following example defines a `Color` constructor function with properties and methods. After the application connects, the `registerClass` method is called to register a class for the objects of type `Color`. When a typed object is sent from the client to the server, this class is called to create the server-side object. After the application stops, the `registerClass` method is called again and passes the value `null` to unregister the class.

```javascript
function Color(){
  this.red = 255;
  this.green = 0;
  this.blue = 0;
}
Color.prototype.getRed = function(){
  return this.red;
}
Color.prototype.getGreen = function(){
  return this.green;
}
Color.prototype.getBlue = function(){
  return this.blue;
}
Color.prototype.setRed = function(value){
  this.red = value;
}
Color.prototype.setGreen = function(value){
  this.green = value;
}
Color.prototype.setBlue = function(value){
  this.blue = value;
}
application.onAppStart = function(){
  application.registerClass("Color", Color);
};
application.onAppStop = function(){
  application.registerClass("Color", null);
};

The following example shows how to use the `application.registerClass` method with the prototype property:

```javascript
function A(){
}
function B(){

B.prototype = new A();
// Set constructor back to that of B.
B.prototype.constructor = B;
// Insert code here.
application.registerClass("B", B);
```
Application.registerProxy()

Availability
Flash Communication Server MX 1.0.

Usage
application.registerProxy(methodName, proxyConnection [,. proxyMethodName])

Parameters
methodName The name of a method. All requests to execute methodName for this application instance are forwarded to the proxyConnection object.
proxyConnection A Client or NetConnection object. All requests to execute the remote method specified by methodName are sent to the Client or NetConnection object specified in the proxyConnection parameter. Any result returned is sent back to the originator of the call. To unregister or remove the proxy, provide a value of null for this parameter.
proxyMethodName An optional parameter. The server calls this method on the object specified by the proxyConnection parameter if proxyMethodName is different from the method specified by the methodName parameter.

Returns
A value that is sent back to the client that made the call.

Description
Method; maps a method call to another function. You can use this method to communicate between different application instances that can be on the same Flash Media Server (or different Flash Media Servers). Clients can execute server-side methods of any application instances to which they are connected. Server-side scripts can use this method to register methods to be proxied to other application instances on the same server or a different server. You can remove or unregister the proxy by calling this method and passing null for the proxyConnection parameter, which results in the same behavior as never registering the method at all.
Example
In the following example, the `application.registerProxy` method is called in a function in the `application.onAppStart` event handler and executes when the application starts. In the function block, a new `NetConnection` object called `myProxy` is created and connected. The `application.registerProxy` method is then called to assign the method `getXyz` to the `myProxy` object.

```javascript
application.onAppStart = function(){
    var myProxy = new NetConnection();
    myProxy.connect("rtmp://xyz.com/myApp");
    application.registerProxy("getXyz", myProxy);
};
```

**Application.rejectConnection()**

**Availability**
Flash Communication Server MX 1.0.

**Usage**
`application.rejectConnection(clientObj, errObj)`

**Parameters**
- `clientObj`  A client to reject.
- `errObj`     An object of any type that is sent to the client, explaining the reason for rejection.

The `errObj` object is available in client-side scripts as the `application` property of the information object that is passed to the `application.onStatus` call when the connection is rejected. For more information, see the appendix, “Client-Side Information Objects,” in the *Client-Side ActionScript Language Reference for Flash Media Server 2.*

**Returns**
Nothing.

**Description**
Method; rejects the connection call from a client to the server. The `application.onConnect` event handler notifies a script when a new client is connecting. In the function assigned to `application.onConnect`, you can either accept or reject the connection. You can also define a function for `application.onConnect` that calls an application server for authentication. In that case, an application could call `application.rejectConnection` from the application server's response callback to disconnect the client from the server.
When you use components and your code includes an explicit call to `application.acceptConnection` or `application.rejectConnection`, the last line (in order of execution) of the `onConnect` method should be either `application.acceptConnection` or `application.rejectConnection`. Also, any logic that follows the explicit `acceptConnection` or `rejectConnection` statement must be placed in `application.onConnectAccept` and `application.onConnectReject` statements, or it will be ignored. This requirement exists only when you use components.

**Example**

In the following example, the client specified by `client1` is rejected and provided with the error message contained in `err.message`. The message “Too many connections” appears on the server side.

```javascript
function onConnect(client1){
    // Insert code here.
    var err = new Object();
    err.message = "Too many connections";
    application.rejectConnection(client1, err);
}
```

The following code should appear on the client side:

```javascript
clientConn.onStatus = function(info){
    if (info.code == "NetConnection.Connect.Rejected"){
        trace(info.application.message);
        // Sends the message
        // "Too many connections" to the Output panel
        // on the client-side.
    }
};
```

**See also**

`Application.onConnect`, `Application.acceptConnection`
Application.server

Availability
Flash Communication Server MX 1.0.

Usage
application.server

Description
Property (read-only); contains the platform and the server-version information.

Example
The following example checks the server property against a string before executing the code in the if statement:

```javascript
if (application.server == "Flash Media Server-Windows/1.0"){
    // Insert code here.
}
```

Application.shutdown()

Availability
Flash Media Server 2.

Usage
application.shutdown()

Parameters
None.

Returns
A Boolean value indicating success (true), or failure (false).
Description
Method: unloads the application instance.

If the application is running in vhost or application-level scope, only the application instance is unloaded, but the core process remains running. If the application is running in instance scope, the application instance is unloaded and the core process terminates. This process is done asynchronously; the instance is unloaded when the unload sequence begins, not when the `shutdown()` call returns.

After `shutdown()` is called, `Application.onAppStop` is called, connected clients are disconnected, and `Application.onDisconnect` is called for each client. Calls made after calling `shutdown()` may not execute.

Example
In the following example, the client specified by `client1` is rejected and provided with the error message contained in `err.message`. The message “Too many connections” appears on the server side.

```
function onConnect(client1){
  // Insert code here.
  var err = new Object();
  err.message = "Too many connections";
  application.rejectConnection(client1, err);
}
```

The following code should appear on the client side:

```
clientConn.onStatus = function (info){
  if (info.code == "NetConnection.Connect.Rejected"){
    trace(info.application.message);
    // Sends the message
    // "Too many connections" to the Output panel
    // on the client-side.
    }
};
```

See also
`Application.onConnect`, `Application.acceptConnection()`
clearInterval()

**Availability**
Flash Communication Server MX 1.0.

**Usage**
clearInterval(intervalID)

**Parameters**

*intervalID*  A unique ID returned by a previous call to the `setInterval()` method.

**Returns**
Nothing.

**Description**
Method (global); cancels a time-out that was set with a call to the `setInterval()` method.

**Example**
The following example creates a function named `callback` and passes it to the `setInterval()` method, which is called every 1000 milliseconds and outputs the message “interval called.” The `setInterval()` method returns a unique identifier that is assigned to the variable `intervalID`. The identifier allows you to cancel a specific `setInterval()` call. In the last line of code, the `intervalID` variable is passed to the `clearInterval()` method to cancel the `setInterval()` call:

```javascript
function callback(){
  trace("interval called");
}
var intervalID;
intervalID = setInterval(callback, 1000);
// sometime later
clearInterval(intervalID);
```
Client class

Availability
Flash Communication Server 1.

The Client class lets you handle each user, or client, connection to a Flash Media Server application instance. The server automatically creates a Client object when a user connects to an application; the object is destroyed when the user disconnects from the application. Users have unique Client objects for each application to which they are connected. Thousands of Client objects can be active at the same time.

You can use the properties of the Client class to determine the version, platform, and IP address of each client. You can also set individual read and write permissions to various application resources such as Stream objects and shared objects. Use the methods of the Client class to set bandwidth limits and call methods in client-side scripts.

When you call `NetConnection.call()` from a client-side ActionScript script, the method that executes in the server-side script must be a method of the Client class. In your server-side script, you must define any method you want to call from the client-side script. You can also call any methods you define in the server-side script directly from the Client class instance in the server-side script.

If all instances of the Client class (each client in an application) require the same methods or properties, you can add those methods and properties to the class itself instead of adding them to each instance of a class. This process is called extending a class. You can extend any server-side or client-side ActionScript class. To extend a class, instead of defining methods in the constructor function of the class or assigning them to individual instances of the class, you assign methods to the `prototype` property of the constructor function of the class. When you assign methods and properties to the `prototype` property, the methods are automatically available to all instances of the class.

The following code shows how to assign methods and properties to an instance of a class. In the `application.onConnect` handler, the client instance `clientObj` is passed to the server-side script as a parameter. You can then assign a property and method to the client instance:

```javascript
application.onConnect = function(clientObj){
  clientObj.birthday = myBDay;
  clientObj.calculateDaysUntilBirthday = function(){
    // Insert code here.
  }
}
```
The previous example works, but must be executed every time a client connects. If you want
the same methods and properties to be available to all clients in the application.clients
array without defining them every time, you must assign them to the prototype property
of the Client class. There are two steps to extending a built-in class using the prototype
method. You can write the steps in any order in your script. The following example extends
the built-in Client class, so the first step is to write the function that you will assign to the
prototype property:

// First step: write the functions.

function Client_getWritePermission(){
    // The writeAccess property is already built in to the client class.
    return this.writeAccess;
}

function Client_createUniqueID(){
    var ipStr = this.ip;
    // The ip property is already built in to the client class.
    var uniqueID = "re123mn"
    // You would need to write code in the above line
    // that creates a unique ID for each client instance.
    return uniqueID;
}

// Second step: assign prototype methods to the functions.

Client.prototype.getWritePermission = Client_getWritePermission;
Client.prototype.createUniqueID = Client_createUniqueID;

// A good naming convention is to start all class method
// names with the name of the class followed by an underscore.

You can also add properties to prototype, as shown in the following example:

Client.prototype.company = "Macromedia";

The methods are available to any instance, so within application.onConnect, which is
passed a clientObj argument, you can write the following code:

application.onConnect = function(clientObj){
    var clientID = clientObj.createUniqueID();
    var clientWritePerm = clientObj.getWritePermission();
};
### Method summary for the Client class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client.call()</td>
<td>Executes a method on the Flash client asynchronously and returns the value from the Flash client to the server.</td>
</tr>
<tr>
<td>Client.getBandwidthLimit()</td>
<td>Returns the maximum bandwidth the client or the server can attempt to use for this connection.</td>
</tr>
<tr>
<td>Client.getStats()</td>
<td>Returns statistics for the client.</td>
</tr>
<tr>
<td>Client.readAccess</td>
<td>Sends a &quot;ping&quot; message to the client. If the client responds, the method returns true; otherwise it returns false.</td>
</tr>
<tr>
<td>Client.__resolve</td>
<td>Provides values for undefined properties.</td>
</tr>
<tr>
<td>Client.setBandwidthLimit()</td>
<td>Sets the maximum bandwidth for the connection.</td>
</tr>
</tbody>
</table>

### Property summary for the Client class

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client.agent</td>
<td>Read-only; the version and platform of the Flash client.</td>
</tr>
<tr>
<td>Client.ip</td>
<td>Read-only; the IP address of the Flash client.</td>
</tr>
<tr>
<td>Client.protocol</td>
<td>Read-only; the protocol used by the client to connect to the server.</td>
</tr>
<tr>
<td>Client.readAccess</td>
<td>A list of access levels to which the client has read access.</td>
</tr>
<tr>
<td>Clientreferrer</td>
<td>Read-only; the URL of the SWF file or server where this connection originated.</td>
</tr>
<tr>
<td>Client.secure</td>
<td>Read-only; a Boolean value indicating whether an Internet connection is secure (true) or not (false).</td>
</tr>
<tr>
<td>Client.uri</td>
<td>Read-only; the URI specified by the client to connect to this application instance.</td>
</tr>
<tr>
<td>Client.virtualKey</td>
<td>The user agent type of the client (which is typically the Flash Player version), but may be set to any legal key value.</td>
</tr>
<tr>
<td>Client.writeAccess</td>
<td>A list of access levels to which the client has write access.</td>
</tr>
</tbody>
</table>
Event handler summary for the Client class

<table>
<thead>
<tr>
<th>Event handler</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client.&quot;commandName&quot;</td>
<td>Invoked when NetConnection.call(commandName) is called in a client-side script.</td>
</tr>
</tbody>
</table>

**Client.agent**

**Availability**
Flash Communication Server MX 1.0.

**Usage**
clientObject.agent

**Description**
Property (read-only); contains the version and platform information of the Flash client.

**Example**
The following example checks the agent property against the string "WIN" and executes different code depending on whether they match. This code is written in an onConnect function.

```javascript
function onConnect(newClient, name){
  if (newClient.agent.indexOf("WIN") > -1){
    trace ("Window user");
  } else {
    trace ("non Window user.agent is" + newClient.agent);
  }
}
```

**Client.call()**

**Availability**
Flash Communication Server MX 1.0.
Usage
clientObject.call(methodName, [resultObj, [p1, ..., pN]])

Parameters
methodName A method specified in the form [objectPath]/method. For example, the command someObj/doSomething tells the client to invoke the NetConnection.someObj.doSomething method on the client.
resultObj An optional parameter that is required when the sender expects a return value from the client. If parameters are passed but no return value is desired, pass the value null. The result object can be any object you define and, in order to be useful, should have two methods that are invoked when the result arrives: onResult and onStatus. The resultObj.onResult event is triggered if the invocation of the remote method is successful; otherwise, the resultObj.onStatus event is triggered.
p1, ..., pN Optional parameters that can be of any ActionScript type, including a reference to another ActionScript object. These parameters are passed to the methodName parameter when the method executes on the Flash client. If you use these optional parameters, you must pass in some value for resultObject; if you do not want a return value, pass null.

Returns
A Boolean value of true if a call to methodName was successful on the client; otherwise, false.

Description
Method; executes a method on the originating Flash client or on another server. The remote method may optionally return data, which is returned as a result to the resultObj parameter, if it is provided. The remote object is typically a Flash client connected to the server, but it can also be another server. Whether the remote agent is a Flash client or another server, the method is called on the remote agent’s NetConnection object.

Example
The following example shows a client-side script that defines a function called random(), which generates a random number:

```javascript
nc = new NetConnection();
nc.connect("rtmp://someserver/someApp/someInst");
nc.random = function(){
    return (Math.random());
};
```
The following server-side script uses the `Client.call()` method in the `application.onConnect` handler to call the `random()` method that was defined on the client side. The server-side script also defines a function called `randHandler()`, which is used in the `Client.call()` method as the `resultObj` parameter.

```actionscript
application.onConnect = function(clientObj){
    trace("we are connected");
    application.acceptConnection(clientObj);
    clientObj.call("random", new randHandler());
}
randHandler = function(){
    this.onResult = function(res){
        trace("random num: " + res);
    }
    this.onStatus = function(info){
        trace("failed and got:" + info.code);
    }
};
```

**Client."commandName"**

**Availability**
Flash Communication Server MX 1.0.

**Usage**

```actionscript
function clientCommand([p1, ..., pN]){}
```

**Parameters**

`p1, ..., pN` Optional parameters passed to the command message handler if the message contains parameters you defined. These parameters are the ActionScript objects passed to the `NetConnection.call()` method.

**Returns**

Any ActionScript object you define. This object is serialized and sent to the client as a response to the command only if the command message supplied a response handler.

**Description**

Event handler; invoked when a Flash client or another server calls the `NetConnection.call()` method. A command name parameter is passed to `NetConnection.call()` on the client side, which causes Flash Media Server to search the `Client` object instance on the server side for a method that matches the command name parameter. If the parameter is found, the method is invoked and the return value is sent back to the client.
Example

The following example creates a method called `sum` as a property of the Client object on the server side:

```javascript
newClient.sum = function sum(op1, op2){
    return op1 + op2;
};
```

The `sum()` method can then be called from `NetConnection.call()` on the Flash client side, as shown in the following example:

```javascript
nc = new NetConnection();
nc.connect("rtmp://myServer/myApp");
nc.call("sum", new result().20, 50);
function result(){
    this.onResult = function (retVal){
        output += "sum is " + retVal;
    };
    this.onStatus = function(errorVal){
        output += errorVal.code + " error occurred";
    };
}
```

The `sum()` method can also be called on the server side, as shown here:

```javascript
newClient.sum();
```

The following example creates two functions that you can call from either a client-side or server-side script:

```javascript
application.onConnect = function(clientObj) {
    // Add a callable function called "foo": foo returns the number 8.
    clientObj.foo = function() {return 8;};
    // Add a remote function that is not defined in the onConnect call.
    clientObj.bar = application.barFunction;
};
// The bar function adds the two values it is given.
application.barFunction = function(v1,v2) {
    return (v1 + v2);
};
```

You can call either of the two functions that were defined in the previous examples (`foo` and `bar`) by using the following code in a client-side script:

```javascript
c = new NetConnection();
c.call("foo");
c.call("bar", null, 1, 1);
```
You can call either of the two functions that were defined in the previous examples (foo and bar) by using the following code in a server-side script:

c = new NetConnection();
c.onStatus = function(info) {
  if(info.code == "NetConnection.Connect.Success") {
    c.call("foo");
    c.call("bar", null, 2, 2);
  }
};

Client.getBandwidthLimit()

Availability
Flash Communication Server MX 1.0.

Usage
clientObject.getBandwidthLimit(idirection)

Parameters
idirection An integer specifying the connection direction: 0 indicates a client-to-server direction, 1 indicates a server-to-client direction.

Returns
An integer indicating bytes per second.

Description
Method; returns the maximum bandwidth that the client or the server can use for this connection. Use the idirection parameter to get the value for each direction of the connection. The value returned indicates bytes per second and can be changed with Client.setBandwidthLimit(). The default value for a connection is set for each application in the Application.xml file.

Example
The following example uses Client.getBandwidthLimit() with the idirection parameter to set two variables, clientToServer and serverToClient:

application.onConnect = function(newClient){
  var clientToServer= newClient.getBandwidthLimit(0);
  var serverToClient= newClient.getBandwidthLimit(1);
};
Client.getStats()

Availability
Flash Communication Server MX 1.0.

Usage
clientObject.getStats()

Returns
An object with various properties for each statistic returned.

Description
Method; returns statistics for the client including the total number of bytes sent and received, the number of RTMP messages sent and received, the number of dropped RTMP messages, and how long it takes the client takes to respond to a ping message.

Example
The following example uses Client.getStats() to output a new client’s statistics:

```actionscript
application.onConnect(newClient) {
    stats = newClient.getStats();
    trace("Total bytes received: " + stats.bytes_in);
    trace("Total bytes sent: " + stats.bytes_out);
    trace("RTMP messages received: " + stats.msg_in);
    trace("RTMP messages sent: " + stats.msg_out);
    trace("RTMP messages dropped: " + stats.msg_dropped);
    trace("Ping roundtrip time: " + stats.ping_rtt);
}
```

See also
Client.readAccess

Client.ip

Availability
Flash Communication Server MX 1.0.

Usage
clientObject.ip

Description
Property (read-only); a string containing the IP address of the Flash client.
Example
The following example uses the `Client.ip` property to verify whether a new client has a specific IP address. The result determines which block of code runs.

```actionscript
application.onConnect = function(newClient, name){
  if (newClient.ip == "127.0.0.1"){
    // Insert code here.
  } else {
    // Insert code here.
  }
};
```

Client.ping()

Availability
Flash Communication Server MX 1.0.

Usage
`clientObject.ping()`

Description
Method; sends a “ping” message to the client and waits for a response. If the client responds, the method returns `true`; otherwise, `false`. Use this method to determine whether the client connection is still active.

Example
The following `onConnect` function pings the connecting client and traces the results of the method:

```actionscript
application.onConnect(newClient) {
  if (newClient.ping()){
    trace("ping successful");
  } else {
    trace("ping failed");
  }
}
```

See also
`Client.getStats()`
Client.prototype

Availability
Flash Communication Server MX 1.0

Usage
clientObject.protocol

description
Property (read-only); a string indicating the protocol used by the client to connect to the server. This string can have one of the following values:

- rtmp (RTMP over a persistent socket connection)
- rtmpt (RTMP tunneled using the HTTP protocol)
- rtmps (RTMP over an SSL connection)

For more information about the HTTP tunneling feature in Flash Communication Server MX 1.0, see the NetConnection.connect() entry in the Client-Side ActionScript Language Reference for Flash Media Server 2.

Example
The following example checks the connection protocol used by a client upon connection to the application:

```xml
application.onConnect(clientObj) {
  if(clientObj.protocol == "rtmp") {
    trace("Client connected over a persistent connection");
  } else if(clientObj.protocol == "rtmpt") {
    trace("Client connected over an HTTP tunneling connection");
  }
}
```
Client.readAccess

Availability
Flash Communication Server MX 1.0.

Usage
clientObject.readAccess

Description
Property; a string that provides read-access rights to directories that contain application resources (shared objects and streams) for this client. To give a client read access to directories containing application resources, list directories in a string delimited by semicolons. By default, all clients have full read access, and the readAccess property is set to slash (/). To give a client read access, specify a list of access levels (in URI format), delimited by semicolons. Any files or directories within a specified URI are also considered accessible. For example, if myMedia is specified as an access level, then any files or directories in the myMedia directory are also accessible (for example, myMedia/mp3s). Similarly, any files or directories in the myMedia/mp3s directory are also accessible, and so on.

Clients with read access to a directory that contains streams can play streams in the specified access levels. Clients with read access to a directory that contains shared objects can subscribe to shared objects in the specified access levels and receive notification of changes in the shared objects.

- For streams, readAccess controls the streams that the connection can play.
- For shared objects, readAccess controls if the connection can listen to shared object changes.

Although you cannot use this property to control access for a particular file, you can create a separate directory for a file if you want to control access to it.

Example
The following onConnect function gives a client read access to myMedia/mp3s, myData/notes, and any files or directories within them:

```javascript
application.onConnect = function(newClient, name){
    newClient.readAccess = “myMedia/mp3s:myData/notes”;
};
```

See also
Client.writeAccess
Client.referrer

**Availability**
Flash Communication Server MX 1.0.

**Usage**
*clientObject.referrer*

**Description**
Property (read-only); a string whose value is set to the URL of the SWF file or the server in which this connection originated.

**Example**
The following example defines an `onConnect` callback function that sends a trace that indicates the origin of the new client when that client connects to the application:

```javascript
application.onConnect = function(newClient, name){
    trace("New user connected to server from" + newClient.referrer);
};
```

Client.__resolve

**Availability**
Flash Communication Server MX 1.0.

**Usage**
`Client.__resolve = function(propName){}`

**Parameters**
*propName* The name of an undefined property.

**Returns**
The value of the undefined property, which is specified by the `propName` parameter.

**Description**
Event handler; provides values for undefined properties. When an undefined property of a Client object is referenced by server-side ActionScript code, that object is checked for a `__resolve` method. If the object has a `__resolve` method, the `__resolve` method is invoked and passed the name of the undefined property. The return value of the `__resolve` method is the value of the undefined property. In this way, `__resolve` can supply the values for undefined properties and make it appear as if they are defined.
Example
The following example defines a function that is called whenever an undefined property is referenced:

```ActionScript
Client.prototype.__resolve = function (name) {
    return "Hello, world!";
};
function onConnect(newClient){
    // Prints "Hello World".
    trace (newClient.property1);
}
```

Client.secure

**Availability**
Flash Media Server 2.

**Usage**
`clientObject.secure`

**Description**
Property (read-only); a Boolean value that indicates whether an Internet connection is secure (`true`) or not (`false`).

**See also**
NetConnection.connect()

Client.setBandwidthLimit()

**Availability**
Flash Communication Server MX 1.0.

**Usage**
`clientObject.setBandwidthLimit(iServerToClient, iClientToServer)`

**Parameters**
- `iServerToClient` A Number; the bandwidth from server to client, in bytes per second. Use 0 if you don't want to change the current setting.
- `iClientToServer` A Number; the bandwidth from client to server, in bytes per second. Use 0 if you don't want to change the current setting.
Returns
Nothing.

Description
Method; sets the maximum bandwidth for this client from client to server, server to client, or both. The default value for a connection is set for each application in the Application.xml file. The value specified cannot exceed the bandwidth cap value specified in the Application.xml file.

Example
The following example sets the bandwidth limits for each direction, based on values passed to the onConnect function:

```javascript
application.onConnect = function(newClient, serverToClient, clientToServer){
    newClient.setBandwidthLimit(serverToClient, clientToServer);
    application.acceptConnection(newClient);
}
```

Client.uri

Availability
Flash Media Server 2.

Usage
clientObject.uri

Description
Property (read-only); the URI that the client specifies to connect to this application instance.

Example
The following example defines an onConnect callback function that sends a message indicating the URI the new client used to connect to the application:

```javascript
application.onConnect = function(newClient, name){
    trace("New user requested to connect to " + newClient.uri);
};
```
Client.virtualKey

Availability
Flash Media Server 2.

Usage
`clientObject.virtualKey`

Description
Property; the user agent type of the client, which is typically the Flash Player version, but can be set to any legal key value. A legal key can contain a string that does not contain the characters “*” or “:”.

For more information, see `Stream.setVirtualPath()` on page 158.

Client.writeAccess

Availability
Flash Communication Server MX 1.0.

Usage
`clientObject.writeAccess`

Description
Property; provides write-access rights to directories containing application resources (such as shared objects and streams) for this client. To give a client write access to directories containing application resources, list directories in a string delimited by semicolons. By default, all clients have full write access, and the writeAccess property is set to slash (/). For example, if `myMedia` is specified as an access level, then any files or directories in the myMedia directory are also accessible (for example, myMedia/myStreams). Similarly, any files or subdirectories in the myMedia/myStreams directory are also accessible, and so on.

- For shared object, `writeAccess` provides control over who can create and update the shared objects.
- For streams, `writeAccess` provides control over who can publish and record a stream.

- **Tip** Although you cannot use this property to control access for a particular file, you can create a separate directory for a file if you want to control access to it.
- **Tip** Don’t precede the stream path with a leading slash (/) on the client side.
Example
The following example provides write access to the /myMedia/myStreams and myData/notes directories.

```javascript
application.onConnect = function(newClient, name){
    newClient.writeAccess = "/myMedia/myStreams;myData/notes";
    application.acceptConnection();
};
```

See also
Client.readAccess

File class

Availability
Flash Media Server 2.

The File class lets applications write to the server's file system. This is useful for storing information without using a database server, creating log files for debugging, or tracking usage. Also, a directory listing is useful for building a content list of streams or shared objects without using Flash Remoting.

The File object allows access to the server file system. To protect against any misuse, Flash Media Server allows access to files within a sandbox specified for the virtual host where the application instance is running.

| Tip | A sandbox is a security feature that determines how an application can interact with the local file system, the network, or both the local file system and network at the same time. Restricting how a file can interact with the local file system, or the network helps keep your computer and files safe. |

The server administrator can set up the sandbox for all applications within a virtual host and can provide additional control to individual applications, if required. The following rules are enforced by the server:

- File objects cannot be created using file path specifications.
- By default, a script can access files and directories only within the application directory of the hosting application. A virtual directory mapping can be set up to access files outside the application directory.
- A slash (/) is used as the path separator.
- The File object path follows the URI convention. Access is denied if a path contains a backslash (\) or if a dot (.) or two dots (..) is the only string component found between path separators.
- Root objects cannot be renamed or deleted.
  
  For example, if a path using the slash (/) is used to create a File object, the application folder is mapped.

- Applications can be granted access to additional directories by specifying virtual directory mappings for File object paths.

  This is done by specifying a `<FileObject>` tag within the JSEngine section in the Application.xml file, as shown in the following example:

  ```xml
  <JSEngine>
    <FileObject>
      <VirtualDirectory>/videos;C:\myvideos</VirtualDirectory>
      <VirtualDirectory>/fcsapps;C:\Program Files\fcs\applications</VirtualDirectory>
    </FileObject>
  </JSEngine>
  
  This example specifies two additional directory mappings in addition to the default application directory. Any path specified with `/videos/xyz/vaction.flv` maps to `c:/myvideos/xyz/vaction.flv`. Similarly, `/fcsapps/conference` maps to `c:/Program Files/fcs/applications/conference`. Any path that does not match this mapping resolves to the default application folder. For example, if `c:/myapps/filetest` is the application directory, then `/streams/hello.flv` maps to `c:/myapps/filetest/streams/hello.flv`.

  The File object invokes the `Application.onStatus` event handler to report errors when the following File class methods fail:

  - `File.copyTo()`
  - `File.flush()`
  - `File.list()`
  - `File.mkdir()`
  - `File.open()`
  - `File.read()`
  - `getGlobal()`
  - `File.remove()`
  - `File.renameTo()`
  - `File.write()`
## Method summary for the File class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>File.close()</code></td>
<td>Closes the file.</td>
</tr>
<tr>
<td><code>File.copyTo()</code></td>
<td>Copies a file to a different location or to the same location with a different filename.</td>
</tr>
<tr>
<td><code>File.eof()</code></td>
<td>Returns a Boolean value indicating whether the file pointer is at the end of file (<code>true</code>) or not (<code>false</code>).</td>
</tr>
<tr>
<td><code>File.flush()</code></td>
<td>Flushes the output buffers of a file.</td>
</tr>
<tr>
<td><code>File.list()</code></td>
<td>If the file is a directory, returns an array with an element for each file in the directory.</td>
</tr>
<tr>
<td><code>File.mkdir()</code></td>
<td>Attempts to create a directory in the file directory.</td>
</tr>
<tr>
<td><code>File.open()</code></td>
<td>Opens a file so that you can read from it or write to it.</td>
</tr>
<tr>
<td><code>File.read()</code></td>
<td>Reads the specified number of characters from a file and returns a string.</td>
</tr>
<tr>
<td><code>File.readAll()</code></td>
<td>Reads the file and returns an array with an element for each line of the file.</td>
</tr>
<tr>
<td><code>File.readByte()</code></td>
<td>Reads the next byte from the file and returns the numeric value of the next byte, or -1 if the operation fails.</td>
</tr>
<tr>
<td><code>File.readln()</code></td>
<td>Reads the next line from the file and returns it as a string.</td>
</tr>
<tr>
<td><code>File.remove()</code></td>
<td>Removes the file or directory pointed to by the File object.</td>
</tr>
<tr>
<td><code>File.renameTo()</code></td>
<td>Moves or renames a file.</td>
</tr>
<tr>
<td><code>File.write()</code></td>
<td>Writes data to a file.</td>
</tr>
<tr>
<td><code>File.writeAll()</code></td>
<td>Takes an array as a parameter and calls the <code>File.writeln()</code> method on each element in the array.</td>
</tr>
<tr>
<td><code>File.writeByte()</code></td>
<td>Writes a byte to a file.</td>
</tr>
<tr>
<td><code>File.writeln()</code></td>
<td>Writes data to a file and adds a platform-dependent end-of-line character after outputting the last parameter.</td>
</tr>
<tr>
<td><code>getGlobal()</code></td>
<td>Skips a specified number of bytes and returns the new file position.</td>
</tr>
<tr>
<td><code>File.toString()</code></td>
<td>Returns a string that indicates the name of the File object.</td>
</tr>
</tbody>
</table>
## Property summary for the File class

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>File.canAppend</code></td>
<td>Read-only; a Boolean value indicating whether a file was open for appending.</td>
</tr>
<tr>
<td><code>File.canRead</code></td>
<td>Read-only; a Boolean value indicating whether a file can be read.</td>
</tr>
<tr>
<td><code>File.canReplace</code></td>
<td>Read-only; a Boolean value indicating whether a file was opened with the replace flag enabled.</td>
</tr>
<tr>
<td><code>File.canWrite</code></td>
<td>Read-only; a Boolean value indicating whether you can write to a file.</td>
</tr>
<tr>
<td><code>File.creationTime</code></td>
<td>Read-only; a Date object containing the time the file was created.</td>
</tr>
<tr>
<td><code>File.exists</code></td>
<td>Read-only; a Boolean value indicating whether the file or directory exists.</td>
</tr>
<tr>
<td><code>File.isDirectory</code></td>
<td>Read-only; a Boolean value indicating whether the file is a directory.</td>
</tr>
<tr>
<td><code>File.isFile</code></td>
<td>Read-only; a Boolean value indicating whether the file is a regular data file.</td>
</tr>
<tr>
<td><code>File.isOpen</code></td>
<td>Read-only; a Boolean value indicating whether the file has been successfully opened and is still open.</td>
</tr>
<tr>
<td><code>File.lastModified</code></td>
<td>Read-only; a Date object containing the time the file was last modified.</td>
</tr>
<tr>
<td><code>File.length</code></td>
<td>Read-only; for a directory, the number of files in the directory, not counting the current directory and parent directory entries; for a file, the number of bytes in the file.</td>
</tr>
<tr>
<td><code>File.mode</code></td>
<td>Read-only; the mode of an open file.</td>
</tr>
<tr>
<td><code>File.name</code></td>
<td>Read-only; a string indicating the name of the file.</td>
</tr>
<tr>
<td><code>File.position</code></td>
<td>The current offset in the file.</td>
</tr>
<tr>
<td><code>File.type</code></td>
<td>Read-only; a string specifying the type of data or encoding used when a file is opened.</td>
</tr>
</tbody>
</table>
Constructor for the File class

Availability
Flash Media Server 2.

Usage
```
fileObject = new File(name)
```

Parameters

- **name**: Specifies the name of the file or directory. The name can contain only UTF-8 encoded characters; high byte values can be encoded using the URI character-encoding scheme. The specified name is mapped to a system path using the mappings specified in the Application.xml file. If the path is invalid, the `name` property of the object is set to an empty string, and no file operation can be performed.

Returns
A File object if successful; otherwise, `null`.

Description
Constructor; to create an instance of the File class, use the standard syntax for object creation.

Example
The following code creates an instance of the File class:
```
var errorLog = new File("/logs/error.txt");
```

File.canAppend

Availability
Flash Media Server 2

Usage
```
fileObject.canAppend
```

Description
Property (read-only); a Boolean value indicating whether a file was open for appending (`true`) or not (`false`). The property is undefined for closed files.
File.canRead

Availability
Flash Media Server 2.

Usage
fileObject.canRead

Description
Property (read-only); a Boolean value indicating whether a file can be read (true) or not (false).

File.canReplace

Availability
Flash Media Server 2.

Usage
fileObject.canReplace

Description
Property (read-only); a Boolean value indicating whether a file was opened in create mode (true) or not (false). This property is undefined for closed files.

See also
File.open()

File.canWrite

Availability
Flash Media Server 2.

Usage
fileObject.canWrite
Description
Property (read-only); a Boolean value indicating whether you can write to a file (true) or not (false).

NOTE
If File.open() was called to open the file, the mode in which the file was opened is respected. For example, if the file was opened in read mode, you can read from the file, but you cannot write to the file.

File.close()

Availability
Flash Media Server 2.

Usage
fileObject.close()

Parameters
None.

Returns
A Boolean value indicating whether the file was closed successfully (true) or not (false).

Description
Method; closes the file. Returns false if the file is not open. This method is called automatically on an open File object when the object is out of scope.

Example
The following code closes the /path/file.txt file:

```actionscript
if (x.open("/path/file.txt", "read") ){
    // Do something here.
    x.close();
}
```

Example
File.copyTo()

Availability
Flash Media Server 2.

Usage
fileObject.copyTo(name)

Parameters
name  Specifies the name of the destination file. The name can contain only UTF-8 characters; high byte values can be encoded using the URI character-encoding scheme. The name specified is mapped to a system path using the mappings specified in the Application.xml file. If the path is invalid or the destination file doesn’t exist, the operation fails, and the method returns false.

Returns
A Boolean value indicating whether the file as copied successfully (true) or not (false).

Description
Method; copies a file to a different location or copies it to the same location with a different file name. The method returns false if the source file doesn’t exist or if the source file is a directory.

Example
The following code copies the file set by myFileObj file object to the location provided by the parameter:
if (myFileObj.copyTo( "/logs/backup/hello.log")) {
    // Do something here.
}

File.creationTime

Availability
Flash Media Server 2.

Usage
fileObject.creationTime

Description
Property (read-only); a Date object containing the time the file was created.
File.eof()

Availability
Flash Media Server 2.

Usage
fileObject.eof()

Parameters
None.

Returns
A Boolean value.

Description
Method; returns a Boolean value indicating whether the file pointer is pointing to the end of
file (true) or not (false). If the file is closed, the method returns true.

Example
The following while statement lets you insert code that executes until the file pointer is at the
end of a file:
while (!myFileObj.eof()){
    // Do something here.
}

File.exists

Availability
Flash Media Server 2.

Usage
fileObject.exists

Description
Property (read-only); a Boolean value indicating whether the file or directory exists (true) or
not (false).
File.flush()

Availability
Flash Media Server 2.

Usage

```actionscript
fileObject.flush()
```

Parameters

Returns

A Boolean value; returns a Boolean value indicating whether the flush operation was successful (true) or not (false).

Description

Method; flushes the output buffers of a file. The operation fails if the file is closed.

Example

The following if statement lets you insert code that executes if `myFileObj` flushes successfully:

```actionscript
if (myFileObj.flush()){
    // Do something here.
}
```

File.isDirectory

Availability
Flash Media Server 2.

Usage

```actionscript
fileObject.isDirectory
```

Description

Property (read-only); a Boolean value indicating whether the file is a directory (true) or not (false).

A File object that represents a directory has properties that represent the files contained in the directory. These properties have the same names as the files in the directory, as shown in the following example:

```actionscript
myDir = new File("/some/directory");
myFileInDir = myDir.foo;
```
The following example uses named property lookup to refer to files that do not have valid property names:

```javascript
mySameFileInDir = myDir["foo"];  
myOtherFile = myDir["some long filename with spaces and such"];  
```

**File.isFile**

**Availability**
Flash Media Server 2.

**Usage**

```javascript
fileObject.isFile  
```

**Description**

Property (read-only); a Boolean value indicating whether a file is a data file (true) or not (false).

**File.isOpen**

**Availability**
Flash Media Server 2.

**Usage**

```javascript
fileObject.isOpen  
```

**Description**

Property (read-only); a Boolean value indicating whether the file has been successfully opened and is still open (true) or not (false).

**NOTE**

Directories do not need to be opened.

**File.lastModified**

**Availability**
Flash Media Server 2.

**Usage**

```javascript
fileObject.lastModified  
```
Description
Property (read-only); a Date object containing the time the file was last modified.

File.length

Availability
Flash Media Server 2.

Usage
fileObject.length

Description
Property (read-only). For a directory, the number of files in the directory, not counting the
current directory and parent directory entries; for a file, the number of bytes in the file.

File.list()

Availability
Flash Media Server 2.

Usage
fileObject.list(filter)

Parameters
filter  A Function object that determines the files in the returned array.
If the function returns true when a file's name is passed to it as a parameter, the file is added
to the array returned by File.list(). This parameter is optional and allows you to filter the
results of the call.

Returns
An Array object containing all the file objects in the directory.

Description
Method; if the file is a directory, returns an array with an element for each file in the directory.

Example
The following example returns files in the current directory that have three-character names:
var a = x.currentDir.list(function(name){return name.length==3;});
File.mkdir()

**Availability**
Flash Media Server 2.

**Usage**
`fileObject.mkdir(newDir)`

**Parameters**
- `newDir` A string indicating the name of the new directory. This name is relative to the current File object instance.

**Returns**
A Boolean value indicating success (`true`) or failure (`false`).

**Description**
Method; creates a directory in the file directory.

**Example**
The following example creates a logs directory in the `myFileObject` instance:
```javascript
if (myFileObject.mkdir("logs")){
    // Do something if a logs directory is created successfully.
}
```

File.mode

**Availability**
Flash Media Server 2.

**Usage**
`fileObject.mode`

**Description**
Property (read-only); the mode of an open file. It can be different from the `mode` parameter that was passed to open the file if you have repeating attributes (for example, `"read, read"`) or if some attributes were ignored. The property is `undefined` if the file is closed.

**See also**
`File.open()`
File.name

Availability
Flash Media Server 2.

Usage
fileObject.name

Description
Property (read-only); a string indicating the name of the file. The value is an empty string if the File object was created with a invalid path.

File.open()

Availability
Flash Media Server 2.

Usage
fileObject.open("type","mode")

Parameters
- **type** A string indicating the encoding type for the file. The following types are supported:
  - "text" Opens the file for text access using the default file encoding.
  - "binary" Opens the file for binary access.
  - "utf8" Opens the file for UTF-8 access.
- **mode** A string indicating the mode in which to open the file. The following modes are valid and can be combined (modes are case-sensitive and multiple modes must be separated by commas):
  - "read" Indicated that the file is opened for reading.
  - "write" Indicates that the file is opened for writing.
  - "readWrite" Indicates that the file is opened for both reading and writing.
  - "append" Opens file for write. Positions the file pointer at the end of the file when you attempt to write to the file.
  - "create" Creates a new file if the file is not present. If file exists, its contents are destroyed.

If both "read" and "write" are set, then "readWrite" is automatically set.
Returns
A Boolean value indicating whether the file opened successfully (true) or not (false).

Description
Method; opens a file so that you can read from it or write to it. First, you must create a file object and then call the open() method on that object. There are no default values for the type and mode parameters—values must be specified.

Example
The following if statement lets you insert code that executes when a text file is opened in read mode:

```javascript
if (myFileObject.open("text", "read") ){
    // Do something here.
}
```

File.position

Availability
Flash Media Server 2.

Usage
`fileObject.position`

Description
Property; the current offset in the file. This is the only property that can be set. Setting this property performs a seek operation on the file. The property is undefined for closed files.

File.read()

Availability
Flash Media Server 2.

Usage
`fileObject.read(numChars)`

Parameters
`numChars` An integer specifying the number of characters to read. If `numChars` specifies more bytes than are left in the file, the method reads to the end of the file.
Returns
A string.

Description
Method; reads the specified number of characters from a file and returns a string. The operation will fail if the file is opened in binary mode.

Example
The following code opens a text file in read mode and sets variables for the first 100 characters, a line, and a byte:

```javascript
if (myFileObject.open( "text", "read") ){
    strVal = myFileObject.read(100);
    strLine = myFileObject.readLine();
    strChar = myFileObject.readByte();
}
```

See also
File.readAll(), File.readByte(), File.readLine()

File.readAll()

Availability
Flash Media Server 2.

Usage
`fileObject.readAll()`

Parameters
None.

Returns
An array.

Description
Method; reads the file after the location of the file pointer and returns an array with an element for each line of the file. The operation fails if the file opened in binary mode.

See also
File.read(), File.readByte(), File.readLine(), File.open()
**File.readByte()**

**Availability**
Flash Media Server 2.

**Usage**
`fileObject.readByte()`

**Parameters**

**Returns**
A number; either a positive integer or -1.

**Description**
Method; reads the next byte from the file and returns the numeric value of the next byte or -1 if the operation fails. The operation fails if file is not opened in binary mode.

**See also**
`File.read()`, `File.readAll()`, `File.readLine()`

**File.readLine()**

**Availability**
Flash Media Server 2.

**Usage**
`fileObject.readLine()`

**Parameters**

**Returns**
A string.

**Description**
Method; reads the next line from the file and returns it as a string. The line-separator characters (either \r\n on Windows or \n on Linux) are not included in the string. The character \r is skipped; \n determines the actual end of the line. The operation fails if the file opened in binary mode.
See also
File.read(), File.readAll(), File.readByte()

File.remove()

Availability
Flash Media Server 2.

Usage
fileObject.remove()

Parameters
None.

Returns
A Boolean value indicating whether the file or directory was removed successfully (true) or not (false).

Description
Method; removes the file or directory specified to by the File object. The method returns false if the file is open, the path points to a root folder, or the directory is not empty.

Example
The following if statement lets you execute code when myFileObject is removed:
if (myFileObject.remove()){
   // Do something here.
}

File.renameTo()

Availability
Flash Media Server 2.

Usage
fileObject.renameTo(name)
Parameters

name  The new name for the file or directory. The name can contain only UTF-8 encoded characters; high byte values can be encoded using the URI character-encoding scheme. The specified name is mapped to a system path using the mappings specified in the Application.xml file. If the path is invalid or the destination file doesn't exist, the operation will fail.

Returns

A Boolean value indicating whether the file was renamed or moved successfully (true) or not (false).

Description

Method; moves or renames a file. The operation fails if the file is open or the directory points to the root directory.

Example

The following code lets you execute code when myFileObject is renamed:

```java
if (myFileObject.renameTo(="/logs/hello.log.old")){
    // Do something here.
}
```

File.seek()

Availability

Flash Media Server 2.

Usage

```java
fileObject.seek(numBytes)
```

Parameters

numBytes  An integer indicating the number of bytes to move the file pointer from the current position.

Returns

If the operation is successful, returns the current position in file; otherwise, returns -1. If the file is closed, the operation fails and reports a warning. The operation returns undefined if it's called on a directory.
Description
Method; skips a specified number of bytes and returns the new file position. This method can accept both positive and negative parameters.

Example
In the following example, if the seek call is successful, the code in the if statement runs:
```
if (fileObj.seek(10) != -1){
    // Do something here.
}
```

File.toString()

Availability
Flash Media Server 2.

Usage
```
fileObject.toString()
```

Parameters
None.

Returns
A string.

Description
Method; returns the path to the File object.

Example
The following example outputs the path of the File object myFileObject:
```
trace(myFileObject.toString());
```

File.type

Availability
Flash Media Server 2.

Usage
```
fileObject.type
```
Description
Property (read-only); a string specifying the type of data or encoding used when a file is opened. The following strings are supported: "text", "utf8", and "binary". The property is undefined for directories and closed files. If the file is opened in "text" mode and UTF-8 BOM (Byte Order Mark) is detected, the type property is set to "utf8".

File.write()

Availability
Flash Media Server 2.

Usage
fileObject.write(param0, param1,...paramN)

Parameters
param0, param1,...paramN Parameters to write to the file.

Returns
A Boolean value indicating whether the write was successful (true) or not (false).

Description
Method; writes data to a file. The write() method converts each parameter to a string, and then writes it to the file without separators.

The file contents are buffered internally. The File.flush() method writes the buffer to the file on disk.

Example
The following example writes "Hello world" at the end of the myFileObject text file:
if (myFileObject.open( "text", "append") ) {
  myFileObject.write("Hello world");
}

See also
File.writeAll(), File.WriteByte(), File.writeln()
File.writeAll()

**Availability**
Flash Media Server 2.

**Usage**
`fileObject.writeAll(array)`

**Parameters**
- `array` An Array object containing all the elements to write to the file.

**Returns**
A Boolean value indicating whether the write was successful (`true`) or not (`false`).

**Description**
Method; takes an array as a parameter and calls the `File.writeln()` method on each element in the array. The file contents are buffered internally. The `File.flush()` method writes the buffer to the file on disk.

**See also**
`File.write()`, `File.writeByte()`, `File.writeln()`

File.writeByte()

**Availability**
Flash Media Server 2.

**Usage**
`fileObject.writeByte(number)`

**Parameters**
- `number` The numeric value to write.

**Returns**
A Boolean value indicating whether the write was successful (`true`) or not (`false`).

**Description**
Method; writes a byte to a file.
The file contents are buffered internally. The `File.flush()` method writes the buffer to the file on disk.

**Example**
The following example writes byte 65 to the end of the `myFileObject` file:

```java
if (myFileObject.open("text","append")) {
    myFileObject.WriteByte(65);
}
```

**See also**
`File.write()`, `File.WriteAll()`, `File.writeln()`

**File.writeln()**

**Availability**
Flash Media Server 2.

**Usage**
`fileObject.writeln(param0, param1,...,paramN)`

**Parameters**
`param0, param1,...,paramN` Strings to write to the file.

**Returns**
A Boolean value indicating whether the write was successful (`true`) or not (`false`).

**Description**
Method; writes data to a file and adds a platform-dependent end-of-line character after the last parameter.

The file contents are buffered internally. The `File.flush()` method writes the buffer to the file on disk.

**Example**
The following example opens a text file for writing and writes a line:

```java
if (fileObj.open( "text", "append" ) ) {
    fileObj.writeln("This is a line!");
}
```

**See also**
`File.write()`, `File.WriteAll()`, `File.WriteByte()`
**getGlobal()**

**Availability**
Flash Media Server 2.

**Usage**
getGlobal()

**Parameters**
None.

**Returns**
Nothing.

**Description**
Method (global); provides access to the global object while the secure.asc file is loading. Use the global object to manipulate built-in global functions. To prevent inadvertent access to the global object, always hold its reference in a temporary variable (declared by `var`); do not hold its reference in a member variable or global variable. This function can only be used in the secure.asc file.

Flash Media Server 2 has two script execution modes: secure and normal. In secure mode, only the secure.asc file (if it exists) is loaded and evaluated—no other application scripts are loaded. Also, the global `getGlobal()` and `protectObject()` functions are only available in secure mode. These are very powerful because they provide complete access to the script execution environment and let you create system objects. Once the secure.asc file is loaded, the server switches to normal script execution mode until the application is unloaded.

Use the `getGlobal()` function to create protected system calls. For more information, see “Implementing secure system objects” in Developing Media Applications.

**Example**
The following code gets a reference to the global object:

```javascript
var global = getGlobal();
```

**See also**
`protectObject()`
load()

Availability
Flash Communication Server MX 1.0.

Usage
load(filename)

Parameters
filename  The relative path to an ActionScript file from the main.asc file.

Returns
Nothing.

Description
Method (global); loads an ActionScript file in the main.asc file. This method executes only when the ActionScript file is loaded first. The loaded file is compiled and executed after the main.asc file is successfully loaded, compiled, and executed, but before application.onAppStart() is executed. The path of the specified file is resolved, relative to main.asc. This method is useful for loading ActionScript libraries.

Example
The following example loads the myLoadedFile.as file:
load("myLoadedFile.as");

LoadVars class

Availability
Flash Media Server 2.

The LoadVars class lets you send all the variables in an object to a specified URL and to load all the variables at a specified URL into an object. It also lets you send specific variables, rather than all variables, which can make your application more efficient. You can use the LoadVars.onLoad handler to ensure that your application runs when data is loaded, and not before.

NOTE For security reasons, your server-side applications directory, which contains ASC files, audio/video FLV files, and ActionScript FLA source files, should not be accessible to users who are browsing your website.
The LoadVars class works much like the XML class; it uses the `load()`, `send()`, and `sendAndLoad()` methods to communicate with a server. The main difference between the LoadVars class and the XML class is that LoadVars transfers ActionScript name and value pairs, rather than an XML Document Object Model (DOM) tree stored in the XML object. The LoadVars class follows the same security restrictions as the XML class.

### Method summary for the LoadVars class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoadVars.addRequestHeader()</td>
<td>Adds or changes HTTP request headers (such as Content-Type or SOAPAction) sent with POST actions.</td>
</tr>
<tr>
<td>LoadVars.decode()</td>
<td>Converts the query string to properties of the specified LoadVars object</td>
</tr>
<tr>
<td>LoadVars.getBytesLoaded()</td>
<td>Returns the number of bytes loaded from the last or current <code>LoadVars.load()</code> or <code>LoadVars.sendAndLoad()</code> method call.</td>
</tr>
<tr>
<td>LoadVars.getBytesTotal()</td>
<td>Returns the number of total bytes loaded during all <code>LoadVars.load()</code> or <code>LoadVars.sendAndLoad()</code> method calls.</td>
</tr>
<tr>
<td>LoadVars.load()</td>
<td>Downloads variables from the specified URL, parses the variable data, and places the resulting variables in a LoadVars object.</td>
</tr>
<tr>
<td>LoadVars.send()</td>
<td>Sends the variables in the <code>myLoadVars</code> object to the specified URL.</td>
</tr>
<tr>
<td>LoadVars.sendAndLoad()</td>
<td>Posts variables in the <code>myLoadVars</code> object to the specified URL.</td>
</tr>
<tr>
<td>LoadVars.toString()</td>
<td>Returns a string containing all enumerable variables in <code>myLoadVars</code>, in the MIME content encoding <code>application/x-www-urlform-encoded</code>.</td>
</tr>
</tbody>
</table>
Property summary for the LoadVars class

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoadVars.contentType</td>
<td>The MIME type sent to the server when you call the LoadVars.send() or LoadVars.sendAndLoad() method.</td>
</tr>
<tr>
<td>LoadVars.loaded</td>
<td>A Boolean value that indicates whether a LoadVars.load() or LoadVars.sendAndLoad() operation has completed (true) or not (false).</td>
</tr>
</tbody>
</table>

Event summary for the LoadVars class

<table>
<thead>
<tr>
<th>Event handler</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoadVars.onData</td>
<td>Invoked when data has completely downloaded from the server or when an error occurs while data is downloading from a server.</td>
</tr>
<tr>
<td>LoadVars.onHTTPStatus</td>
<td>Invoked when Flash Media Server receives an HTTP status code from the server.</td>
</tr>
<tr>
<td>LoadVars.onLoad</td>
<td>Invoked when a LoadVars.load() or LoadVars.sendAndLoad() operation has completed.</td>
</tr>
</tbody>
</table>

Constructor for the LoadVars class

Availability
Flash Media Server 2.

Usage
new LoadVars()

Parameters
None.

Description
Constructor; creates a LoadVars object. You can use the methods of the LoadVars object to send and load data.

Example
The following example creates a LoadVars object called my_lv:

```javascript
var my_lv = new LoadVars();
```
LoadVars.addRequestHeader()

**Availability**
Flash Media Server 2.

**Usage**
myLoadVars.addRequestHeader(header, headerValue)

**Parameters**
- **header**  A string or array of strings that represents an HTTP request header name.
- **headerValue**  A string that represents the value associated with header.

**Returns**
Nothing.

**Description**
Method; adds or changes HTTP request headers (such as Content-Type or SOAPAction) sent with POST actions. There are two possible use cases for this method: you can pass two strings, header and headerValue, or you can pass an array of strings, alternating header names and header values.

If multiple calls are made to set the same header name, each successive value will replace the value set in the previous call.

The following standard HTTP headers cannot be added or changed with this method: Accept-Ranges, Age, Allow, Allowed, Connection, Content-Length, Content-Location, Content-Range, ETag, Host, Last-Modified, Locations, Max-Forwards, Proxy-Authenticate, Proxy-Authorization, Public, Range, Retry-After, Server, TE, Trailer, Transfer-Encoding, Upgrade, URI, Vary, Via, Warning, and WWW-Authenticate.

**Example**
The following example adds a custom HTTP header named SOAPAction with a value of Foo to the my_lv object:

```actionscript
my_lv.addRequestHeader("SOAPAction", "'Foo'");
```

The following example creates an array named headers that contains two alternating HTTP headers and their associated values. The array is passed as an argument to the addRequestHeader() method.

```actionscript
var headers = ["Content-Type", "text/plain", "X-ClientAppVersion", "2.0"]; my_lv.addRequestHeader(headers);
```
The following example creates a new LoadVars object that adds a request header called
FLASH-UUID. The header contains a variable that the server can check.

```javascript
var my_lv = new LoadVars();
my_lv.addRequestHeader("FLASH-UUID", "41472");
my_lv.name = "Mort";
my_lv.age = 26;
my_lv.send("http://flash-mx.com/mm/cgivars.cfm", "_blank", "POST");
```

**LoadVars.contentType**

**Availability**
Flash Media Server 2.

**Usage**

```javascript
myLoadVars.contentType
```

**Description**

Property; the MIME type that is sent to the server when you call the `LoadVars.send()` or `LoadVars.sendAndLoad()` method. The default is `application/x-www-urlform-encoded`.

**Example**

The following example creates a LoadVars object and displays the default content type of the
data that is sent to the server:

```javascript
var my_lv = new LoadVars();
trace(my_lv.contentType);
// output: application/x-www-form-urlencoded
```

**LoadVars.decode()**

**Availability**
Flash Media Server 2.

**Usage**

```javascript
myLoadVars.decode(queryString)
```

**Parameters**

- `queryString` A URL-encoded query string containing name/value pairs.

**Returns**

Nothing.
Description
Method; converts the query string to properties of the specified LoadVars object.
This method is used internally by the LoadVars.onData event handler. Most users do not need to call this method directly. If you override the LoadVars.onData event handler, you can explicitly call LoadVars.decode() to parse a string of variables.

Example
The following example traces the three variables:

```actionscript
// Create a new LoadVars object
var my_lv = new LoadVars();
// Convert the variable string to properties
my_lv.decode("name=Mort&score=250000");
trace(my_lv.toString());
// Iterate over properties in my_lv
for (var prop in my_lv) {
    trace(prop + " -> " + my_lv[prop]);
}
```

LoadVars.getBytesLoaded()

Availability
Flash Media Server 2.

Usage
```actionscript
myLoadVars.getBytesLoaded()
```

Parameters
None.

Returns
A Number object.

Description
Method; returns the number of bytes loaded from the last or current LoadVars.load() or LoadVars.sendAndLoad() method call.
The value of the contentType property does not affect the value of getBytesLoaded().

See also
LoadVars.getBytesTotal()
LoadVars.getBytesTotal()

Availability
Flash Media Server 2.

Usage
myLoadVars.getBytesTotal()

Parameters
None.

Returns
A Number object.

The `getBytesTotal()` method returns undefined if no load operation is in progress or if a load operation has not been initiated. The `getBytesTotal()` method also returns undefined if the number of total bytes can't be determined—for example, if the download was initiated but the server did not transmit an HTTP content-length.

Description
Method; returns the total number of bytes loaded into an object during all `LoadVars.load()` or `LoadVars.sendAndLoad()` method calls. Each time a call to `load()` or `sendAndLoad()` is issued, the `getBytesLoaded()` method resets, but the `getBytesTotal()` method continues to grow.

The value of the `contentType` property does not affect the value of `getBytesLoaded()`.

See also
LoadVars.getBytesLoaded()

LoadVars.load()

Availability
Flash Media Server 2.

Usage
myLoadVars.load(url)

Parameters

url A string indicating the URL from which to download variables.
Returns
A Boolean value indicating success (true) or failure (false).

Description
Method; downloads variables from the specified URL, parses the variable data, and places the resulting variables into a LoadVars object (myLoadVars). You can load variables from a remote URL or from a URL in the local file system; the same encoding standards apply to both.

Any properties in the myLoadVars object that have the same names as downloaded variables are overwritten. The downloaded data must be in the MIME content type and be application/x-www-urlform-encoded.

The LoadVars.load() method call is asynchronous.

Example
The following code defines an onLoad handler function that signals when data is returned:

```javascript
var my_lv = new LoadVars();
my_lv.onLoad = function(success) {
  if (success) {
    trace(this.toString());
  } else {
    trace("Error loading/parsing LoadVars.");
  }
};
my_lv.load("http://www.helpexamples.com/flash/params.txt");
```

LoadVars.loaded

Availability
Flash Media Server 2.

Usage
myLoadVars.loaded

Description
Property; a Boolean value that indicates whether a LoadVars.load() or LoadVars.sendAndLoad() operation has completed (true) or not (false).
Example
The following example loads a text file and writes information to the log file when the operation completes:

```javascript
var my_lv = new LoadVars();
my_lv.onLoad = function(success) {
    trace("LoadVars loaded successfully: "+this.loaded);
};
my_lv.load("http://www.helpexamples.com/flash/params.txt");
```

See also
LoadVars.onLoad

LoadVars.onData

Availability
Flash Media Server 2.

Usage

```javascript
myLoadVars.onData(src){}
```

Parameters

- `src` A string or undefined; the raw (unparsed) data from a LoadVars.load() or LoadVars.sendAndLoad() method call.

Description

Event handler; invoked when data has completely downloaded from the server or when an error occurs while data is downloading from a server. This handler is invoked before the data is parsed and can be used to call a custom parsing routine instead of the one built in to Flash Player. The value of the `src` parameter passed to the function assigned to LoadVars.onData can be either undefined or a string that contains the URL-encoded name-value pairs downloaded from the server. If the `src` parameter is undefined, an error occurred while downloading the data from the server.

The default implementation of LoadVars.onData invokes LoadVars.onLoad. You can override this default implementation by assigning a custom function to LoadVars.onData, but LoadVars.onLoad is not called unless you call it in your implementation of LoadVars.onData.
Example
The following example loads a text file and displays content in a TextArea instance called content_ta when the operation completes. If an error occurs, then information displays in the Output panel. If an error occurs, then information writes to the log file.

```actionscript
var my_lv = new LoadVars();
my_lv.onData = function(src) {
  if (src == undefined) {
    trace("Error loading content.");
    return;
  }
  content_ta.text = src;
};
my_lv.load("content.txt", my_lv, "GET");
```

LoadVars.onHTTPStatus

Availability
Flash Media Server 2.

Usage
```actionscript
myLoadVars.onHTTPStatus(httpStatus){}
```

Parameters

- `httpStatus`: A number; the HTTP status code returned by the server. For example, a value of 404 indicates that the server has not found a match for the requested URI. HTTP status codes can be found in sections 10.4 and 10.5 of the HTTP specification.

Description
Event handler; invoked when Flash Media Server receives an HTTP status code from the server. This handler lets you capture and act on HTTP status codes.

The onHTTPStatus handler is invoked before onData, which triggers calls to onLoad with a value of undefined if the load fails. After onHTTPStatus is triggered, onData is always triggered, whether or not you override onHTTPStatus. To best use the onHTTPStatus handler, you should write a function to catch the result of the onHTTPStatus call; you can then use the result in your onData and onLoad handlers. If onHTTPStatus is not invoked, this indicates that FMS did not try to make the URL request.

If Flash Media Server cannot get a status code from the server, or if it cannot communicate with the server, the default value of 0 is passed to your ActionScript code.
Example

The following example shows how to use `onHTTPStatus` to help with debugging. The example collects HTTP status codes and assigns their value and type to an instance of the `LoadVars` object. (Notice that this example creates the instance members `this.httpStatus` and `this.httpStatusType` at runtime.) The `onData` handler uses these instance members to trace information about the HTTP response that can be useful in debugging.

```javascript
var myLoadVars = new LoadVars();

myLoadVars.onHTTPStatus = function(httpStatus) {
    this.httpStatus = httpStatus;
    if(httpStatus < 100) {
        this.httpStatusType = "flashError";
    } else if(httpStatus < 200) {
        this.httpStatusType = "informational";
    } else if(httpStatus < 300) {
        this.httpStatusType = "successful";
    } else if(httpStatus < 400) {
        this.httpStatusType = "redirection";
    } else if(httpStatus < 500) {
        this.httpStatusType = "clientError";
    } else if(httpStatus < 600) {
        this.httpStatusType = "serverError";
    }
}

myLoadVars.onData = function(src) {
    trace(">> " + this.httpStatusType + ": " + this.httpStatus);
    if(src != undefined) {
        this.decode(src);
        this.loaded = true;
        this.onLoad(true);
    } else {
        this.onLoad(false);
    }
}

myLoadVars.onLoad = function(success) {
}

myLoadVars.load("http://weblogs.macromedia.com/mxna/flashservices/getMostRecentPosts.cfm");
```
LoadVars.onLoad

Availability
Flash Media Server 2.

Usage
myLoadVars.onLoad(success){}

Parameters
success  A Boolean value indicating whether the LoadVars.load() operation ended in success (true) or failure (false).

Description
Event handler; invoked when a LoadVars.load() or LoadVars.sendAndLoad() operation has completed. If the variables load successfully, the success parameter is true. If the variables were not received, or if an error occurred in receiving the response from the server, the success parameter is false.

If the success parameter is true, the loadVarsObject is populated with variables downloaded by the LoadVars.load() or LoadVars.sendAndLoad() operation, and these variables are available when the onLoad handler is invoked.

This method is undefined by default, but you can define it by assigning it a callback function.

Example
The following example creates a new LoadVars object, attempts to load variables into it from a remote URL, and prints the result:

myLoadVars = new LoadVars();
myLoadVars.onLoad = function(result){
    trace("myLoadVars load success is " + result);
}
myLoadVars.load("http://www.someurl.com/somedata.txt");

LoadVars.send()

Availability
Flash Media Server 2.

Usage
myLoadVars.send(url [, target, method])
Parameters

url A string; the URL to which to upload variables.

target A File object. If you use this optional parameter, any returned data is output to the specified File object. If this parameter is omitted, the response is discarded.

method A string indicating the GET or POST method of the HTTP protocol. The default value is POST. This parameter is optional.

Returns

A Boolean value indicating success (true) or failure (false).

Description

Method; sends the variables in the myLoadVars object to the specified URL.

All enumerable variables in the myLoadVars object are concatenated into a string that is posted to the URL using the HTTP POST method.

The MIME content type sent in the HTTP request headers is the value of LoadVars.contentType.

See also

LoadVars.sendAndLoad()

LoadVars.sendAndLoad()

Availability

Flash Media Server 2.

Usage

myLoadVars.sendAndLoad(url, target[, method ])

Parameters

url A string; the URL to which to upload variables.

target The LoadVars object that receives the downloaded variables.

method A string; the GET or POST method of the HTTP protocol. The default value is POST. This parameter is optional.

Returns

A Boolean value indicating success (true) or failure (false).
Description
Method: posts the variables in the `myLoadVars` object to the specified URL. The server response is downloaded, parsed as variable data, and the resulting variables are placed in the `target` object. Variables are posted in the same way as `LoadVars.send()`. Variables are downloaded into `target` in the same way as `LoadVars.load()`.

See also
`LoadVars.send()`

LoadVars.toString()

Availability
Flash Media Server 2.

Usage
`myLoadVars.toString()`

Parameters
None.

Returns
A string.

Description
Method: returns a string containing all enumerable variables in `myLoadVars`, in the MIME content encoding `application/x-www-form-urlencoded`.

Example
The following example instantiates a new `LoadVars()` object, creates two properties, and uses `toString()` to return a string containing both properties in URL-encoded format:

```actionscript
var my_lv = new LoadVars();
my_lv.name = "Gary";
my_lv.age = 26;
trace (my_lv.toString());
//output: age=26&name=Gary
```
Log class

Availability
Flash Media Server 2.
The Log class lets you create a Log object that can be passed as an optional argument to the constructor for the WebService class. For more information, see “Constructor for the WebService class” on page 164.

Event handler summary for the Log class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log.onLog</td>
<td>Invoked when a log message is sent to a log.</td>
</tr>
</tbody>
</table>

Constructor for the Log class

Availability
Flash Media Server 2.

Usage
new Log([LogLevel[, logName]])

Parameters

- **LogLevel** Level must be set to one of the following (if not set explicitly, the level defaults to Log.BRIEF):
  - Log.BRIEF Primary life cycle event and error notifications are received.
  - Log.VERBOSE All life cycle event and error notifications are received.
  - Log.DEBUG Metrics and fine-grained events and errors are received.

- **logName** An optional parameter that can be used to distinguish between multiple logs that are running simultaneously to the same output.

Returns
A Log object.

Description
Constructor; creates a Log object that can be passed as an optional parameter to the constructor for the WebService class.
Example
The following example creates a new instance of the Log class:

```javascript
newLog = new Log();
```

Log.onLog

Availability
Flash Media Server 2.

Usage
```
myLog.onLog(message)()
```

Parameters
```
message  A log message.
```

Returns
Nothing.

Description
Event handler; invoked when a log message is sent to a log.

NetConnection class

Availability
Flash Communication Server MX 1.0.

The server-side NetConnection class lets you create a two-way connection between a Flash Media Server application instance and an application server, another Flash Media Server, or another Flash Media Server application instance on the same server. You can use NetConnection objects to create powerful applications; for example, you can get weather information from an application server or share an application load with other Flash Media Servers or application instances.

You can use the `NetConnection.connect()` method to connect to an application server for server-to-server interactions using standard protocols (such as HTTP) or connect to another Flash Media Server for sharing audio, video, and data using the Macromedia Real-Time Messaging Protocol format (RTMP) or SSL (RTMPS). SSL in Flash Media Server uses a third-party open source library called OpenSSL.
Method summary for the NetConnection class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetConnection.addHeader()</td>
<td>Adds a context header.</td>
</tr>
<tr>
<td>NetConnection.call()</td>
<td>Invokes a method or operation on a remote server.</td>
</tr>
<tr>
<td>NetConnection.close()</td>
<td>Closes a server connection.</td>
</tr>
<tr>
<td>NetConnection.connect()</td>
<td>Connects to an application server or another Flash Media</td>
</tr>
<tr>
<td></td>
<td>Server server.</td>
</tr>
</tbody>
</table>

Property summary for the NetConnection class

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetConnection.isConnected</td>
<td>Read-only; a Boolean value indicating whether a connection has been made (true) or not (false).</td>
</tr>
<tr>
<td>NetConnection.uri</td>
<td>Read-only; the URI that the NetConnection.connect() method passed.</td>
</tr>
</tbody>
</table>

Event handler summary for the NetConnection class

<table>
<thead>
<tr>
<th>Event handler</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetConnection.onStatus</td>
<td>Called when there is a change in connection status.</td>
</tr>
</tbody>
</table>

Constructor for the NetConnection class

**Availability**
Flash Communication Server MX 1.0.

**Usage**

```javascript
new NetConnection()
```

**Parameters**

None.

**Returns**

A NetConnection object.
Description
Constructor; creates a new instance of the NetConnection class.

Example
The following example creates a new instance of the NetConnection class:

```javascript
newNC = new NetConnection();
```

**NetConnection.addHeader()**

**Availability**
Flash Communication Server MX 1.0.

**Usage**

```javascript
myNetConn.addHeader(name, mustUnderstand, object)
```

**Parameters**

- **name** A string; identifies the header and the ActionScript object data associated with it.
- **mustUnderstand** A Boolean value; `true` indicates that the server must understand and process this header before it handles any of the following headers or messages.
- **object** Any ActionScript object.

**Returns**

Nothing.

**Description**
Method; adds a context header to the AMF packet structure. This header is sent with every future AMF packet. If you call `NetConnection.addHeader()` using the same name, the new header replaces the existing header, and the new header persists for the duration of the NetConnection object. You can remove a header by calling `NetConnection.addHeader()` with the name of the header to remove and an undefined object.

**Example**

The following example creates a new NetConnection instance, `nc`, and connects to an application at web server www.foo.com that is listening at port 1929. This application dispatches the service `/blag/SomeCoolService`. The last line of code adds a header called `foo`:

```javascript
nc=new NetConnection();
nc.connect("http://www.foo.com:1929/blag/SomeCoolService");
nc.addHeader("foo", true, new Foo());
```
NetConnection.call()

Availability
Flash Communication Server MX 1.0.

Usage
myNetConnection.call(methodName, [resultObj, p1, ..., pN])

Parameters
methodName  A method specified in the form [objectPath]/method. For example, the command someObj/doSomething tells the remote server to invoke the clientObj.someObj.doSomething method, with all the p1, ..., pN parameters. If the object path is missing, clientObj.doSomething() is invoked on the remote server.
resultObj  An optional parameter that is used to handle return values from the server. The result object can be any object you defined and can have two defined methods to handle the returned result: onResult and onStatus. If an error is returned as the result, onStatus is invoked; otherwise, onResult is invoked.
p1, ..., pN  Optional parameters that can be of any ActionScript type, including a reference to another ActionScript object. These parameters are passed to the methodName specified above when the method is executed on the remote application server.

Returns
For RTMP connections, returns a Boolean value of true if a call to methodName is sent to the client; otherwise, false. For application server connections, it always returns true.

Description
Method; invokes a command or method on a Flash Media Server or an application server to which the application instance is connected. The NetConnection.call method on the server works the same way as the NetConnection.call method on the client: it invokes a command on a remote server.

NOTE
If you want to call a method on a client from a server, use the Client.call() method.

Example
The following example uses RTMP to execute a call from one Flash Media Server to another Flash Media Server. The code makes a connection to the App1 application on server 2 and then invokes the method Sum on server 2:
nc1.connect("rtmp://server2.mydomain.com/App1", "svr2");
nc1.call("Sum", new Result(), 3, 6);
The following server-side ActionScript code is on server 2. When the client is connecting, this code checks to see whether it has an argument that is equal to svr1. If the client has that argument, the `Sum` method is defined so that when the method is called from svr1, svr2 can respond with the appropriate method:

```actionscript
application.onConnect = function(clientObj){
  if(arg1 == "svr1"){
    clientObj.Sum = function(p1, p2){
      return p1 + p2;
    }
  }
  return true;
};
```

The following example uses an AMF request to make a call to an application server. This allows Flash Media Server to connect to an application server and then invoke the `quote` method. The Java adaptor dispatches the call by using the identifier to the left of the dot as the class name and the identifier to the right of the dot as a method of the class.

```javascript
nc = new NetConnection;
nco.connect("http://www.xyz.com/java");
nco.call("myPackage.quote", new Result());
```

**NetConnection.close()**

**Availability**
Flash Communication Server MX 1.0.

**Usage**

```actionscript
myNetConnection.close()
```

**Parameters**
None.

**Returns**
Nothing.

**Description**
Method; closes the connection with the server. After you close the connection, you can reuse the NetConnection instance and reconnect to an old application or connect to a new one.

**NOTE**
The `NetConnection.close` method has no effect on HTTP connections.
Example
The following code closes the NetConnection instance myNetConn:
myNetConn.close();

NetConnection.connect()

Availability
Flash Communication Server MX 1.0.

Usage
myNetConnection.connect(URI, [p1, ..., pN])

Parameters

URI A URI to connect to.

p1, ..., pN Optional parameters that can be of any ActionScript type, including references to other ActionScript objects. These parameters are sent as connection parameters to the application.onConnect event handler for RTMP connections. For AMF connections to application servers, any RTMP parameters are ignored.

Returns
For RTMP connections, a Boolean value of true for success; false otherwise. For AMF connections to application servers, true is always returned.

Description
Method; connects to an application server or another Flash Media Server. The host URI has the following format:

[protocol://]host[:port]/appName[/instanceName]

The following example shows legal URIs:

http://appServer.mydomain.com/webApp
rtmp://rtserver.mydomain.com/realtimeApp
rtmps://rtserver.mydomain.com/secureApp

You can use the NetConnection.connect() method to connect to an application server for server-to-server interactions using standard protocols (such as HTTP) or connect to another Flash Media Server for sharing audio, video, and data using the Macromedia Real-Time Messaging Protocol format (RTMP) or SSL (RTMPS). SSL in Flash Media Server uses a third-party open source library called OpenSSL.
It is good practice to write an `application.onStatus` callback function and check the `NetConnection.isConnected` property for RTMP connections to see whether a successful connection was made. For Action Message Format connections, check `NetConnection.onStatus`.

About secure connections

Flash Media Server accepts only RTMPS connections on designated secure ports. A port is marked as secure by specifying a minus sign (-) in front of the port in the `HostPort` tag of the `Adaptor.xml` file, as shown in the following code:

```
<HostPort>:1935,80,-443</HostPort>
```

This code specifies that FMS can listen on any interface, on ports 1935, 80, and 443, where 443 is designated as a secure port that receives only RTMPS connections. Attempting an RTMPS connection to 1935 or 80 fails because the client tries to perform an SSL “handshake” that the server cannot complete. Similarly, a regular RTMP connection to port 443 fails because the server tries to perform an SSL handshake that the client cannot complete.

You can determine whether a connection to the server is over a secure channel by checking the server-side `Client.secure` property, as shown in the following example:

```
application.onConnect = function(client){
  if (client.secure){
    trace("This client is connected over a secure connection.");
  }
}
```

Creating a debugging connection

You can append a property and key to an Internet connection request to create a debug connection. A debug connection gives you greater access to an application; for example, you can play streams and view shared objects.

**To create a debug connection:**

- Call `NetConnection.connect()` and append a 4-digit debug request number, as shown in the following example:

  ```
  nc.connect("rtmp://fmsaddress/appName/instanceName?_fcs_debugreq=1234
  
  For security, the Internet connection is put in a pending state.
  ```

- Call the server management API `approveDebugSession()` and pass it the same debug request number.

  The debug connection attempts to connect to the application.
If the `Application.allowDebug` property is `true`, the connection is approved.

To provide total security against debug connections, set `application.allowDebug` to `false` in the `application.xml` file. This setting overrides debug connections that are approved in server-side code.

If the connection is approved, `Services.onDebugConnect` is invoked.

If you can use the `onConnect` handler to reject the connection.

**Escaping the connection URI**

It is important to convert (or `escape`) special characters in any URI used to connect to Flash Media Server to URL-encoding. If you do not escape special characters, shared objects cannot function correctly over an Internet connection. Escaping URIs is especially important when creating a debugging connection because there are special characters in the debugging property that must be appended to the URI. Use the following code to escape a URI:

```javascript
function escapeURI(uri){
    index = uri.indexOf('?');
    if (index == -1){
        return uri;
    }

    prefix = uri.substring(0, index);
    uri = uri.substring(index);

    return prefix += escape(uri);
}

basicString = "rtmp://serverName/appName/instance?fcsdebug_req=someNumber";
escapedString = escapeURI(basicString);
nc.connect(escapedString);
```

**Example**

The following example creates an RTMP connection to a Flash Media Server for the `myConn` instance of `NetConnection`:

```javascript
myConn = new NetConnection();
myConn.connect("rtmp://tc.foo.com/myApp/myConn");
```

The following example creates an AMF connection to an application server for the `myConn` instance of `NetConnection`:

```javascript
myConn = new NetConnection();
myConn.connect("http://www.xyz.com/myApp/");
```
The following example creates a debug connection:

```actionscript
cnc_admin = new NetConnection();
cnc_admin.connect("rtmp://tc.foo.com/myApp/myConn?_fcs_debugreq_=1234");
cnc_admin.call("approveDebugSession", null, "myApp/myConn", 1234);
```

### NetConnection.isConnected

**Availability**
Flash Communication Server MX 1.0.

**Usage**

```actionscript
myNetConnection.isConnected
```

**Description**

Property (read-only): A Boolean value that indicates whether a connection has been made. It is set to `true` if there is a connection to the server. It’s a good idea to check this property value in an `onStatus` callback function. This property is always `true` for AMF connections to application servers.

**Example**

The following example uses `NetConnection.isConnected` in an `onStatus` definition to check if a connection has been made:

```actionscript
nc = new NetConnection();
nc.connect("rtmp://tc.foo.com/myApp");
nc.onStatus = function(infoObj){
  if (info.code == "NetConnection.Connect.Success" && nc.isConnected){
    trace("We are connected");
  }
};
```

### NetConnection.onStatus

**Availability**
Flash Communication Server MX 1.0.

**Usage**

```actionscript
myNetConnection.onStatus = function(infoObject) {}
```

**Parameters**

- `infoObject`   An information object. For more information about this parameter, see “Server-Side Information Objects” on page 229.
Returns
Nothing.

Description
Event handler; invoked every time the status of the NetConnection object changes. For example, if the connection with the server is lost in an RTMP connection, the NetConnection.isConnected property is set to false, and NetConnection.onStatus is invoked with a status message of NetConnection.Connect.closed. For AMF connections, NetConnection.onStatus is used only to indicate a failed connection. Use this event handler to check for connectivity.

Example
The following example defines a function for the onStatus handler that outputs messages to indicate whether the NetConnection was successful:

```javascript
nc = new NetConnection();
nc.onStatus = function(info){
  if (info.code == "NetConnection.Connect.Success") {
    _root.gotoAndStop(2);
  } else {
    if (! nc.isConnected){
      _root.gotoAndStop(1);
    }
  }
};
```

NetConnection.uri

Availability
Flash Communication Server MX 1.0.

Usage
myNetConnection.uri

Description
Property (read-only): A string indicating the URI that was passed by the NetConnection.connect() method. This property is set to null before a call to NetConnection.connect() or after NetConnection.close is called.
**protectObject()**

**Availability**
Flash Media Server 2.

**Usage**

```
protectObject(userObj)
```

**Parameters**

- **userObj**  An object to wrap in a C wrapper object.

**Returns**
An object.

**Description**
Method (global); protects user-defined or built-in objects behind a C wrapper object. The methods of any object that has been passed to the `protectObject()` function become system calls because application code can never access or inspect the methods directly—only through the wrapper. You can only use this function in the secure.asc file.

Flash Media Server 2 has two script execution modes: secure and normal. In secure mode, only the secure.asc file (if it exists) is loaded and evaluated—no other application scripts are loaded. Also, the global `getGlobal()` and `protectObject()` functions are only available in secure mode. These are very powerful because they provide complete access to the script execution environment and let you create system objects. Once the secure.asc file is loaded, the server switches to normal script execution mode until the application is unloaded.

After an object is protected, you must make sure it isn't directly accessible; don't reference it in global variables or make it a member of an accessible object. The wrapper object returned by the `protectObject()` call dispatches all method invocations to the underlying user object but blocks access to member data. As a result, you can't enumerate or modify members directly. The wrapper object keeps an outstanding reference to the underlying user object, which ensures that the object is valid. The wrapper follows normal reference rules and exists while it is referred to.

For more information, see “Implementing secure system objects” in Developing Media Applications.
Example
After secure.asc is executed, calls to load() are directed through the user-defined system call, as shown in the following example:

```javascript
var sysobj = {};
sysobj._load = load; // Hide the load function
load = null; // Make it unavailable unpriviliged code
sysobj.load = function(fname){
    // User-defined code to validate/modify fname
    return this._load(fname);
}
// Grab the global object
var global = getGlobal();

// Now protect our sysobj and make it available as
// 'system' globally. Also, set its attributes
// so that it is read-only and not deletable.

global["system"] = protectObject(sysobj);
setAttributes( global, "system", false, true, true );

// Now add a global load() function for compatibility.
// Make it read-only and non-deletable.

global["load"] = function(path){
    return system.load( path );
}
setAttributes( global, "load", false, true, true );

See also
getGlobal()
```

setAttributes()

Availability
Flash Media Server 2.

Usage
setAttributes(obj, propName, enumerable, readonly, permanent)
Parameters

obj  An object.

propName  A string; the URL to upload variables. The name of the property that exists in the obj parameter. Setting attributes on nonexistent properties has no effect.

enumerable  One of the following values: true, false, or null. Makes a property enumerable if true or nonenumerable if false; a null value leaves this attribute unchanged. Nonenumerable properties are hidden from enumerations (for var i in obj).

readonly  One of the following values: true, false, or null. Makes a property read-only if true or writable if false; a null value leaves this attribute unchanged. Any attempt to assign a new value will be ignored. Typically, you assign a value to a property while the property is writable and then make the property read-only.

permanent  One of the following values: true, false, or null. Makes a property permanent (nondeletable) if true or deletable if false; a null value leaves this attribute unchanged. Any attempt to delete a permanent property (by calling delete obj.prop) is ignored.

Description

Method (global); lets you prevent certain methods and properties from being enumerated, writable, and deletable.

In a Flash Media Server server-side script, all properties in an object are always enumerable, writable, and deletable. You can call setAttributes() to change the default attributes of a property or to define constants.

Example

The following code prevents the __resolve method from appearing in enumerations:

```javascript
Object.prototype.__resolve = function(methodName){ ... };
setAttributes( Object.prototype, "__resolve", false, null, null );
```

The following example creates three constants on a Constants object and makes them permanent and read-only:

```javascript
Constants.Kilo = 1000; setAttributes(Constants, "Kilo", null, true, true);
Constants.Mega = 1000*Constants.Kilo;
setAttributes(Constants, "Mega", null, true, true);
Constants.Giga = 1000*Constants.Mega; setAttributes(Constants, "Giga", null, true, true);
```
setInterval()

Availability
Flash Communication Server MX 1.0.

Usage
```javascript
setInterval(function, interval[, p1, ..., pN])
setInterval(object, methodName, interval[, p1, ..., pN])
```

Parameters
- `function` The name of a defined ActionScript function or a reference to an anonymous function.
- `object` An object derived from the ActionScript Object object.
- `methodName` The name of the method to call on `object`.
- `interval` The time (interval) between calls to `function`, in milliseconds.
- `p1, ..., pN` Optional parameters passed to `function`.

Returns
A unique ID for this call. If the interval is not set, the method returns -1.

Description
Method (global); continually calls a function or method at a specified time interval until the `clearInterval()` method is called. This method allows a server-side script to run a generic routine. The `setInterval` method returns a unique ID that you can pass to the `clearInterval()` method to stop the routine.

**NOTE**
Standard JavaScript supports an additional usage for the `setInterval()` method, `setInterval(stringToEvaluate, timeInterval)`, which is not supported by server-side ActionScript.

Example
The following example uses an anonymous function to send the message “interval called” to the server log every second:
```javascript
setInterval(function(){trace("interval called");}, 1000);
```

The following example also uses an anonymous function to send the message “interval called” to the server log every second, but it passes the message to the function as a parameter:
```javascript
setInterval(function(s){trace(s);}, 1000, "interval called");
```
The following example uses a named function, `callback1`, to send the message “interval called” to the server log:

```actionscript
function callback1(){
    trace("interval called");
}
setInterval(callback1, 1000);
```

The following example also uses a named function, `callback2`, to send the message “interval called” to the server log, but it passes the message to the function as a parameter:

```actionscript
function callback2(s){
    trace(s);
}
setInterval(callback2, 1000, "interval called");
```

## SharedObject class

### Availability

Flash Communication Server MX 1.0.

The `SharedObject` class lets you store data on the server and share data between multiple client applications in real time. Shared objects can be temporary, or they can persist on the server after an application has closed; you can consider shared objects as real-time data transfer devices.

**NOTE**

This entry explains the server-side `SharedObject` class. You can also create shared objects with the client-side `SharedObject` class. For more information, see “SharedObject class” in Client-Side ActionScript Language Reference for Flash Media Server 2.

The following list describes common ways to use shared objects in server-side ActionScript:

- **Storing and sharing data on a server**
  
  A shared object can store data on the server for other clients to retrieve. For example, you can open a remote shared object, such as a phone list, that is persistent on the server. Whenever a client makes a change to the shared object, the revised data is available to all clients that are currently connected to the object or that connect to it later. If the object is also persistent locally and a client changes the data while not connected to the server, the changes are copied to the remote shared object the next time the client connects to the object.
Sharing data in real time

A shared object can share data among multiple clients in real time. For example, you can open a remote shared object that stores real-time data that is visible to all clients connected to the object, such as a list of users connected to a chat room. When a user enters or leaves the chat room, the object is updated and all clients that are connected to the object see the revised list of chat room users.

It is important to understand the following information about using shared objects in server-side ActionScript:

- The server-side ActionScript method `SharedObject.get()` creates remote shared objects (RSOs); there is no server-side method that creates local shared objects (LSOs).
- The RSO file extension is .fso; RSOs are stored on the server in a subdirectory of the application that created them.
- Server-side shared objects can be non-persistent (exist for the duration of an application instance) or persistent (stored on the server after an application closes).

To create a persistent shared object, set the `persistence` parameter of the `SharedObject.get()` method to `true`. Persistent shared objects let you maintain an application's state.

- Every RSO is identified by a unique name and contains a list of name-value pairs, called properties, like any other ActionScript object. A name must be a unique string and a value can be any ActionScript data type.

To get the value of a server-side shared object property, call `SharedObject.getProperty()`. To set the value of a server-side shared object property, call `SharedObject.setProperty()`.

- To clear a shared object, call the `SharedObject.clear()` method; to delete multiple shared objects, call the `Application.clearSharedObjects()` method.

Server-side shared objects can be owned by the current application instance or by another application instance. The other application instance can be on the same server or on a different server. References to shared objects that are owned by a different application instance are called proxied shared objects.

If you write a server-side script that modifies multiple properties, you can prevent other clients from modifying the object during the update by calling the `SharedObject.lock()` method before updating the object. Then you can call `SharedObject.unlock()` to commit the changes and allow other changes to be made. Call `SharedObject.mark()` to deliver change events in groups within the `lock()` and `unlock()` methods.

**NOTE**

Unlike client-side shared objects, server-side shared objects do not have a data property.
When you get a reference to a proxied shared object, any changes made to the object are sent to the instance that owns the object. The success or failure of any changes are sent using the `SharedObject.onSync` event handler, if it is defined.

The `SharedObject.lock()` and `SharedObject.unlock()` methods cannot lock or unlock proxied shared objects.

For more information on shared objects, see "About streams and shared objects", "Shared object flow", and "Shared object files" in Developing Media Applications.

### Method summary for the SharedObject class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SharedObject.clear()</code></td>
<td>Deletes all the properties of a persistent shared object.</td>
</tr>
<tr>
<td><code>SharedObject.close()</code></td>
<td>Unsubscribes from a shared object.</td>
</tr>
<tr>
<td><code>SharedObject.commit()</code></td>
<td>Static; stores either a specific persistent shared object instance or all persistent shared object instances that are marked dirty.</td>
</tr>
<tr>
<td><code>SharedObject.flush()</code></td>
<td>Saves the current state of a persistent shared object.</td>
</tr>
<tr>
<td><code>SharedObject.get()</code></td>
<td>Static; creates a shared object or returns a reference to an existing shared object.</td>
</tr>
<tr>
<td><code>SharedObject.getProperty()</code></td>
<td>Gets the value of a shared object property.</td>
</tr>
<tr>
<td><code>SharedObject.getPropertyNames()</code></td>
<td>Returns array of all the current valid properties in the shared object.</td>
</tr>
<tr>
<td><code>SharedObject.lock()</code></td>
<td>Locks the shared object instance. Prevents any changes to this object by clients until the <code>SharedObject.unlock()</code> method is called.</td>
</tr>
<tr>
<td><code>SharedObject.mark()</code></td>
<td>Delivers all change events to a subscribing client as a single message.</td>
</tr>
<tr>
<td><code>SharedObject.purge()</code></td>
<td>Causes the server to remove all deleted properties that are older than the specified version.</td>
</tr>
<tr>
<td><code>SharedObject.send()</code></td>
<td>Sends a message to the client subscribing to this shared object.</td>
</tr>
<tr>
<td><code>SharedObject.setProperty()</code></td>
<td>Sets a new value for a shared object property.</td>
</tr>
<tr>
<td><code>SharedObject.size()</code></td>
<td>Returns the number of valid properties in a shared object.</td>
</tr>
<tr>
<td><code>SharedObject.unlock()</code></td>
<td>Unlocks a shared object instance that was locked with <code>SharedObject.lock()</code>.</td>
</tr>
</tbody>
</table>
Property summary for the SharedObject class

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SharedObject.autoCommit</td>
<td>A Boolean value indicating whether the server periodically commits all persistent shared objects to be stored (true) or not (false).</td>
</tr>
<tr>
<td>SharedObject.isDirty</td>
<td>A Boolean value indicating whether the persistent SharedObject has been modified since the last time it was stored (true) or not (false).</td>
</tr>
<tr>
<td>SharedObject.name</td>
<td>The name of a shared object.</td>
</tr>
<tr>
<td>SharedObject.resyncDepth</td>
<td>The depth that indicates when the deleted values of a shared object should be permanently deleted.</td>
</tr>
<tr>
<td>SharedObject.version</td>
<td>The current version number of a shared object.</td>
</tr>
</tbody>
</table>

Event summary for the SharedObject class

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SharedObject.handlerName</td>
<td>A placeholder for an event handler; the handler is invoked when a shared object receives a broadcast message with the same name.</td>
</tr>
<tr>
<td>SharedObject.onStatus</td>
<td>Reports errors, warnings, and status messages for a shared object.</td>
</tr>
<tr>
<td>SharedObject.onSync</td>
<td>Invoked when a shared object changes.</td>
</tr>
</tbody>
</table>

SharedObject.autoCommit

**Availability**
Flash Media Server 2.

**Usage**
mySO.autoCommit

**Description**
Property; a Boolean value indicating whether the server periodically stores all persistent shared objects (true) or not (false). If autoCommit is false, the application must call SharedObject.commit() to save the shared object; otherwise, the data is lost.
This property is `true` by default. This can be overridden by specifying the initial state using the following configuration key in the Application.xml file, as shown in the following example:

```xml
<SharedObjManager>
  <AutoCommit>false</AutoCommit>
</SharedObjManager>
```

**SharedObject.clear()**

**Availability**
Flash Communication Server MX 1.0.

**Usage**
```actionscript
mySO.clear()
```

**Parameters**
None.

**Returns**
Returns `true` if successful; `false` otherwise.

**Description**
Method; deletes all the properties of a single shared object and sends a `clear` event to all clients that subscribe to a persistent shared object. The persistent data object is also removed from a persistent shared object.

**Example**
The following example calls the `clear()` method on the shared object `myShared`:

```actionscript
var myShared = SharedObject.get("foo", true);
var len = myShared.clear();
```

**See also**
`Application.clearSharedObjects()`

**SharedObject.close()**

**Availability**
Flash Communication Server MX 1.0.

**Usage**
```actionscript
mySO.close()
```
Parameters
None.

Returns
Nothing.

Description
Method; detaches a reference from a shared object. A call to the `SharedObject.get()` method returns a reference to a shared object instance. The reference is valid until the variable that holds the reference is no longer in use and the script is garbage-collected. To destroy a reference immediately, you can call `SharedObject.close()`. You can use `SharedObject.close()` when you don't want to proxy a shared object any longer.

Example
In the following example, `mySO` is attached as a reference to shared object `foo`. When you call `mySO.close` you detach the reference `mySO` from the shared object `foo`.

```javascript
mySO = SharedObject.get("foo");
// Insert code here.
mySO.close();
```

See also
`SharedObject.get()`

**SharedObject.commit()**

Availability
Flash Media Server 2.

Usage
`SharedObject.commit([name])`

Parameters
`name` A string indicating the name of the persistent shared object instance to store. If no name is specified or an empty string is passed, all persistent shared objects are stored. This parameter is optional.

Returns
A Boolean value indicating success (`true`) or failure (`false`).

Description
Method (static); stores either a specific persistent shared object instance or all persistent shared object instances with an isDirty property whose value is true. Use this method if the SharedObject.autoCommit property is false and you need to manage when a shared object is stored locally.

Example
The following code commits all dirty shared objects to local storage when the application stops:
```javascript
application.onAppStop = function (info){
  // Insert code here.
  SharedObject.commit();
}
```

SharedObject.flush()

Availability
Flash Communication Server MX 1.0.

Usage
```javascript
mySO.flush()
```

Parameters
None.

Returns
A Boolean value of true if successful, false otherwise.

Description
Method; saves the current state of a persistent shared object.

Example
The following example places a reference to the shared object foo in the variable myShared. It then locks the shared object instance so that no one can make any changes to it, and then saves the shared object by calling myShared.flush(). After the shared object is saved, it is unlocked so that further changes can be made.
```javascript
var myShared = SharedObject.get("foo", true);
myShared.lock();
// Insert code here that operates on the shared object.
myShared.flush();
myShared.unlock();
```
SharedObject.get()

Availability
Flash Communication Server MX 1.0.

Usage
SharedObject.get(name, persistence [, netConnection])

Parameters
name    Name of the shared object instance to return.
persistence A Boolean value: true for a persistent shared object; false for a nonpersistent shared object. If no value is specified, the default value is false.
netConnection A NetConnection object that represents a connection to an application instance. You can pass this parameter to get a reference to a shared object on another server or a shared object that is owned by another application instance. All update notifications for the shared object specified by the name parameter are proxied to this instance, and the remote instance notifies the local instance when a persistent shared object changes. The NetConnection object that is used as the netConnection parameter does not need to be connected when you call SharedObject.get(). The server connects to the remote shared object when the NetConnection state changes to connected. This parameter is optional.

Returns
A reference to an instance of the SharedObject class.

Description
Method (static); creates a shared object or returns a reference to an existing shared object. To perform any operation on a shared object, the server-side script must get a reference to the shared object using the SharedObject.get() method. If the requested object is not found, a new instance is created.

There are two types of shared objects, persistent and nonpersistent, and they have separate namespaces. This means that a persistent and nonpersistent shared object can have the same name and exist as two distinct shared objects. Shared objects are scoped to the namespace of the application instance and are identified by a string. The shared object names should conform to the URI specification.
You can also call `SharedObject.get()` to get a reference to a shared object that is in a namespace of another application instance. This instance can be on the same server or on a different server and is called a *proxied shared object*. To get a reference to a shared object from another instance, create a `NetConnection` object and use the `NetConnection.connect()` method to connect to the application instance that owns the shared object. Pass the `NetConnection` object as the `netConnection` parameter of the `SharedObject.get()` method. The server-side script must get a reference to a proxied shared object before there is a request for the shared object from any client. To do this, call `SharedObject.get()` in the `application.onAppStart` handler.

If you call `SharedObject.get()` with a `netConnection` parameter and the local application instance already has a shared object with the same name, the shared object converts to a proxied shared object. All shared object messages for clients connected to a proxied shared object are sent to the master instance.

If the connection state of the `NetConnection` object that was used as the `netConnection` parameter changes state from connected to disconnected, the proxied shared object is set to idle and any messages received from subscribers are discarded. The `NetConnection.onStatus` handler is called when a connection is lost. You can then re-establish a connection to the remote instance and call `SharedObject.get()`, which changes the state of the proxied shared object from idle to connected.

If you call `SharedObject.get()` with a new `NetConnection` object on a proxied shared object that is already connected and if the URI of the new `NetConnection` object doesn't match the current `NetConnection` object, the proxied shared object unsubscribes from the previous shared object, sends a `clear` event to all subscribers, and subscribes to the new shared object instance. When a subscribe to a proxied shared object is successful, all subscribers are reinitialized to the new state. This process lets you migrate a shared object from one application instance to another without disconnecting the clients.

Updates received by proxied shared objects from subscribers are checked to see if the update can be rejected based on the current state of the proxied shared object version and the version of the subscriber. If the change can be rejected, the proxied shared object doesn’t forward the message to the remote instance; the reject message is sent to the subscriber.

The corresponding client-side ActionScript method is `SharedObject.getRemote()`.
Example
The following example creates a shared object named `foo` in the function `onProcessCmd`. The function is passed a parameter, `cmd`, that is assigned to a property in the shared object.

```javascript
function onProcessCmd(cmd){
    // Insert code here.
    var shObj = SharedObject.get("foo", true);
    propName = cmd.name;
    shObj.getProperty (propName, cmd.newAddress);
}
```

The following example uses a proxied shared object. A proxied shared object resides on a server or in an application instance (called `master`) that is different than the server or application instance that the client connects to (called `proxy`). When the client connects to the proxy and gets a remote shared object, the proxy connects to the master and gives the client a reference to this shared object. The following code is in the main.asc file:

```javascript
application.appStart = function() {
    nc = new NetConnection();
    nc.connect("rtmp://" + master_server + "+" + master_instance);
    proxySO = SharedObject.get("myProxy",true,nc);
    // Now, whenever the client asks for a persistent
    // shared object called myProxy they will receive myProxy
    // shared object from the master_server/master_instance.
};
```

### SharedObject.getProperty()

**Availability**
Flash Communication Server MX 1.0.

**Usage**

```javascript
mySO.getProperty(name)
```

**Parameters**

- `name` The name of the property in the shared object.

**Returns**
The value of a SharedObject property.

**Description**
Method; retrieves the value of a named property in a shared object. The returned value is a copy associated with the property, and any changes made to the returned value do not update the shared object. To update a property, use the `SharedObject.setProperty()` method.
Example
The following example gets the value of the `name` property and passes it to the `value` variable:

```ACTIONSCRIPT
value = sharedInfo.getProperty(name);
```

See also
`SharedObject.setProperty()`
`SharedObject.getPropertyNames()`

**SharedObject.getPropertyNames()**

**Availability**
Flash Communication Server MX 1.0.

**Usage**
`mySO.getPropertyNames()`

**Parameters**
None.

**Returns**
An array of strings that contain all the property names of a shared object.

**Description**
Method; enumerates all the property names for a given shared object.

**Example**
The following example calls `getPropertyNames()` on the `myInfo` shared object and places the names in the `names` variable. It then enumerates those property names in a for loop.

```ACTIONSCRIPT
myInfo = SharedObject.get("foo");
var addr = myInfo.getProperty("address");
myInfo.setProperty("city", "San Francisco");
var names = myInfo.getPropertyNames();
for (x in names)
    var propVal = myInfo.getProperty(names[x]);
    trace("Value of property " + names[x] + " = " + propVal);
```
SharedObject.handlerName

Availability
Flash Communication Server MX 1.0.

Usage
mySO.onBroadcastMsg = function([p1,..., pN]){}

Parameters
onBroadcastMsg  A property name.
p1, ..., pN  Optional parameters passed to the handler method if the message contains user-defined parameters. These parameters are the user-defined JavaScript objects passed to the SharedObject.send method.

Returns
Any return value is ignored by the server.

Description
Event handler; a placeholder for an event handler (onBroadcastMsg in the Usage example above); the handler is invoked when a shared object receives a broadcast message with the same name. You must define a function object and assign it to the event handler. A shared object can receive a broadcast message from the client-side SharedObject.send() method.
The this keyword used in the body of the function is set to the shared object instance returned by SharedObject.get().
If you don't want the server to receive a particular broadcast message, do not define this handler.

Example
The following example defines a function called broadcastMsg:
var mySO = SharedObject.get("userList", false);
mySO.broadcastMsg = function(msg1, msg2){
    trace(msg1 + ": " + msg2);
};
SharedObject.isDirty

**Availability**
Flash Media Server 2.

**Usage**
mySO.isDirty

**Description**
Property (read-only); a Boolean value indicating whether a persistent shared object has been modified since the last time it was stored (true) or not (false).

The **SharedObject.commit()** method stores shared objects with an **isDirty** property that is true.

This property is always false for non-persistent shared objects.

**Example**
The following example saves the so shared object if it has been changed:

```javascript
var so = SharedObject.get("foo", true);
if (so.isDirty){
  SharedObject.commit(so.name);
}
```

SharedObject.lock()

**Availability**
Flash Communication Server MX 1.0.

**Usage**
mySO.lock()

**Parameters**
None.

**Returns**
An integer indicating the lock count: 0 or greater indicates success; -1 indicates failure. For proxied shared objects, always returns -1.
Description
Method; locks the shared object instance. This method gives the server-side script exclusive access to the shared object; when the SharedObject.unlock() method is called, all changes are batched and one update message is sent, through the SharedObject.onSync handler, to all the clients that subscribe to this shared object. If you nest the SharedObject.lock() and SharedObject.unlock() methods, make sure there is an unlock() method for every lock() method; otherwise, clients are blocked from accessing the shared object.
You cannot use the SharedObject.lock() method on proxied shared objects.

Example
The following example locks the myShared shared object, executes the code that is to be inserted, and then unlocks the object:
```javascript
var myShared = SharedObject.get("foo");
myShared.lock();
// Insert code here that operates on the shared object.
myShared.unlock();
```

SharedObject.mark()

Availability
Flash Media Server 2.

Usage
```javascript
mySO.mark(handlerName, p1, ..., pN)
```

Parameters
- `handlerName` Calls the specified function property on the client-side SharedObject instance. For example, if the `handlerName` parameter is `onChange`, the client invokes the SharedObject.onChange() method with all the `p1, ..., pN` parameters.

**Note**
Do not use a built-in method name for a handler name. For example, the subscribing stream will be closed if the handler name is `close`. 
Parameters of any ActionScript type, including references to other ActionScript objects. These parameters are passed to the specified handler when it is executed on the Flash client.

Returns
A Boolean value. Returns true if the message can be dispatched to the client; false otherwise.

Description
Method; delivers all change events to a subscribing client as a single message. In a server-side script, you can call the setProperty() method to update multiple shared object properties between a call to the lock() and unlock() methods. All subscribing clients receive a change event notification through the SharedObject.onSync handler. However, because the server may collapse multiple messages to optimize bandwidth, the change event notifications might not be sent in the same order as they were in the code. Use the mark() method when you want to execute code after all the properties in a set have been updated.

If an application calls the lock() method to lock a shared object, calls the setProperty() method to make a series of property changes, and calls the mark() method before calling the unlock() method, Flash Media Server guarantees that all the change events are delivered to the subscribing clients as a single message. You can call the handlerName parameter passed to the mark() method knowing that all property changes before the mark() call have been updated.

Example
The following example calls the mark() method twice to group two sets of shared object property updates for clients:

```javascript
var myShared = SharedObject.get("foo", true);

myShared.lock();
myShared.setProperty("name", "Stephen");
myShared.setProperty("address", "Xyz lane");
myShared.setProperty("city", "SF");
myShared.mark("onAdrChange", "name");
myShared.setProperty("account", 12345);
myShared.mark("onActChange");
myShared.unlock();
```
The following example shows the receiving client-side script:

```javascript
connection = new NetConnection();
connection.connect("rtmp://flashmediaserver/someApp");
var x = SharedObject.get("foo", connection.uri, true);
x.connect(connection);
x.onAdrChange = function(str) {
    // Shared Object has been updated,
    // can look at the "name", "address" and "city" now
}

x.onActChange = function(str) {
    // Shared Object has been updated
    // can look at the "account" property now
}
```

**SharedObject.name**

**Availability**
Flash Communication Server MX 1.0.

**Usage**

```javascript
mySO.name
```

**Description**
Property (read-only); the name of a shared object.

**SharedObject.onStatus**

**Availability**
Flash Communication Server MX 1.0.

**Usage**

```javascript
mySO.onStatus = function(info) {}
```

**Parameters**

- `info` An information object. For more information, see “Client-Side Information Objects,” in the Client-Side ActionScript Language Reference for Flash Media Server 2.

**Returns**
Nothing.
Description
Event handler; reports errors, warnings, and status messages associated with either a local instance of a shared object or a persistent shared object.

Example
The following example defines an `onStatus` event handler for the shared object `soInstance`:

```actionscript
soInstance = SharedObject.get("foo", true);
soInstance.onStatus = function(infoObj){
    //Handle SO status messages.
};
```

`SharedObject.onSync`

Availability
Flash Communication Server MX 1.0.

Usage
```
mySO.onSync = function(list){}
```

Parameters
`list` An array of objects that contain information about the properties of a shared object that have changed since the last time the `onSync` handler was called. The notifications for proxied shared objects are different than for shared objects that are owned by the local application instance. The following tables list the descriptions for each type of shared object.

Local shared objects

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>change</td>
<td>A property was changed by a subscriber.</td>
</tr>
<tr>
<td>delete</td>
<td>A property was deleted by a subscriber.</td>
</tr>
<tr>
<td>name</td>
<td>The name of a property that has changed or been deleted.</td>
</tr>
<tr>
<td>oldValue</td>
<td>The old value of a property. This is true for both change and delete messages; on the client, oldValue is not set for delete.</td>
</tr>
</tbody>
</table>

**NOTE** Changing or deleting a property on the server side using the `SharedObject.setProperty()` method always succeeds, so there is no notification of these changes.
Proxied shared objects

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>A server change of the shared object was accepted.</td>
</tr>
<tr>
<td>reject</td>
<td>A server change of the shared object was rejected. The value on the remote instance was not changed.</td>
</tr>
<tr>
<td>change</td>
<td>A property was changed by another subscriber.</td>
</tr>
<tr>
<td>delete</td>
<td>A property was deleted. This notification can occur when a server deletes a shared object or if another subscriber deletes a property.</td>
</tr>
<tr>
<td>clear</td>
<td>All the properties of a shared object are deleted.</td>
</tr>
<tr>
<td>name</td>
<td>The name of a property that has changed or been deleted.</td>
</tr>
<tr>
<td>oldValue</td>
<td>The old value of the property. This is valid only for the reject, change, and delete codes.</td>
</tr>
</tbody>
</table>

**NOTE**
The `SharedObject.onSync` handler is invoked when a shared object has been successfully synchronized with the server. The list object may be empty if there is no change in the shared object.

**Returns**
Nothing.

**Description**
Event handler; invoked when a shared object changes. Use the `onSync` handler to define a function that handles changes made to a shared object by subscribers.

For proxied shared objects, define the function to get the status of changes made by the server and other subscribers.

**NOTE**
You cannot define the `onSync` handler on the `prototype` property of the `SharedObject` class in server-side ActionScript.
Example

The following example creates a function that is invoked whenever a property of the shared object `so` changes:

```javascript
// Create a new NetConnection object.
nc = new NetConnection();
nc.connect("rtmp://server1.xyx.com/myApp");
// Create the shared object.
so = SharedObject.get("MasterUserList", true, nc);
// The list parameter is an array of objects containing information
// about successfully or unsuccessfully changed properties
// from the last time onSync() was called.
so.onSync = function(list) {
    for (var i = 0; i < list.length; i++) {
        switch (list[i].code ) {
            case "success":
                trace ("success");
                break;
            case "change":
                trace ("change");
                break;
            case "reject":
                trace ("reject");
                break;
            case "delete":
                trace ("delete");
                break;
            case "clear":
                trace ("clear");
                break;
        }
    }
};
```

`SharedObject.purge()`

**Availability**
Flash Communication Server MX 1.0.

**Usage**

```javascript
mySO.purge(version)
```

**Parameters**

- `version` A version number. All deleted data that is older than this version is removed.
**Returns**
Nothing.

**Description**
Method; causes the server to purge all deleted properties that are older than the specified version. Although you can also accomplish this task by setting the `SharedObject.resyncDepth` property, the `SharedObject.purge()` method gives the script more control over which properties to delete.

**Example**
The following example deletes all the properties of the `myShared` shared object that are older than the value of `myShared.version - 3`:

```javascript
var myShared = SharedObject.get("foo", true);
myShared.lock();
myShared.purge(myShared.version - 3);
myShared.unlock();
```

**SharedObject.resyncDepth**

**Availability**
Flash Communication Server MX 1.0.

**Usage**
`mySO.resyncDepth`

**Description**
Property; an integer that indicates when the deleted properties of a shared object should be permanently deleted. You can use this property in a server-side script to resynchronize shared objects and to control when shared objects are deleted. The default value is infinity.

If the current revision number of the shared object minus the revision number of the deleted property is greater than the value of `SharedObject.resyncDepth`, the property is deleted. Also, if a client connecting to this shared object has a client revision that, when added to the value of `SharedObject.resyncDepth` is less than the value of the current revision on the server, all the current elements of the client shared object are deleted, the valid properties are sent to the client and the client receives a “clear” message.
This method is useful when you add and delete many properties and you don't want to send too many messages to the Flash client. Suppose a client is connected to a shared object that has 12 properties and then disconnects. After that client disconnects, other clients that are connected to the shared object delete 20 properties and add 10 properties. When the client reconnects, it could, for example, receive a delete message for the 10 properties it previously had and then a change message on two properties. You could use `SharedObject.resyncDepth` to send a “clear” message, followed by a change message for two properties, which saves the client from receiving 10 delete messages.

**Example**
The following example resynchronizes the shared object `mySO` if the revision number difference is greater than 10:

```actionscript
mySo = SharedObject.get("foo");
mySo.resyncDepth = 10;
```

**SharedObject.send()**

**Availability**
Flash Communication Server MX 1.0.

**Usage**
`mySO.send(methodName, [p1, ..., pN])`

**Parameters**
- `methodName` The name of a method on a client shared object instance. For example, if you specify `doSomething`, the client must invoke the `SharedObject.doSomething` method, with all the `p1, ..., pN` parameters.
- `p1, ..., pN` Parameters of any ActionScript type, including references to other ActionScript objects. These parameters are passed to the specified `methodName` when it executes on the Flash client.

**Returns**
A Boolean value of `true` if the message was sent to the client; `false` otherwise.

**Description**
Method; executes a method in a client-side script. You can use `SharedObject.send()` to asynchronously execute a method on all the Flash clients subscribing to a shared object. The server does not receive any notification from the client on the success, failure, or return value in response to this message.
Example
The following example calls the SharedObject.send() method to execute the
doSomething() method in the client-side ActionScript and passes doSomething() the string"this is a test":

```javascript
var myShared = SharedObject.get("foo", true);
myShared.send("doSomething", "this is a test");
```

The following example is the client-side ActionScript code that defines the doSomething method:

```javascript
connection = new NetConnection();
connection.connect("rtmp://www.macromedia.com/someApp");
var x = SharedObject.getRemote("foo", connection.uri, true);
x.connect(connection);
x.doSomething = function(str) {
    // Process the str object.
};
```

**SharedObject.setProperty()**

**Availability**
Flash Communication Server MX 1.0.

**Usage**
```javascript
mySO.setProperty(name, value)
```

**Parameters**
- **name**  The name of the property in the shared object.
- **value**  An ActionScript object associated with the property, or null to delete the property.

**Returns**
Nothing.

**Description**
Method; updates the value of a property in a shared object.

The name parameter on the server side is the same as an attribute of the data property on the client side. For example, the following two lines of code are equivalent; the first line is server-side ActionScript, and the second is client-side ActionScript:

```javascript
mySO.setProperty(nameVal, "foo");
clientSO.data[nameVal] = "foo";
```
A shared object property can be modified by a client between successive calls to `SharedObject.getProperty()` and `SharedObject.setProperty()`. If you want to preserve transactional integrity, call the `SharedObject.lock()` method before modifying the shared object; make sure to call `SharedObject.unlock()` when you finish making modifications. If you call `SharedObject.setProperty()` without first calling `SharedObject.lock()`, the change is made to the shared object, and all object subscribers are notified before `SharedObject.setProperty()` returns. If you call `SharedObject.lock()` before you call `SharedObject.setProperty()`, all changes are batched and sent when the `SharedObject.unlock()` method is called. The `SharedObject.onSync` handler on the client side is invoked when the local copy of the shared object is updated.

**NOTE**
If only one source (whether client or server) is updating a shared object in a server-side script, you don’t need to use the `lock()` or `unlock()` methods or the `onSync` handler.

**Example**
The following example uses the `SharedObject.setProperty()` method to create the property `city` with the value San Francisco. It then enumerates all the property values in a `for` loop and prints out the results by using a `trace()` action.

```actionscript
myInfo = SharedObject.get("foo");
var addr = myInfo.getProperty("address");
myInfo.setProperty("city", "San Francisco");
var names = sharedInfo.getPropertyNames();
for (x in names){
    var propVal = sharedInfo.getProperty(names[x]);
    trace("Value of property " + names[x] + " = " + propVal);
}
```

**See also**
`SharedObject.getProperty()`

**SharedObject.size()**

**Availability**
Flash Communication Server MX 1.0.

**Usage**
```actionscript
mySO.size()
```

**Parameters**
None.
**Returns**
An integer indicating the number of properties.

**Description**
Method; returns the total number of valid properties in a shared object.

**Example**
The following example gets the number of properties of a shared object and assigns that number to the variable `len`:
```javascript
var myShared = SharedObject.get("foo", true);
var len = myShared.size();
```

**SharedObject.unlock()**

**Availability**
Flash Communication Server MX 1.0.

**Usage**
```javascript
mySO.unlock()
```

**Parameters**
None.

**Returns**
An integer indicating the lock count: 0 or greater if successful, -1 otherwise. For proxied shared objects, this method always returns -1.

**Description**
Method; unlocks a shared object instance. It causes the script to relinquish exclusive access to the shared object and lets other clients update the instance. This method also causes the server to commit all changes made after the `SharedObject.lock()` method is called and sends an update message to all clients.

You cannot use the `SharedObject.unlock` method on proxied shared objects.

**Example**
The following example shows how easy it is to unlock a shared object:
```javascript
var myShared = SharedObject.get("foo", true);
myShared.lock();
// Insert code to manipulate the shared object.
myShared.unlock();
```
See also
SharedObject.lock()

**SharedObject.version**

**Availability**
Flash Communication Server MX 1.0.

**Usage**

```javascript
mySO.version
```

**Description**
Property (read-only); the current version number of the shared object. Changes made to the shared object either by the Flash Player client or by the server-side script using the `SharedObject.setProperty()` method increment the value of the `version` property.

**SOAPCall class**

**Availability**
Flash Media Server 2.

The SOAPCall class is the object type that is returned from all web service calls. These objects are most typically constructed automatically when a WSDL is parsed and a stub is generated.

**Property summary for the SOAPCall class**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOAPCall.request</td>
<td>An XML object that represents the current SOAP request.</td>
</tr>
<tr>
<td>SOAPCall.response</td>
<td>An XML object that represents the most recent SOAP response.</td>
</tr>
</tbody>
</table>

**Event handler summary for the SOAPCall class**

<table>
<thead>
<tr>
<th>Event handler</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOAPCall.onFault</td>
<td>Invoked when a method has failed and returned an error.</td>
</tr>
<tr>
<td>SOAPCall.onResult</td>
<td>Invoked when a method has successfully invoked and returned.</td>
</tr>
</tbody>
</table>
SOAPCall.onFault

Availability
Flash Media Server 2.

Usage
mySOAPCall.onFault(fault)

Parameters
fault The fault parameter is an object version of an XML SOAP Fault (see “SOAPFault
class” on page 142).

Returns
Nothing.

Description
Event handler; invoked when a method has failed and returned an error.

SOAPCall.onResult

Availability
Flash Media Server 2.

Usage
mySOAPCall.onResult(result){}

Parameters
result The decoded ActionScript object returned by the operation (if any). To get the raw
XML returned instead of the decoded result, access the SOAPCall.response property.

Returns
Nothing.

Description
Event handler; invoked when a method has successfully invoked and returned.
SOAPCall.request

Availability
Flash Media Server 2.

Usage
mySOAPCall.request

Description
Property; an XML object that represents the current SOAP request.

SOAPCall.response

Availability
Flash Media Server 2.

Usage
mySOAPCall.response

Description
Property; an XML object that represents the most recent SOAP response.

SOAPFault class

Availability
Flash Media Server 2.

The SOAPFault class is the object type of the error object returned to WebService.onFault and SOAPCall.onFault functions. This object is returned as the result of a failure and is an ActionScript mapping of the SOAP Fault XML type.

Property summary for the SOAPFault class

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOAPFault.detail</td>
<td>A string indicating the application-specific information associated with the error, such as a stack trace or other info returned by the web service engine.</td>
</tr>
<tr>
<td>SOAPFault.faultactor</td>
<td>A string indicating the source of the fault.</td>
</tr>
</tbody>
</table>
### SOAPFault.detail

**Availability**
Flash Media Server 2.

**Usage**
mySOAPFault.detail

**Description**
Property; a string indicating the application-specific information associated with the error, such as a stack trace or other info returned by the web service engine.

### SOAPFault.faultactor

**Availability**
Flash Media Server 2.

**Usage**
mySOAPFault.faultactor

**Description**
Property; a string indicating the source of the fault. This property is optional if an intermediary is not involved.

### SOAPFault.faultcode

**Availability**
Flash Media Server 2.

**Usage**
mySOAPFault.faultcode

**Description**
Property; a string indicating the short standard qualified name describing the error.
SOAPFault.faultstring

Availability
Flash Media Server 2.

Usage
mySOAPFault.faultstring

Description
Property; a string indicating the human-readable description of the error.

Example
The following example shows the fault code in a text field if the Web Service Definition Language (WSDL) fails to load:

```javascript
// Prepare the WSDL location:
var wsdlURI = "http://www.flash-db.com/services/ws/companyInfo.wsdl";

// Instantiate the web service object using the WSDL location:
stockService = new WebService(wsdlURI);

// Handle the WSDL parsing and web service instantiation event.
stockService.onLoad = function(wsdl){
    wsdlField.text = wsdl;
}

// If the wsdl fails to load the onFault event is fired:
stockService.onFault = function(fault){
    wsdlField.text = fault.faultstring;
}
```

Stream class

Availability
Flash Communication Server MX.

The Stream class lets you handle each stream in a Flash Media Server application. A stream is a one-way connection between Flash Player and Flash Media Server, or between two servers. Flash Media Server automatically creates a Stream object with a unique name when `NetStream.play` or `NetStream.publish` are called in a client-side script. You can also create a stream in server-side ActionScript by calling `Stream.get()`. A user can access multiple streams at the same time, and there can be hundreds or thousands of Stream objects active at the same time.
You can use the properties and methods of the Stream class to shuffle streams in a playlist, pull streams from other servers, and play and record streams.
You can also use the Stream class to play MP3 files over a stream, as well as ID3 tags associated with MP3 files.
You can create other Stream properties of any legal ActionScript type, including references to other ActionScript objects, for a particular instance of the Stream class. The properties are available until the stream is removed from the application.
For more information about streams, see the “NetStream class” entry in the Client-Side ActionScript Language Reference for Flash Media Server 2.

Property summary for the Stream class

<table>
<thead>
<tr>
<th>Property (read-only)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream.bufferTime</td>
<td>Indicates how long to buffer messages before a stream is played.</td>
</tr>
<tr>
<td>Stream.name</td>
<td>The unique name of a live stream.</td>
</tr>
<tr>
<td>Stream.syncWrite</td>
<td>A Boolean value that controls when a stream writes the contents of the buffer to a FLV file when the stream is recording.</td>
</tr>
</tbody>
</table>

Method summary for the Stream class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream.clear()</td>
<td>Deletes a stream previously recorded by the server.</td>
</tr>
<tr>
<td>Stream.flush()</td>
<td>Flushes a stream.</td>
</tr>
<tr>
<td>Stream.get()</td>
<td>Static; returns a reference to a Stream object.</td>
</tr>
<tr>
<td>Stream.length()</td>
<td>Static; returns the length of a recorded stream in seconds.</td>
</tr>
<tr>
<td>Stream.play()</td>
<td>Controls the data source of the Stream object.</td>
</tr>
<tr>
<td>Stream.record()</td>
<td>Records all the data going into the stream.</td>
</tr>
<tr>
<td>Stream.send()</td>
<td>Sends a call with parameters to all subscribers on a stream.</td>
</tr>
<tr>
<td>Stream.setBufferTime()</td>
<td>Sets the length of the buffer time in seconds.</td>
</tr>
<tr>
<td>Stream.setVirtualPath()</td>
<td>Sets the virtual directory path for video stream playback.</td>
</tr>
<tr>
<td>Stream.size()</td>
<td>Static; returns the size of the stream in bytes.</td>
</tr>
</tbody>
</table>
Event handler summary for the Stream class

<table>
<thead>
<tr>
<th>Event handler</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream.onStatus</td>
<td>Called when there is a change in status.</td>
</tr>
</tbody>
</table>

Stream.bufferTime

Availability
Flash Communication Server MX 1.0.

Usage

```actionscript
myStream.bufferTime
```

Description
Property (read-only); indicates how long to buffer messages before a stream plays. This property applies only when playing a stream from a remote server or when playing a recorded stream locally. To set this property use `Stream.setBufferTime`.

A message is data that is sent back and forth between Flash Media Server and Flash Player. The data is divided into small packets (messages), and each message has a type (audio, video, or data).

See also

- `Stream.setBufferTime()`

Stream.clear()

Availability
Flash Communication Server MX 1.0.

Usage

```actionscript
myStream.clear()
```

Parameters
None.

Returns
A Boolean value of `true` if the call succeeds; `false` otherwise.

Description
Method; deletes a recorded stream file (FLV) from the server.
Example
The following example deletes a recorded stream called foo.flv. Before the stream is deleted, the example defines an onStatus handler that uses two information object error codes, NetStream.Clear.Success and NetStream.Clear.Failed, to send status messages to the application log file and Application inspector.

```javascript
s = Stream.get("foo");
if (s){
  s.onStatus = function(info){
    if(info.code == "NetStream.Clear.Success"){
      trace("Stream cleared successfully.");
    }
    if(info.code == "NetStream.Clear.Failed"){
      trace("Failed to clear stream.");
    }
  };
  s.clear();
}
```

See also
Stream.get()
Stream.flush()

Stream.flush()

Availability
Flash Media Server 2.

Usage
myStream.flush()

Parameters
None.

Returns
A Boolean value of true if the buffer was successfully flushed; false otherwise.

Description
Method; flushes a stream. If the stream is used for recording, the flush() method writes the contents of the buffer associated with the stream to the FLV file.
It is highly recommended that you call `flush()` on a stream that contains only data. Synchronization problems can occur if you call the `flush()` method on a stream that contains data and either audio, video, or both.

**Example**
The following example flushes the `myStream` stream:

```javascript
// Setup the server stream
application.myStream = Stream.get("foo");

if (application.myStream)
  application.myStream.record();
  application.myStream.send("test", "hello world");
  application.myStream.flush();

```

### Stream.get()

**Availability**
Flash Communication Server MX 1.0.

**Usage**
`myStream.get(name)`

**Parameters**
- `name` The name of the stream instance to return.

**Returns**
A reference to a stream instance.

**Description**
Method (static); returns a reference to a Stream object. If the requested object is not found, a new instance is created.

**Examples**
The following example gets the stream `foo` and assigns it to the variable `playStream`. It then calls the `Stream.play` method from `playStream`.

```javascript
function onProcessCmd(cmd){
  var playStream = Stream.get("foo");
  playStream.play("file1", 0, -1);
}
```

**See also**
`Stream.clear()`
Stream.length()

Availability
Flash Communication Server MX 1.0.

Usage
myStream.length(name[, virtualKey])

Parameters
name Name of a recorded stream (FLV) file or MP3 file. To get the length of an MP3 file, precede the name of the file with mp3: (for example, "mp3:beethoven").
virtualKey A string indicating a key value. Starting with Flash Media Server 2, stream names are not always unique; you can create multiple streams with the same name, place them in different physical directories, and use the VirtualDirectory section and VirtualKeys section of the vhost.xml file to direct clients to the appropriate stream. Because the Stream.length() method is not associated with a client, but connects to a stream on the server, you may need to specify a virtual key to identify the correct stream. For more information about keys, see Client.virtualKey. This parameter is optional.

Returns
The length of a recorded stream file or MP3 file in seconds.

Description
Method (static); returns the length of a recorded stream file or MP3 file in seconds. If the requested file is not found, the return value is 0.

Example
The following example gets the length of the recorded stream file myVideo and assigns it to the variable streamLen:

```javascript
function onProcessCmd(cmd){
    var streamLen = Stream.length("myVideo");
    trace("Length: " + streamLen + 
\n");
}
```

The following example gets the length of the MP3 file beethoven.mp3 and assigns it to the variable streamLen:

```javascript
function onProcessCmd(cmd){
    var streamLen = Stream.length("mp3:beethoven");
    trace("Length: " + streamLen + "\n");
}
```
Stream.name

Availability
Flash Communication Server MX 1.0.

Usage
myStream.name

Description
Property (read-only); contains a unique string associated with a live stream. You can also use this as an index to find a stream within an application.

Example
The following function, getStreamName, takes a stream reference as an argument, myStream, and returns the name of the stream associated with it:

```javascript
function getStreamName (myStream) {
  return myStream.name;
}
```

Stream.onStatus

Availability
Flash Communication Server MX 1.0.

Usage
myStream.onStatus = function([infoObject]) {}

Parameters
infoObject An optional parameter that is defined according to the status message. For more information about this parameter, see “Server-Side Information Objects” on page 229.

Returns
Nothing.

Description
Event handler; invoked every time the status of a Stream object changes. For example, if you play a file in a stream, Stream.onStatus is invoked. Use Stream.onStatus to check when play starts and ends, when recording starts, and so on.
Example
The following example defines a function that executes whenever the Stream.onStatus event is invoked:
```javascript
s = Stream.get("foo");
s.onStatus = function(info){
    // Insert code here.
};
```

Stream.play()

Availability
Flash Communication Server MX 1.0.

Usage
`myStream.play(streamName, [startTime, length, reset, remoteConnection, virtualKey])`

Parameters
- **streamName**: The name of any published live stream, recorded stream (FLV file), or MP3 file.
  
  To play FLV files, the default Flash file format for recorded streams, specify the name of the stream (for example, "myVideo"). To play MP3 files that you've stored on the server, or the ID3 tags of MP3 files, you must precede the stream name with `mp3:` or `id3:` (for example, "mp3:bolero" or "id3:bolero"). For more information on playing MP3 files, see *Developing Media Applications*.

- **startTime**: The start time of the stream playback, in seconds. If no value is specified, it is set to -2. If `startTime` is -2, the server tries to play a live stream with the name specified in `streamName`. If no live stream is available, the server tries to play a recorded stream with the name specified in `streamName`. If no recorded stream is found, the server creates a live stream with the name specified in `streamName` and waits for someone to publish to that stream. If `startTime` is -1, the server attempts to play a live stream with the name specified in `streamName` and waits for a publisher if no specified live stream is available. If `startTime` is greater than or equal to 0, the server plays the recorded stream with the name specified in `streamName`, starting from the time given. If no recorded stream is found, the `play` method is ignored. If a negative value other than -1 is specified, the server interprets it as -2. This is an optional parameter.
length  The length of play, in seconds. For a live stream, a value of -1 plays the stream as long as the stream exists. Any positive value plays the stream for the corresponding number of seconds. For a recorded stream, a value of -1 plays the entire file, and a value of 0 returns the first video frame. Any positive number plays the stream for the corresponding number of seconds. By default, the value is -1. This is an optional parameter.

reset  A Boolean value, or number, that flushes the playing stream. If reset is false (0), the server maintains a playlist, and each call to Stream.play is appended to the end of the playlist so that the next play does not start until the previous play finishes. You can use this technique to create a dynamic playlist. If reset is true (1), any playing stream stops, and the playlist is reset. By default, the value is true.

You can also specify a number value of 2 or 3 for the reset parameter, which is useful when playing recorded stream files that contain message data. These values are analogous to false (0) and true (1), respectively: a value of 2 maintains a playlist, and a value of 3 resets the playlist. However, the difference is that specifying either 2 or 3 for reset returns all messages in the specified recorded stream at once, rather than at the intervals which the messages were originally recorded (the default behavior).


remoteConnection  A reference to a NetConnection object that is used to connect to a remote server. The requested stream plays from the remote server if this parameter is provided. This parameter is optional.

virtualKey  A string indicating a key value. Starting with Flash Media Server 2, stream names are not always unique; you can create multiple streams with the same name, place them in different physical directories, and use the VirtualDirectory section and VirtualKeys section of the vhost.xml file to direct clients to the appropriate stream. Because the Stream.length() method is not associated with a client, but connects to a stream on the server, you may need to specify a virtual key to identify the correct stream. For more information about keys, see Client.virtualKey. This parameter is optional.

Returns
A Boolean value: true if the Stream.play call is accepted by the server and added to the playlist; false otherwise. The Stream.play method can fail if the server fails to find the stream or if an error occurs. To get information about the Stream.play call, you can define a Stream.onStatus handler to catch the play status or error.

If the streamName parameter is false, the stream stops playing. A Boolean value of true is returned if the stop succeeds; false otherwise.
Description
Method; controls the data source of a stream with an optional start time, duration, and reset flag to flush any previously playing stream. The Stream.play() method also has a parameter that lets you reference a NetConnection object to play a stream from another server. The Stream.play() method allows you to do the following:

- Chain streams between servers.
- Create a hub to switch between live streams and recorded streams.
- Combine different steams into a recorded stream.

You can combine multiple streams to create a playlist for clients. The server-side Stream.play() method behaves a bit differently than the NetStream.play method on the client side. A play call on the server is similar to a publish call on the client. It controls the source of the data that is coming into a stream. When you call Stream.play on the server, the server becomes the publisher. Because the server has a higher priority than the client, the client is forced to unpublish from the stream if the server calls a play method on the same stream.

In general, if any recorded streams are included in a server playlist, you cannot play the server stream as a live stream.

NOTE A stream that plays from a remote server by means of the NetConnection object is a live stream.

If you require a value to begin a stream, you may need to change the Application.xml file's “Enhanced seeking” flag at the server. “Enhanced seeking” is a Boolean flag in the Application.xml file. By default, this flag is set to false. When a play occurs, the server seeks to the closest video keyframe possible and starts from that keyframe. For example, if you want to play at time 15, and there are keyframes only at time 11 and time 17, seeking will start from time 17 instead of time 15. This is an approximate seeking method that works well with compressed streams.

If the flag is set to true, some compression is invoked on the server. Using the previous example, if the flag is set to true, the server creates a keyframe—based on the preexisting keyframe at time 11—for each keyframe from 11 through 15. Even though a keyframe does not exist at the seek time, the server generates a keyframe, which involves some processing time on the server.
Example

The following example shows how streams can be chained between servers:

```actionscript
application.myRemoteConn = new NetConnection();
application.myRemoteConn.onStatus = function(info){
  trace("Connection to remote server status " + info.code + ";
  // Tell all the clients.
  for (var i = 0; i < application.clients.length; i++){
    application.clients[i].call("onServerStatus", null,
    info.code, info.description);
  }
};
// Use the NetConnection object to connect to a remote server.
application.myRemoteConn.connect(rtmp://movie.com/movieApp);
// Setup the server stream.
application.myStream = Stream.get("foo");
if (application.myStream){
  application.myStream.play("Movie1", 0, -1, true,
  application.myRemoteConn);
}

The following example shows `Stream.play` is used as a hub to switch between live streams and recorded streams:

```actionscript```
// Set up the server stream.
application.myStream = Stream.get("foo");
if (application.myStream){
  // This server stream plays "Live1", "Record1", and "Live2" for 5 seconds each.
  application.myStream.play("Live1", -1, 5);
  application.myStream.play("Record1", 0, 5, false);
  application.myStream.play("Live2", -1, 5, false);
}
```

The following example combines different streams into a recorded stream:

```actionscript```
// Set up the server stream.
application.myStream = Stream.get("foo");
if (application.myStream){
  // Like the previous example, this server stream
  // plays "Live1", "Record1", and "Live2"
  // for 5 seconds each. But this time,
  // all the data will be recorded to a recorded stream "foo".
  application.myStream.record();
  application.myStream.play("Live1", -1, 5);
  application.myStream.play("Record1", 0, 5, false);
  application.myStream.play("Live2", -1, 5, false);
}
```

The following example uses `Stream.play` to stop playing the stream foo:

```actionscript```
application.myStream.play(false);
```

The following example shows how streams can be chained between servers:

```actionscript```
application.myRemoteConn = new NetConnection();
application.myRemoteConn.onStatus = function(info){
  trace("Connection to remote server status " + info.code + ";
  // Tell all the clients.
  for (var i = 0; i < application.clients.length; i++){
    application.clients[i].call("onServerStatus", null,
    info.code, info.description);
  }
};
// Use the NetConnection object to connect to a remote server.
application.myRemoteConn.connect(rtmp://movie.com/movieApp);
// Setup the server stream.
application.myStream = Stream.get("foo");
if (application.myStream){
  application.myStream.play("Movie1", 0, -1, true,
  application.myRemoteConn);
}

The following example shows `Stream.play` is used as a hub to switch between live streams and recorded streams:

```actionscript```
// Set up the server stream.
application.myStream = Stream.get("foo");
if (application.myStream){
  // This server stream plays "Live1", "Record1", and "Live2" for 5 seconds each.
  application.myStream.play("Live1", -1, 5);
  application.myStream.play("Record1", 0, 5, false);
  application.myStream.play("Live2", -1, 5, false);
}
```

The following example combines different streams into a recorded stream:

```actionscript```
// Set up the server stream.
application.myStream = Stream.get("foo");
if (application.myStream){
  // Like the previous example, this server stream
  // plays "Live1", "Record1", and "Live2"
  // for 5 seconds each. But this time,
  // all the data will be recorded to a recorded stream "foo".
  application.myStream.record();
  application.myStream.play("Live1", -1, 5);
  application.myStream.play("Record1", 0, 5, false);
  application.myStream.play("Live2", -1, 5, false);
}
```

The following example uses `Stream.play` to stop playing the stream foo:

```actionscript```
application.myStream.play(false);
```

154 Server-Side ActionScript Language Reference
The following example creates a playlist of three MP3 files (beethoven.mp3, mozart.mp3, and chopin.mp3) and plays each file in turn over the live stream foo:

```javascript
application.myStream = Stream.get("foo");
if(application.myStream) {
  application.myStream.play("mp3:beethoven", 0);
  application.myStream.play("mp3:mozart", 0, false);
  application.myStream.play("mp3:chopin.mp3", 0, false);
}
```

In the following example, data messages in the recorded stream file log.flv are returned at the intervals which they were originally recorded.

```javascript
application.myStream = Stream.get("data");
if (application.myStream) {
  application.myStream.play("log", 0, -1);
}
```

In the following example, data messages in the recorded stream file log.flv are returned all at once, rather than at the intervals which they were originally recorded.

```javascript
application.myStream = Stream.get("data");
if (application.myStream) {
  application.myStream.play("log", 0, -1, 2);
}
```

**Stream.record()**

**Availability**
Flash Communication Server MX 1.0.

**Usage**

```javascript
myStream.record(flag)
```

**Parameters**

`flag` This parameter can have the value `record`, `append`, or `false`. If the value is `record`, the data file is overwritten if it exists. If the value is `append`, the incoming data is appended to the end of the existing file. If the value is `false`, any previous recording stops. By default, the value is `record`.

**Returns**
A Boolean value of `true` if the recording succeeds; `false` otherwise.

**Description**
Method; records all the data going through a Stream object.
Example
The following example opens a stream `s` and, when it is open, plays `sample` and records it. Because no value is passed to the `record` method, the default value, `record`, is passed.

```javascript
// Start recording.
s = Stream.get("foo");
if (s){
    s.play("sample");
    s.record();
}
// Stop recording.
s = Stream.get("foo");
if (s){
    s.record(false);
}
```

Stream.send()

Availability
Flash Communication Server MX 1.0.

Usage
```javascript
myStream.send(handlerName, [p1, .... pN])
```

Parameters
- `handlerName` Calls the specified handler in client-side ActionScript code. The `handlerName` value is the name of a method relative to the subscribing Stream object. For example, if `handlerName` is `doSomething`, the `doSomething` method at the stream level is invoked with all the `p1, .... pN` parameters. Unlike the method names in `Client.call()` and `NetConnection.call()`, the handler name can be only one level deep (that is, it cannot have the form of `object/method`).

- `p1, .... pN` Parameters of any ActionScript type, including references to other ActionScript objects. These parameters are passed to the specified handler when it is executed on the Flash client.

Returns
A Boolean value of `true` if the message was sent to the client; `false` otherwise.

**NOTE** Do not use a built-in method name for a handler name. For example, the subscribing stream will close if the handler name is `close`.


Description
Method; sends a message to all clients subscribing to the stream and the message is processed by the handler specified on the client. Because the server has higher priority than the clients, the server can still send a message on a stream owned by a client. Unlike the Stream.play() method, the server does not need to take ownership of a stream from the client in order to send a message. After send() is called, the client still owns the stream as a publisher.

Example
The following example calls the method Test on the client-side Stream object and sends it the string "hello world":
application.streams["foo"].send("Test", "hello world");
The following example is the client-side ActionScript that receives the Stream.send call. The method Test is defined on the Stream object:
tStream.Test = function(str) {
    // Insert code to process the str object.
}

Stream.setBufferTime()

Availability
Flash Communication Server MX 1.0.

Usage
myStream.setBufferTime()

Description
Method; increases the message queue length. When you play a stream from a remote server, the Stream.setBufferTime method sends a message to the remote server that adjusts the length of the message queue. The default length of the message queue is 2 seconds. You should set the buffer time higher when playing a high-quality recorded stream over a low-bandwidth network.

When a user clicks a seek button in an application, buffered packets are sent to the server. The buffered seeking in a Flash Media Server application occurs on the server; the Flash Media Server doesn't support client-side buffering. The seek time can be smaller or larger than the buffer size, and it has no direct relationship to the buffer size. Every time the server receives a seek request from Flash Player, it clears the message queue on the server. The server tries to seek to the desired position and starts filling the queue again. At the same time, Flash Player also clears its own buffer after a seek request, and the buffer is eventually filled after the server starts sending the new messages.
Stream.setVirtualPath()

**Availability**
Flash Media Server 2.

**Usage**

```javascript
myStream.setVirtualPath(virtualPath, directory, virtualKey)
```

**Parameters**

- `virtualPath` A string indicating the virtual directory path of a stream. If the stream is not located in the virtual path, the default virtual directory path is searched.
- `directory` A string indicating the physical directory in which to store streams.
- `virtualKey` A string that sets or removes the key value for each virtual path entry.

**Returns**

Nothing.

**Description**

Method; sets or removes the virtual directory path for video stream playback.

Flash Media Server can deliver two video codecs to clients: Sorenson Spark and the On2 VP6 codec. Flash Player 8 supports both codecs; Flash Player 7 and earlier supports only Sorenson Spark. You can use the `Stream.setVirtualPath()` method to dynamically serve Flash Player 8 clients a better version of a video stream.

When Flash Player requests a stream from Flash Media Server, the Flash Player version determines the server-side `Client.virtualKey` property (typically a mapping of the Flash Player version). Flash Media Server then serves the client a stream from whichever virtual directory the virtual key is mapped to. For example, if the client is Flash Player 8, the virtual key might map to the On2 streams directory, and the client would be served a higher quality stream. If the client is Flash Player 7, the virtual key might map to the default stream directory. This feature allows you to serve the same content to all clients in the optimum encoding for each client's platform.

The mapping between the virtual keys and virtual directories is made in the vhost.xml file. (For more information about the vhost.xml file, see “The Vhost.xml file” in *Managing Flash Media Server*). When you call `setVirtualPath()`, you are changing values in the `VirtualDirectory` tag of that file.
The following example shows the VirtualKeys section from the vhost.xml file in which the virtual keys can be mapped to a range of Flash Player versions:

```xml
<VirtualKeys>
  <Key from="WIN 7,0,19,0" to="WIN 9,0,0,0"></Key>
  <Key from="WIN 6,0,0,0" to="WIN 7,0,18,0"></Key>
  <Key from="MAC 6,0,0,0" to="MAC 7,0,55,0"></Key>
</VirtualKeys>
```

By default, there are no values in the key tags. To implement this feature, you must add the key values, either directly in the vhost.xml file, or with the `setVirtualPath()` method. For example, to implement this feature, set the first key to A, and the second two keys to B, as shown in the following example:

```xml
<VirtualKeys>
  <Key from="WIN 7,0,19,0" to="WIN 9,0,0,0">A</Key>
  <Key from="WIN 6,0,0,0" to="WIN 7,0,18,0">B</Key>
  <Key from="MAC 6,0,0,0" to="MAC 7,0,55,0">B</Key>
</VirtualKeys>
```

The following example shows what the VirtualDirectory section from the vhost.xml file could look like (the key values and Streams values are not in the file by default). The virtual keys are mapped to a virtual path and a physical directory, which are separated by a semi-colon (for example, foo;c:\streams). To set up several virtual directories for different Flash Player versions, use the same virtual path with different physical directories for each Streams tag, as shown in the following example:

```xml
<VirtualDirectory>
  <!-- Specifies virtual directory mapping for recorded streams. -->
  <!-- To specify multiple virtual directory mappings for a stream, -->
  <!-- add additional <Streams> tags; one for each virtual dir -->
  <!-- mapping. Syntax for virtual directories is as follows: -->
  <!-- <Streams>foo;c:\data</Streams>. This maps all streams whose -->
  <!-- names begin with "foo/" to the physical directory c:\data. -->
  <!-- For example, the stream named "foo/bar" would map to the -->
  <!-- physical file "c:\data\bar.flv". Similarly, if you had a -->
  <!-- stream named "foo/bar/x", then we first try to find a vdir -->
  <!-- mapping for "foo/bar". Failing to do so, we then check for -->
  <!-- a vdir mapping for "foo". Since there is one, the stream -->
  <!-- "foo/bar" corresponds to the file "c:\data\bar\x.flv". -->
  <Streams key="A">foo;c:\streams\on2</Streams>
  <Streams key="B">foo;c:\streams\sorenson</Streams>
  <Streams key="">foo;c:\streams</Streams>
</VirtualDirectory>
```

NOTE: You can also change the values of the VirtualKeys and VirtualDirectory tags directly in the vhost.xml file. This is the most common usage scenario. For more information about the vhost.xml file, see "The Vhost.xml file" in Managing Flash Media Server.
Example
The following code sets the virtual key to B, the virtual path to /foo, and the physical directory to c:\streams\on2:

```ActionScript
myStream.setVirtualPath("B", "/foo", "c:\streams\on2");
```

Stream.size()

Availability
Flash Media Server 2.

Usage

```ActionScript
Stream.size(name[, virtualKey])
```

Parameters

- **name** A string indicating the name of a stream. You can use the format tag in the name parameter to specify the type.
- **virtualKey** A string indicating a key value. Starting with Flash Media Server 2, stream names are not always unique; you can create multiple streams with the same name, place them in different physical directories, and use the VirtualDirectory section and VirtualKeys section of the vhost.xml file to direct clients to the appropriate stream. Because the Stream.size() method is not associated with a client, but connects to a stream on the server, you may need to specify a virtual key to identify the correct stream. For more information about keys, see Client.virtualKey. This parameter is optional.

Returns
A number indicating the size of the stream; if the requested stream is not found, returns 0.

Description
Method (static); returns the size of a recorded stream in bytes.

Example
The following examples return the size of a stream and an MP3 stream, respectively:

```ActionScript
function onProcessCmd(cmd){
    // Insert code here...
    var streamSize = Stream.size("foo");
    trace("Size: " + streamSize + "\n");
}

//For mp3
```

160 Server-Side ActionScript Language Reference
function onProcessCmd(cmd){
    // Insert code here...
    var streamSize = Stream.size("mp3:foo");
    trace("Size: "+streamSize+"\n");
}

Stream.syncWrite

Availability
Flash Media Server 2.

Usage
myStream.syncWrite

Description
Property; a Boolean value that controls when a stream writes the contents of the buffer to a FLV file when the stream is recording. When syncWrite is true, all the messages that pass through the stream are flushed to the FLV file immediately. It is highly recommended that user should only set syncWrite to true in a stream that contains only data. Synchronization problems might occur when syncWrite is set to true in a stream that contains data and audio, video, or both.

Example
The following example flushes data immediately to the FLV file:

// Assume foo is a data-only stream
application.myStream = Stream.get("foo");

if (application.myStream){
    application.myStream.syncWrite = true;
    application.myStream.record();
    application.myStream.send("test", "hello world");
}

trace()

Availability
Flash Communication Server MX 1.0.

Usage
trace(expression)
Parameters

expression Any valid ActionScript expression.

Returns
Nothing.

Description
Method (global); evaluates an expression and displays the value. The expression appears in the Console window when the server is running in console mode; the expression is also published to the logs/application appname stream, which is available in the management console or in the Application inspector.

The values in the trace expression are converted to strings if possible before they are output. You can use the trace() function to debug a script, record programming notes, or to display messages while testing a file. The trace() function is similar to the alert function in JavaScript.

WebService class

Availability
Flash Media Server 2.

Description
You can use the WebService class to create and access a WSDL/SOAP web service. There are several classes that comprise the Flash Media Server web services feature: WebService class, SOAPFault class, SOAPCall class, and Log class.

NOTE
The WebService class is not able to retrieve complex data or an array returned by a web service. Also, the WebService class does not support security features.

The following steps outline the process of creating and accessing a web service.

To create and access a web service:
1. Prepare the WSDL location:
   var wsdlURI = "http://www.flash-db.com/services/ws/companyInfo.wsdl";
2. Instantiate the web service object using the WSDL location:
   stockService = new WebService(wsdlURI);
3. (Optional) Handle the WSDL parsing and web service instantiation event through the `WebService.onLoad` handler:

```javascript
// Handle the WSDL loading event.
stockService.onLoad = function(wsdl){
    wsdlField.text = wsdl;
}
```

4. (Optional) Handle the fault if the WSDL doesn’t load:

```javascript
// If the WSDL fails to load the onFault event is fired.
stockService.onFault = function(fault){
    wsdlField.text = fault.faultstring;
}
```

5. (Optional) Set the SOAP headers:

```javascript
// If headers are required they are added as follows:
var myHeader = new XML(headerSource);
stockService.addHeader(myHeader);
```

6. Invoke a web service operation:

```javascript
// Method invocations return an asynchronous callback.
callback = stockService.doCompanyInfo("anyuser", "anypassword", "MACR");
// NOTE: callback is undefined if the service itself is not created
// (and service.onFault is also invoked).
```

7. Handle either the output or the error fault returned from the invocation:

```javascript
// Handle a successful result.
callback.onResult = function(result){
    stock.companyInfo = result;
}
// Handle an error result.
callback.onFault = function(fault){
    stock.companyInfo = fault.faultstring;
}
```

---

**Event handler summary for the WebService class**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>WebService.onFault</code></td>
<td>Invoked when an error occurs during WSDL parsing.</td>
</tr>
<tr>
<td><code>WebService.onLoad</code></td>
<td>Invoked when the web service has successfully loaded and parsed its WSDL file.</td>
</tr>
</tbody>
</table>

---
Constructor for the WebService class

**Availability**
Flash Media Server 2.

**Usage**

```actionscript
new WebService(wsdlURI)
```

**Parameters**

- `wsdlURL` A string specifying the URL of a WSDL URL.

**Returns**

A WebService object.

**Description**

Constructor; creates a new WebService object. You must use the constructor to create a WebService object before you call any of the WebService class methods.

**Example**

The following example prepares the WSDL location and passes it to the WebService constructor to create a new WebService object, `stockService`:

```actionscript
var wsdlURI = "http://www.flash-db.com/services/ws/companyInfo.wsdl";
stockService = new WebService(wsdlURI);
```

---

**WebService.onFault**

**Availability**
Flash Media Server 2.

**Usage**

```actionscript
myWS.onFault(fault){}
```

**Parameters**

- `fault` An object version of an XML SOAP fault (see “SOAPFault class” on page 142).

**Returns**

**Description**

Event handler; called when an error occurs during WSDL parsing. The web services features convert parsing and network problems into SOAP faults for simple handling.
Example
The following example displays the fault code in a text field if the WSDL fails to load and the onFault event fires:
```javascript
// Prepare the WSDL location:
var wsdlURI = "http://www.flash-db.com/services/ws/companyInfo.wsdl";

// Instantiate the web service object using the WSDL location:
stockService = new WebService(wsdlURI);

// Handle the WSDL parsing and web service instantiation event.
stockService.onLoad = function(wsdl){
    wsdlField.text = wsdl;
}

// If the wsdl fails to load the onFault event is fired:
stockService.onFault = function(fault){
    wsdlField.text = fault.faultstring;
}
```

WebService.onLoad

Availability
Flash Media Server 2.

Usage
myWS.onLoad(wsdlDocument)

Parameters

Returns
Nothing.

Description
Event handler; called when the web service has successfully loaded and parsed its WSDL file. Operations can be invoked in an application before this event occurs; when this happens, they are queued internally and not actually transmitted until the WSDL has loaded.
Example

In the following example, the onLoad event is used to handle the WSDL parsing:

```actionscript
// Prepare the WSDL location:
var wsdlURI = "http://www.flash-db.com/services/ws/companyInfo.wsdl":

// Instantiate the web service object using the WSDL location:
stockService = new WebService(wsdlURI);

// Handle the WSDL parsing and web service instantiation event.
stockService.onLoad = function(wsdl){
    wsdlField.text = wsdl;
}
```

XML class

Availability

Flash Media Server 2.

Description

The XML class lets you load, parse, send, build, and manipulate XML document trees. You must use the constructor `new XML()` to create an XML object before calling any method of the XML class. An XML document is represented in Flash by the XML class. Each element of the hierarchical document is represented by an XMLNode object.

NOTE

The XML and XMLNode objects are modeled after the W3C DOM Level 1 Recommendation: That recommendation specifies a Node interface and a Document interface. The Document interface inherits from the Node interface, and adds methods such as `createElement()` and `createTextNode()`. In ActionScript, the XML and XMLNode objects are designed to divide functionality along similar lines.
## Property summary for the XML class

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>XML.attributes</code></td>
<td>An object that contains all the attributes of the specified XML object.</td>
</tr>
<tr>
<td><code>XML.childNodes</code></td>
<td>Read-only; an array of the specified XML object’s children. Each element in the array is a reference to an XML object that represents a child node.</td>
</tr>
<tr>
<td><code>XML.contentType</code></td>
<td>The MIME type transmitted to the server.</td>
</tr>
<tr>
<td><code>XML.docTypeDecl</code></td>
<td>Sets and returns information about an XML document’s DOCTYPE declaration.</td>
</tr>
<tr>
<td><code>XML.firstChild</code></td>
<td>Read-only; references the first child in the list for the specified node.</td>
</tr>
<tr>
<td><code>XML.ignoreWhite</code></td>
<td>When set to true, discards, during the parsing process, text nodes that contain only white space.</td>
</tr>
<tr>
<td><code>XML.lastChild</code></td>
<td>Read-only; references the last child in the list for the specified node.</td>
</tr>
<tr>
<td><code>XML.loaded</code></td>
<td>Checks whether the specified XML object has loaded.</td>
</tr>
<tr>
<td><code>XML.localName</code></td>
<td>Read-only; the local name portion of the XML node’s name.</td>
</tr>
<tr>
<td><code>XML.namespaceURI</code></td>
<td>Read-only; if the XML node has a prefix, namespaceURI is the value of the xmlns declaration for that prefix (the URI), which is typically called the namespace URI.</td>
</tr>
<tr>
<td><code>XML.nextSibling</code></td>
<td>Read-only; references the next sibling in the parent node’s child list.</td>
</tr>
<tr>
<td><code>XML.nodeName</code></td>
<td>The node name of an XML object.</td>
</tr>
<tr>
<td><code>XML.nodeType</code></td>
<td>Read-only; the type of the specified node (XML element or text node).</td>
</tr>
<tr>
<td><code>XML.nodeValue</code></td>
<td>The text of the specified node if the node is a text node.</td>
</tr>
<tr>
<td><code>XML.parentNode</code></td>
<td>Read-only; references the parent node of the specified node.</td>
</tr>
<tr>
<td><code>XML.prefix</code></td>
<td>Read-only; the prefix portion of the XML node name.</td>
</tr>
<tr>
<td><code>XML.previousSibling</code></td>
<td>Read-only; references the previous sibling in the parent node’s child list.</td>
</tr>
<tr>
<td><code>XML.status</code></td>
<td>A numeric status code that indicates the success or failure of an XML document parsing operation.</td>
</tr>
<tr>
<td><code>XML.xmlDecl</code></td>
<td>Specifies information about a document’s XML declaration.</td>
</tr>
</tbody>
</table>
### Method summary for the XML class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>XML.addRequestHeader()</code></td>
<td>Adds or changes HTTP headers for POST operations.</td>
</tr>
<tr>
<td><code>XML.appendChild()</code></td>
<td>Appends a node to the end of the specified object’s child list.</td>
</tr>
<tr>
<td><code>XML.cloneNode()</code></td>
<td>Clones the specified node, and optionally, recursively clones all children.</td>
</tr>
<tr>
<td><code>XML.createElement()</code></td>
<td>Creates a new XML element.</td>
</tr>
<tr>
<td><code>XML.createTextNode()</code></td>
<td>Creates a new XML text node.</td>
</tr>
<tr>
<td><code>XML.getBytesLoaded()</code></td>
<td>Returns the number of bytes loaded for the specified XML document.</td>
</tr>
<tr>
<td><code>XML.getBytesTotal()</code></td>
<td>Returns the size of the XML document, in bytes.</td>
</tr>
<tr>
<td><code>XML.getNamespaceForPrefix()</code></td>
<td>Returns the namespace URI that is associated with the specified prefix for the node.</td>
</tr>
<tr>
<td><code>XML.getPrefixForNamespace()</code></td>
<td>Returns the prefix that is associated with the specified namespace URI for the node.</td>
</tr>
<tr>
<td><code>XML.hasChildNodes()</code></td>
<td>Returns true if the specified node has child nodes; otherwise, returns false.</td>
</tr>
<tr>
<td><code>XML.insertBefore()</code></td>
<td>Inserts a node in front of an existing node in the specified node’s child list.</td>
</tr>
<tr>
<td><code>XML.load()</code></td>
<td>Loads a document (specified by the XML object) from a URL.</td>
</tr>
<tr>
<td><code>XML.parseXML()</code></td>
<td>Parses an XML document into the specified XML object tree.</td>
</tr>
<tr>
<td><code>XML.removeNode()</code></td>
<td>Removes the specified node from its parent.</td>
</tr>
<tr>
<td><code>XML.send()</code></td>
<td>Sends the specified XML object to a URL.</td>
</tr>
<tr>
<td><code>XML.sendAndLoad()</code></td>
<td>Sends the specified XML object to a URL, and loads the server response into another XML object.</td>
</tr>
<tr>
<td><code>XML.toString()</code></td>
<td>Converts the specified node and any children to XML text.</td>
</tr>
</tbody>
</table>
Event handler summary for the XML class

<table>
<thead>
<tr>
<th>Event handler</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML.onData</td>
<td>Invoked when XML text has been completely downloaded from the server or when an error occurs downloading XML text from a server.</td>
</tr>
<tr>
<td>XML.onHTTPStatus</td>
<td>Invoked when Flash Media Server receives an HTTP status code from the server. This handler lets you capture and act on HTTP status codes.</td>
</tr>
<tr>
<td>XML.onLoad</td>
<td>Returns a Boolean value indicating whether the XML object was successfully loaded with XML.load() or XML.sendAndLoad().</td>
</tr>
</tbody>
</table>

Collections summary for the XML class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML.attributes</td>
<td>Returns an associative array that contains all the attributes of the specified node.</td>
</tr>
<tr>
<td>XML.childNodes</td>
<td>Read-only; returns an array that contains references to the child nodes of the specified node.</td>
</tr>
</tbody>
</table>

Constructor for the XML class

**Availability**
Flash Media Server 2.

**Usage**
```javascript
new XML([source])
```

**Parameters**
- `source`: A string; the XML text parsed to create the new XML object.

**Returns**
A reference to an XML object.
Description
Constructor; creates a new XML object. You must use the constructor to create an XML
object before you call any of the XML class methods.

Example
The following example creates a new, empty XML object:
```actionscript
var my_xml = new XML();
```
The following example creates an XML object by parsing the XML text specified in the
`source` parameter and populates the newly created XML object with the resulting XML
document tree:
```actionscript
var other_xml = new XML("<state name="California">
  <city>San Francisco</city>
</state>");
```

See also
XML.createElement(), XML.createTextNode()

XML.addRequestHeader()

Availability
Flash Media Server 2.

Usage
```actionscript
my_xml.addRequestHeader(headerName, headerValue)
my_xml.addRequestHeader(["headerName_1": "headerValue_1" ...
  "headerName_n": "headerValue_n"])
```

Parameters
- `headerName` A string that represents an HTTP request header name.
- `headerValue` A string that represents the value associated with `headerName`.

Returns
Nothing.

NOTE
Use the `createElement()` and `createTextNode()` methods to add elements and text
nodes to an XML document tree.
Description
Method: adds or changes HTTP request headers (such as Content-Type or SOAPAction) sent with POST actions. In the first usage, you pass two strings, headerName and headerValue, to the method. In the second usage, you pass an array of strings, alternating header names, and header values.

If multiple calls are made to set the same header name, each successive value replaces the value set in the previous call.

You cannot add or change the following standard HTTP headers using this method:

Example
The following example adds a custom HTTP header named SOAPAction with a value of Foo to an XML object named my_xml:
my_xml.addRequestHeader("SOAPAction", "'Foo'");

The following example creates an array named headers that contains two alternating HTTP headers and their associated values. The array is passed as a parameter to the addRequestHeader() method.
var headers = new Array("Content-Type", "text/plain", "X-ClientAppVersion", "2.0");
my_xml.addRequestHeader(headers);

XML.appendChild()

Availability
Flash Media Server 2.

Usage
my_xml.appendChild(childNode)

Parameters

childNode An XMLNode object that represents the node to be moved from its current location to the child list of the my_xml object.

Returns
Nothing.
Description
Method: appends the specified node to the XML object’s child list. This method operates directly on the node referenced by the childNode parameter; it does not append a copy of the node. If the node to be appended already exists in another tree structure, appending the node to the new location will remove it from its current location. If the childNode parameter refers to a node that already exists in another XML tree structure, the appended child node is placed in the new tree structure after it is removed from its existing parent node.

Example
The following example performs the actions in this list:
1. Creates two empty XML documents, doc1 and doc2.
2. Creates a new node, using the createElement() method, and appends it, using the appendChild() method, to the XML document named doc1.
3. Shows how to move a node using the appendChild() method, by moving the root node from doc1 to doc2.
4. Clones the root node from doc2 and appends it to doc1.
5. Creates a new node and appends it to the root node of the XML document doc1.

```javascript
var doc1 = new XML();
var doc2 = new XML();

// Create a root node and add it to doc1.
var rootnode = doc1.createElement("root");
doc1.appendChild(rootnode);
trace("doc1: " + doc1); // output: doc1: <root />
trace("doc2: " + doc2); // output: doc2:

// Move the root node to doc2.
doc2.appendChild(rootnode);
trace("doc1: " + doc1); // output: doc1: <root />
trace("doc2: " + doc2); // output: doc2: <root />

// Clone the root node and append it to doc1.
var clone = doc2.firstChild.cloneNode(true);
doc1.appendChild(clone);
trace("doc1: " + doc1); // output: doc1: <root />
trace("doc2: " + doc2); // output: doc2: <root />

// Create a new node to append to root node (named clone) of doc1.
var newbie = doc1.createElement("newbie");
clone.appendChild(newbie);
trace("doc1: " + doc1); // output: doc1: <root><newbie /></root>
```

Note
In Flash Media Server, the output of trace() statements appears in the application log file and Application inspector.

172 Server-Side ActionScript Language Reference
XML.attributes

Availability
Flash Media Server 2.

Usage
my_xml.attributes

Description
Property; an object that contains all the attributes of the specified XML object. Associative arrays use keys as indexes, not ordinal integer indexes used by regular arrays. In the XML.attributes associative array, the key index is a string representing the name of the attribute. The value associated with that key index is the string value associated with that attribute. For example, if you have an attribute named color, you would retrieve that attribute's value by using the color as the key index, as shown in the following code:

```javascript
var myColor = doc.firstChild.attributes.color
```

Example
The following example shows the XML attribute names:

```javascript
// Create a tag called 'mytag' with
// an attribute called 'name' with value 'Val'.
var doc = new XML("<mytag name="Val"> item </mytag>"SCREENCODE_START

// Assign the value of the 'name' attribute to variable y.
var y = doc.firstChild.attributes.name;
trace(y);  // output: Val

// Create a new attribute named 'order' with value 'first'.
doc.firstChild.attributes.order = "first";

// Assign the value of the 'order' attribute to variable z.
var z = doc.firstChild.attributes.order
trace(z);  // output: first
```

NOTE
In Flash Media Server, the output of trace() statements appears in the application log file and Application inspector.
XML.childNodes

Availability
Flash Media Server 2.

Usage
my_xml.childNodes

Description
Property (read-only); an array of the specified XML object’s children. Each element in the array is a reference to an XML object that represents a child node. This is a read-only property and cannot be used to manipulate child nodes. Use the XML.appendChild(), XML.insertBefore(), and XML.removeNode() methods to manipulate child nodes.

This property is undefined for text nodes (nodeType == 3).

Example
The following example shows how to use the XML.childNodes property to return an array of child nodes:

```actionscript
// Create a new XML document.
var doc = new XML();

// Create a root node.
var rootNode = doc.createElement("rootNode");

// Create three child nodes.
var oldest = doc.createElement("oldest");
var middle = doc.createElement("middle");
var youngest = doc.createElement("youngest");

// Add the rootNode as the root of the XML document tree.
doc.appendChild(rootNode);

// Add each of the child nodes as children of rootNode.
rootNode.appendChild(oldest);
rootNode.appendChild(middle);
rootNode.appendChild(youngest);

// Create an array and use rootNode to populate it.
var firstArray:Array = doc.childNodes;
trace (firstArray);
// Output: <rootNode><oldest /><middle /><youngest /></rootNode>
```
// Create another array and use the child nodes to populate it.
var secondArray = rootNode.childNodes;
trace(secondArray);
// Output: <oldest />,<middle />,<youngest />
Example
The following example shows how to use the XML.cloneNode() method to create a copy of a node:

```actionscript
// Create a new XML document.
var doc = new XML();

// Create a root node.
var rootNode = doc.createElement("rootNode");

// Create three child nodes.
var oldest = doc.createElement("oldest");
var middle = doc.createElement("middle");
var youngest = doc.createElement("youngest");

// Add the rootNode as the root of the XML document tree.
doc.appendChild(rootNode);

// Add each of the child nodes as children of rootNode.
rootNode.appendChild(oldest);
rootNode.appendChild(middle);
rootNode.appendChild(youngest);

// Create a copy of the middle node using cloneNode().
var middle2 = middle.cloneNode(false);

// Insert the clone node into rootNode between
// the middle and youngest nodes.
rootNode.insertBefore(middle2, youngest);
trace(rootNode);
// Output (with line breaks added):
// <rootNode>
//   <oldest/>
//   <middle/>
//   <middle/>
//   <youngest/>
// </rootNode>

// Create a copy of rootNode using cloneNode() to demonstrate a deep copy.
var rootClone = rootNode.cloneNode(true);

// Insert the clone, which contains all child nodes, to rootNode.
rootNode.appendChild(rootClone);
trace(rootNode);
// Output (with line breaks added):
// <rootNode>
//   <oldest/>
//   <middle/>
//   <middle/>
//   <youngest/>
// </rootNode>
```
// <rootNode>
//   <oldest/>
//   <middle/>
//   <middle/>
//   <youngest/>
// </rootNode>

XML.contentType

Availability
Flash Media Server 2.

Usage
my_xml.contentType

Description
Property; the MIME content type that is sent to the server when you call the XML.send() or XML.sendAndLoad() method. The default is application/x-www-form-urlencoded, which is the standard MIME content type used for most HTML forms.

Example
The following example creates a new XML document and checks its default content type:

    // Create a new XML document.
    var doc = new XML();

    // Trace the default content type.
    trace(doc.contentType);

    // output: application/x-www-form-urlencoded

See also
XML.send(), XML.sendAndLoad()
XML.createElement()  

**Availability**  
Flash Media Server 2.

**Usage**  
my_xml.createElement(name)

**Parameters**  
name The tag name of the XML element being created.

**Returns**  
An XML node; an XML element.

**Description**  
Method; creates a new XML element with the name specified in the name parameter. The new element initially has no parent, children, or siblings. The method returns a reference to the newly created XML object that represents the element. This method and the XML.createTextNode() method are the constructor methods for creating nodes for an XML object.

**Example**  
The following example creates three XMLNode objects using the createElement() method:

```actionscript
// Create an XML document.
var doc = new XML();

// Create three XML nodes using createElement().
var element1 = doc.createElement("element1");
var element2 = doc.createElement("element2");
var element3 = doc.createElement("element3");

// Place the new nodes into the XML tree.
doc.appendChild(element1);
element1.appendChild(element2);
element1.appendChild(element3);

trace(doc);
// Output: <element1><element2 /><element3 /></element1>
```

**NOTE**  
In Flash Media Server, the output of trace() statements appears in the application log file and Application inspector.

**See also**  
XML.createTextNode()
**XML.createTextNode()**

**Availability**
Flash Media Server 2.

**Usage**

```javascript
my_xml.createTextNode(text)
```

**Parameters**

- `text` A string; the text used to create the new text node.

**Returns**
An XML node.

**Description**
Method; creates a new XML text node with the specified text. The new node initially has no parent, and text nodes cannot have children or siblings. This method returns a reference to the XML object that represents the new text node. This method and the `XML.createElement()` method are the constructor methods for creating nodes for an XML object.

**Example**
The following example creates two XML text nodes using the `createTextNode()` method and places them into existing XML nodes:

```javascript
// Create an XML document.
var doc = new XML();

// Create three XML nodes using createElement().
var element1 = doc.createElement("element1");
var element2 = doc.createElement("element2");
var element3 = doc.createElement("element3");

// Place the new nodes into the XML tree.
doc.appendChild(element1);
  element1.appendChild(element2);
  element1.appendChild(element3);

// Create two XML text nodes using createTextNode().
var textNode1 = doc.createTextNode("textNode1");
var textNode2 = doc.createTextNode("textNode2");

// Place the new nodes into the XML tree.
  element2.appendChild(textNode1);
  element3.appendChild(textNode2);
```
trace(doc);
// Output (with line breaks added between tags):
//   <element1>
//     <element2>textNode1</element2>
//     <element3>textNode2</element3>
//   </element1>

NOTE
In Flash Media Server, the output of trace() statements appears in the application log file and Application inspector.

See also
XML.createElement()

XML.docTypeDecl

Availability
Flash Media Server 2.

Usage
my_xml.docTypeDecl

Description
Property; specifies information about the XML document's DOCTYPE declaration. After the XML text has been parsed into an XML object, the XML.docTypeDecl property of the XML object is set to the text of the XML document's DOCTYPE declaration (for example, <!DOCTYPE greeting SYSTEM "hello.dtd">). This property is set using a string representation of the DOCTYPE declaration, not an XMLNode object.

The ActionScript XML parser is not a validating parser. The DOCTYPE declaration is read by the parser and stored in the XML.docTypeDecl property, but no DTD validation is performed.

If no DOCTYPE declaration occurs during a parse operation, the XML.docTypeDecl property is set to undefined. The XML.toString() method outputs the contents of XML.docTypeDecl immediately after the XML declaration stored in XML.xmlDecl and before any other text in the XML object. If XML.docTypeDecl is undefined, there is no DOCTYPE declaration.

Example
The following example uses the XML.docTypeDecl property to set the DOCTYPE declaration for an XML object:

my_xml.docTypeDecl = "<!DOCTYPE greeting SYSTEM "hello.dtd">";

See also
XML.toString(), XML.xmlDecl
XML.firstChild

Availability
Flash Media Server 2.

Usage
my_xml.firstChild

Description
Property (read-only); evaluates the specified XML object and references the first child in the parent node’s child list. This property is null if the node does not have children. This property is null if the node is a text node. This is a read-only property and cannot be used to manipulate child nodes; use the appendChild(), insertBefore(), and removeNode() methods to manipulate child nodes.

Example
The following example shows how to use XML.firstChild to loop through a node’s child nodes:

```javascript
// Create a new XML document.
var doc = new XML();

// Create a root node.
var rootNode = doc.createElement("rootNode");

// Create three child nodes.
var oldest = doc.createElement("oldest");
var middle = doc.createElement("middle");
var youngest = doc.createElement("youngest");

// Add the rootNode as the root of the XML document tree.
doc.appendChild(rootNode);

// Add each of the child nodes as children of rootNode.
rootNode.appendChild(oldest);
rootNode.appendChild(middle);
rootNode.appendChild(youngest);

// Use firstChild to iterate through the child nodes of rootNode.
for (var aNode = rootNode.firstChild; aNode != null; aNode =
aNode.nextSibling) {
  trace(aNode);
}

// Output:
// <oldest />
```
// <middle />
// <youngest />

NOTE
In Flash Media Server, the output of trace() statements appears in the application log file and Application inspector.

See also
XML.appendChild(), XML.insertBefore(), XML.removeNode()

XML.getBytesLoaded()

Availability
Flash Media Server 2.

Usage
my_xml.getBytesLoaded()

Parameters
None.

Returns
An integer that indicates the number of bytes loaded.

Description
Method; returns the number of bytes loaded (streamed) for the XML document. You can compare the value of getBytesLoaded() with the value of getBytesTotal() to determine what percentage of an XML document has loaded.

See also
XML.getBytesTotal()

XML.getBytesTotal()

Availability
Flash Media Server 2.

Usage
my_xml.getBytesTotal()

Parameters
None.
Returns
An integer.

Description
Method; returns the size of the XML document in bytes.

Example
See the example for XML.getBytesLoaded() on page 182.

See also
XML.getBytesLoaded()

XML.getNamespaceForPrefix()

Availability
Flash Media Server 2.

Usage
my_xml.getNamespaceForPrefix(prefix)

Parameters
prefix  A string; the prefix for which the method returns the associated namespace.

Returns
A String; the namespace that is associated with the specified prefix.

Description
Method; returns the namespace URI that is associated with the specified prefix for the node.
To determine the URI, getPrefixForNamespace() searches up the XML hierarchy from the node, as necessary, and returns the namespace URI of the first xmlns declaration for the given prefix.
If no namespace is defined for the specified prefix, the method returns null.
If you specify an empty string (""") as the prefix and there is a default namespace defined for the node (as in xmlns="http://www.example.com/"), the method returns that default namespace URI.
Example
The following example creates a very simple XML object and outputs the result of a call to
getNamespaceForPrefix():

```javascript
function createXML() {
    var str = "<Outer xmlns:exu="http://www.example.com/util">" + "<exu:Child id='1' />" + "<exu:Child id='2' />" + "<exu:Child id='3' />" + "</Outer>":
    return new XML(str).firstChild;
}

var xml = createXML();
trace(xml.getNamespaceForPrefix("exu")); // output: http://www.example.com/util
trace(xml.getNamespaceForPrefix("")); // output: null
```

See also
XML.getPrefixForNamespace()

XML.getPrefixForNamespace()

Availability
Flash Media Server 2.

Usage
```
my_xml.getPrefixForNamespace(nsURI)
```

Parameters

- **nsURI** A String; the namespace URI for which the method returns the associated prefix.

Returns
A String; the prefix associated with the specified namespace.

Description
Method; returns the prefix that is associated with the specified namespace URI for the node. To determine the prefix, getPrefixForNamespace() searches up the XML hierarchy from the node, as necessary, and returns the prefix of the first xmlns declaration with a namespace URI that matches `nsURI`.

If there is no xmlns assignment for the given URI, the method returns null. If there is an xmlns assignment for the given URI but no prefix is associated with the assignment, the method returns an empty string (""").
Example

The following example creates a very simple XML object and outputs the result of a call to the `getPrefixForNamespace()` method. The Outer XML node, which is represented by the `xmlDoc` variable, defines a namespace URI and assigns it to the `exu` prefix. Calling the `getPrefixForNamespace()` method with the defined namespace URI ("http://www.example.com/util") returns the prefix `exu`, but calling this method with an undefined URI ("http://www.example.com/other") returns null. The first `exu:Child` node, which is represented by the `child1` variable, also defines a namespace URI ("http://www.example.com/child"), but does not assign it to a prefix. Calling this method on the defined, but unassigned, namespace URI returns an empty string.

```javascript
function createXML() {
    var str = "<Outer xmlns:exu="http://www.example.com/util">
        <exu:Child id='1' xmlns="http://www.example.com/child"/>
        <exu:Child id='2' />
        <exu:Child id='3' />
    </Outer>";
    return new XML(str).firstChild;
}

var xmlDoc = createXML();
trace(xmlDoc.getPrefixForNamespace("http://www.example.com/util")); // output: exu
trace(xmlDoc.getPrefixForNamespace("http://www.example.com/other")); // output: null

var child1 = xmlDoc.firstChild;
trace(child1.getPrefixForNamespace("http://www.example.com/child")); // output: [empty string]
trace(child1.getPrefixForNamespace("http://www.example.com/other")); // output: null
```

See also

`XML.getNamespaceForPrefix()`
XML.hasMoreChildNodes()

Availability
Flash Media Server 2.

Usage
my_xml.hasChildNodes()

Parameters
None.

Returns
A Boolean value.

Description
Method; returns true if the specified XML object has child nodes; false otherwise.

Example
The following example creates a new XML packet. If the root node has child nodes, the code loops over each child node to display the name and value of the node.

```actionscript
code
var my_xml = new XML("<login><username>hank</username>
   <password>rudolph</password></login>");
if (my_xml.firstChild.hasMoreChildNodes()) {
   // Use firstChild to iterate through the child nodes of rootNode.
   for (var aNode = my_xml.firstChild.firstChild; aNode != null;
        aNode = aNode.nextSibling) {
      if (aNode.nodeType == 1) {
         trace(aNode.nodeName+":	"+aNode.firstChild.nodeValue);
      }
   }
}
```

The following output appears:

username:hank
password:rudolph

```output
text
username:hank
password:rudolph
```
**XML.ignoreWhite**

**Availability**
Flash Media Server 2.

**Usage**

my_xml.ignoreWhite

XML.prototype.ignoreWhite

**Parameters**

*boolean* A Boolean value.

**Description**

Property; the default setting is false. When set to true, text nodes that contain only white space are discarded during the parsing process. Text nodes with leading or trailing white spaces are unaffected.

Usage 1: You can set the ignoreWhite property for individual XML objects, as shown in the following code:

```
my_xml.ignoreWhite = true;
```

Usage 2: You can set the default ignoreWhite property for XML objects, as shown in the following code:

```
XML.prototype.ignoreWhite = true;
```

**Example**

The following example loads an XML file with a text node that contains only white space; the foyer tag comprises 14 space characters. To run this example, create a text file named flooring.xml, and copy the following tags into it:

```
<house>
  <kitchen> ceramic tile </kitchen>
  <bathroom> linoleum </bathroom>
  <foyer> </foyer>
</house>
```
Create a new Flash document named flooring.fla and save it to the same directory as the XML file. Place the following code into the main timeline:

```actionscript
// Create a new XML object.
var flooring = new XML();

// Set the ignoreWhite property to true (default value is false).
flooring.ignoreWhite = true;

// After loading is complete, trace the XML object.
flooring.onload = function(success) {
    trace(flooring);
}

// Load the XML into the flooring object.
flooring.load("flooring.xml");

/* output (line breaks added for clarity):
<house>
    <kitchen>ceramic tile</kitchen>
    <bathroom>linoleum</bathroom>
    <foyer>
</house>
*/
```

**NOTE**

In Flash Media Server, the output of `trace()` statements appears in the application log file and Application inspector.

If you change the setting of `flooring.ignoreWhite` to `false`, or simply remove that line of code entirely, the 14 space characters in the `foyer` tag are preserved:

```actionscript
... // Set the ignoreWhite property to false (default value). 
flooring.ignoreWhite = false;
...

/* output (line breaks added for clarity):
<house>
    <kitchen> ceramic tile </kitchen>
    <bathroom>linoleum</bathroom>
    <foyer> </foyer>
</house>
*/
```
**XML.insertBefore()**

**Availability**
Flash Media Server 2.

**Usage**

```javascript
my_xml.insertBefore(childNode, beforeNode)
```

**Parameters**

- `childNode`: The XMLNode object to be inserted.
- `beforeNode`: The XMLNode object before the insertion point for the `childNode` node.

**Returns**

Nothing.

**Description**

Method; inserts a new child node into the XML object’s child list, before the `beforeNode` node. If `beforeNode` is not a child of `my_xml`, the insertion fails.

**Example**

The following example is an excerpt from the `XML.cloneNode()` example:

```javascript
// Create a copy of the middle node using cloneNode().
var middle2 = middle.cloneNode(false);

// Insert the clone node into rootNode
// between the middle and youngest nodes.
rootNode.insertBefore(middle2, youngest);
```

**XML.lastChild**

**Availability**
Flash Media Server 2.

**Usage**

```javascript
my_xml.lastChild
```

**Description**

Property (read-only); an XMLNode value that references the last child in the node’s child list. The `XML.lastChild` property is `null` if the node does not have children. This property cannot be used to manipulate child nodes; use the `appendChild()`, `insertBefore()`, and `removeNode()` methods to manipulate child nodes.
Example

The following example uses the `XML.lastChild` property to iterate through the child nodes of an `XMLNode` object, starting with the last item in the node’s child list and ending with the first child of the node’s child list:

```javascript
// Create a new XML document.
var doc = new XML();

// Create a root node.
var rootNode = doc.createElement("rootNode");

// Create three child nodes.
var oldest = doc.createElement("oldest");
var middle = doc.createElement("middle");
var youngest = doc.createElement("youngest");

// Add the rootNode as the root of the XML document tree.
doc.appendChild(rootNode);

// Add each of the child nodes as children of rootNode.
rootNode.appendChild(oldest);
rootNode.appendChild(middle);
rootNode.appendChild(youngest);

// Use lastChild to iterate through the child nodes of rootNode.
for (var aNode = rootNode.lastChild; aNode != null; aNode =
aNode.previousSibling) {
    trace(aNode);
}

/*
output:
<youngeth />
<middle />
<oldest />
*/
```

**Note**

In Flash Media Server, the output of `trace()` statements appears in the application log file and Application inspector.
The following example creates a new XML packet and uses the `XML.lastChild` property to iterate through the child nodes of the root node:

```
// Create a new XML document.
var doc = new XML("<rootNode><oldest /><middle /><youngest /></rootNode>");

var rootNode = doc.firstChild;

// Use lastChild to iterate through the child nodes of rootNode.
for (var aNode = rootNode.lastChild; aNode != null; 
aNode=aNode.previousSibling) {
    trace(aNode);
}
```

```
/*
output:
<younget />
<middle />
<oldest />
*/
```

```
NOTE In Flash Media Server, the output of `trace()` statements appears in the application log file and Application inspector.
```

See also

`XML.appendChild()`, `XML.insertBefore()`, `XML.removeNode()`

### XML.load()

**Availability**

Flash Media Server 2.

**Usage**

`my_xml.load(url)`

**Parameters**

- `url` A string; the URL where the XML document to be loaded is located. If the SWF file that issues this call is running in a web browser, `url` must be in the same domain as the SWF file.

**Returns**

Nothing.
Description
Method; loads an XML document from the specified URL, and replaces the contents of the specified XML object with the downloaded XML data. The URL is relative and is called using HTTP. The load process is asynchronous; it does not finish immediately after the load() method is executed.

When the load() method is executed, the XML object property loaded is set to false. When the XML data finishes downloading, the loaded property is set to true, and the onLoad event handler is invoked. The XML data is not parsed until it is completely downloaded. If the XML object previously contained any XML trees, they are discarded.

You can define a custom function that executes when the onLoad event handler of the XML object is invoked.

Example
The following simple example uses the XML.load() method:

```javascript
// Create a new XML object.
var flooring = new XML();

// Set the ignoreWhite property to true (default value is false).
flooring.ignoreWhite = true;

// After loading is complete, trace the XML object.
flooring.onLoad = function(success) {
    trace(flooring);
};

// Load the XML into the flooring object.
flooring.load("flooring.xml");
```

For the contents of the flooring.xml file, and the output that this example produces, see the example for XML.IgnoreWhite on page 187.

In Flash Media Server, the output of trace() statements appears in the application log file and Application inspector.

See also
XML.loaded, XML.onLoad
XML.loaded

Availability
Flash Media Server 2.

Usage
\textit{my\_xml}.loaded

Description
Property; a Boolean value that is \texttt{true} if the document-loading process initiated by the \texttt{XML.load()} call completed successfully; \texttt{false} otherwise.

Example
The following example uses the \texttt{XML.loaded} property in a simple script:

\begin{verbatim}
var my_xml = new XML();
my_xml.ignoreWhite = true;
my_xml.onLoad = function(success) {
    trace("success: "+success);
    trace("loaded: "+my_xml.loaded);
    trace("status: "+my_xml.status);
};
my_xml.load("http://www.flash-mx.com/mm/problems/products.xml");
\end{verbatim}

Information writes to the log file when the \texttt{onLoad} handler is invoked. If the call completes successfully, the \texttt{loaded} status \texttt{true} is written to the log file, as shown in the following example:

\begin{verbatim}
success: true
loaded: true
status: 0
\end{verbatim}

\textbf{NOTE}
In Flash Media Server, the output of \texttt{trace()} statements appears in the application log file and Application inspector.

See also
\texttt{XML.load()}, \texttt{XML.onLoad}
XML.localName

**Availability**
Flash Media Server 2.

**Usage**

```as
my_xml.localName
```

**Description**

Property (read-only); the local name portion of the XML node's name. This is the element name without the namespace prefix. For example, the node `<contact:mailbox>bob@example.com</contact:mailbox>` has the local name "mailbox", and the prefix "contact", which comprise the full element name "contact.mailbox".

You can access the namespace prefix via the `XML.prefix` property of the XML node object. The `XML.nodeName` property returns the full name (including the prefix and the local name).

**Example**

This example uses a SWF file and an XML file located in the same directory. The XML file, named "SoapSample.xml" contains the following:

```xml
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
  <soap:Body xmlns:w="http://www.example.com/weather">
    <w:GetTemperature>
      <w:City>San Francisco</w:City>
    </w:GetTemperature>
  </soap:Body>
</soap:Envelope>
```

The source for the SWF file contains the following script (note the comments for the Output strings):

```as
var xmlDoc = new XML()
xmlDoc.ignoreWhite = true;
xmlDoc.load("SoapSample.xml")
xmlDoc.onLoad = function(success)
{
  var tempNode = xmlDoc.childNodes[0].childNodes[0].childNodes[0];
  trace("w:GetTemperature localname: " + tempNode.localName); // Output:  ... GetTemperature
  var soapEnvNode = xmlDoc.childNodes[0];
  trace("soap:Envelope localname: " + soapEnvNode.localName); // Output:  ... Envelope
}
```
See also
XML.nodeName, XML.prefix

XML.namespaceURI

Availability
Flash Media Server 2.

Usage
my_xml.namespaceURI

Description
Property (read-only); if the XML node has a prefix, namespaceURI is the value of the xmlns declaration for that prefix (the URI), which is typically called the namespace URI. The xmlns declaration is in the current node or in a node higher in the XML hierarchy.

If the XML node does not have a prefix, the value of the namespaceURI property depends on whether there is a default namespace defined (as in xmlns="http://www.example.com/`). If there is a default namespace, the value of the namespaceURI property is the value of the default namespace. If there is no default namespace, the namespaceURI property for that node is an empty string (""").

You can use the XML.getNamespaceForPrefix() method to identify the namespace associated with a specific prefix. The namespaceURI property returns the prefix associated with the node name.

Example
The following example shows how the namespaceURI property is affected by the use of prefixes. The XML file used in the example is named SoapSample.xml and contains the following tags:

```xml
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
<soap:Body xmlns:wm="http://www.example.com/weather">
<w:GetTemperature>
<w:City>San Francisco</w:City>
</w:GetTemperature>
</soap:Body>
</soap:Envelope>
```
The source for the server-side ASC file contains the following script (note the comments for the Output strings). For tempNode, which represents the w:GetTemperature node, the value of namespaceURI is defined in the soap:Body tag. For soapBodyNode, which represents the soap:Body node, the value of namespaceURI is determined by the definition of the soap prefix in the node above it, rather than the definition of the w prefix that the soap:Body node contains.

```actionscript
var xmlDoc = new XML();
xmlDoc.load("SoapSample.xml");
xmlDoc.ignoreWhite = true;
xmlDoc.onLoad = function(success:Boolean)
{
    var tempNode:XMLNode =
        xmlDoc.childNodes[0].childNodes[0].childNodes[0];
    trace("w:GetTemperature namespaceURI: " + tempNode.namespaceURI);
    // Output: ... http://www.example.com/weather

    trace("w:GetTemperature soap namespace: " +
        tempNode.getNamespaceForPrefix("soap"));
    // Output: ... http://www.w3.org/2001/12/soap-envelope

    var soapBodyNode = xmlDoc.childNodes[0].childNodes[0];
    trace("soap:Envelope namespaceURI: " + soapBodyNode.namespaceURI);
    // Output: ... http://www.w3.org/2001/12/soap-envelope
}
```

The following example uses XML tags without prefixes. It uses a SWF file and an XML file located in the same directory. The XML file, named NoPrefix.xml contains the following tags.

```xml
<?xml version="1.0"?>
<rootnode>
    <simplenode xmlns="http://www.w3.org/2001/12/soap-envelope">
        <innernode />
    </simplenode>
</rootnode>
```

The source for the server-side script file contains the following code (note the comments for the Output strings). The rootNode does not have a default namespace, so its namespaceURI value is an empty string. The simpleNode defines a default namespace, so its namespaceURI value is the default namespace. The innerNode does not define a default namespace, but uses the default namespace defined by simpleNode, so its namespaceURI value is the same as that of simpleNode.

```actionscript
var xmlDoc = new XML();
xmlDoc.load("NoPrefix.xml");
xmlDoc.ignoreWhite = true;
xmlDoc.onLoad = function(success)
{
    var rootNode = xmlDoc.childNodes[0];
```
var simpleNode = xmlDoc.childNodes[0].childNodes[0];
trace("simpleNode Node namespaceURI: " + simpleNode.namespaceURI);
// Output: ... http://www.w3.org/2001/12/soap-envelope

var innerNode = xmlDoc.childNodes[0].childNodes[0].childNodes[0];
trace("innerNode Node namespaceURI: " + innerNode.namespaceURI);
// Output: ... http://www.w3.org/2001/12/soap-envelope

XML.nextSibling

Availability
Flash Media Server 2.

Usage
my_xml.nextSibling

Description
Property (read-only); an XMLNode value that references the next sibling in the parent node's child list. This property is null if the node does not have a next sibling node. This property cannot be used to manipulate child nodes; use the appendChild(), insertBefore(), and removeNode() methods to manipulate child nodes.

Example
The following example is an excerpt from the example for the XML.firstChild property. It shows how you can use the XML.nextSibling property to loop through an XMLNode object's child nodes.

for (var aNode = rootNode.firstChild; aNode != null; aNode = aNode.nextSibling) {
  trace(aNode);
}

NOTE
In Flash Media Server, the output of trace() statements appears in the application log file and Application inspector.

See also
XML.appendChild(), XML.insertBefore(), XML.removeNode()
XML.nodeName

Availability
Flash Media Server 2.

Usage
my_xml.nodeName

Description
Property; a string representing the node name of the XML object. If the XML object is an
XML element (nodeType==1), nodeName is the name of the tag that represents the node in
the XML file. For example, TITLE is the nodeName of an HTML TITLE tag. If the XML
object is a text node (nodeType==3), nodeName is null.

Example
The following example creates an element node and a text node, and checks the node name of
each:

```
// Create an XML document.
var doc = new XML();

// Create an XML node using createElement().
var myNode = doc.createElement("rootNode");

// Place the new node into the XML tree.
doc.appendChild(myNode);

// Create an XML text node using createTextNode().
var myTextNode = doc.createTextNode("textNode");

// Place the new node into the XML tree.
myNode.appendChild(myTextNode);

trace(myNode.nodeName);
trace(myTextNode.nodeName);
```

/*
output:
rootNode
null
*/
The following example creates a new XML packet. If the root node has child nodes, the code loops over each child node to display the name and value of the node. Add the following ActionScript to your ASC file:

```actionscript
var my_xml = new XML("<login><username>hank</username>
 <password>rudolph</password></login>);
if (my_xml.firstChild.hasChildNodes()) {
    // Use firstChild to iterate through the child nodes of rootNode.
    for (var aNode = my_xml.firstChild.firstChild; aNode != null; 
        aNode=aNode.nextSibling) {
        if (aNode.nodeType == 1) {
            trace(aNode.nodeName+":	"+aNode.firstChild.nodeValue);
        }
    }
}
```

The following node names appear:

username:hank
password:rudolph

<table>
<thead>
<tr>
<th>Integer value</th>
<th>Defined constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ELEMENT_NODE</td>
</tr>
<tr>
<td>2</td>
<td>ATTRIBUTE_NODE</td>
</tr>
<tr>
<td>3</td>
<td>TEXT_NODE</td>
</tr>
</tbody>
</table>

In Flash Media Server, the output of `trace()` statements appears in the application log file and Application inspector.

See also

XML.nodeType

**XML.nodeType**

**Availability**

Flash Media Server 2.

**Usage**

`my_xml.nodeType`

**Description**

Property (read-only); a `nodeType` value, either 1 for an XML element or 3 for a text node.

The `nodeType` is a numeric value from the NodeType enumeration in the W3C DOM Level 1 Recommendation. The following table lists the values:
In Flash Player, the built-in XML class supports only 1 (**ELEMENT_NODE**) and 3 (**TEXT_NODE**).

**Example**

The following example creates an element node and a text node and checks the node type of each:

```actionscript
// Create an XML document.
var doc = new XML();

// Create an XML node using createElement().
var myNode = doc.createElement("rootNode");

// Place the new node into the XML tree.
doc.appendChild(myNode);

// Create an XML text node using createTextNode().
var myTextNode = doc.createTextNode("textNode");

// Place the new node into the XML tree.
myNode.appendChild(myTextNode);

trace(myNode.nodeType);
trace(myTextNode.nodeType);
```

```
output:
1
3
```

**NOTE**

In Flash Media Server, the output of **trace()** statements appears in the application log file and Application inspector.
See also
XML.nodeValue

XML.nodeValue

Availability
Flash Media Server 2.

Usage
my_xml.nodeValue

Description
Property; the node value of the XML object. If the XML object is a text node, the nodeType is 3, and the nodeValue is the text of the node. If the XML object is an XML element (nodeType is 1), nodeValue is null and read-only.

Example
The following example creates an element node and a text node and checks the node value of each:

// Create an XML document.
var doc = new XML();

// Create an XML node using createElement().
var myNode = doc.createElement("rootNode");

// Place the new node into the XML tree.
doc.appendChild(myNode);

// Create an XML text node using createTextNode().
var myTextNode = doc.createTextNode("myTextNode");

// Place the new node into the XML tree.
myNode.appendChild(myTextNode);

trace(myNode.nodeValue);
trace(myTextNode.nodeValue);

/*
output:
null
myTextNode
*/
The following example creates and parses an XML packet. The code loops through each child node, and displays the node value using the `firstChild` property and `firstChild.nodeValue`.

```actionscript
var my_xml = new XML("<login><username>morton</username>
   <password>good&amp;evil</password></login>"租车
trace("using firstChild:");
for (var i = 0; i<my_xml.firstChild.childNodes.length; i++) {
   trace("\t"+my_xml.firstChild.childNodes[i].firstChild);
}
trace(""租车
trace("using firstChild.nodeValue:");
for (var i = 0; i<my_xml.firstChild.childNodes.length; i++) {
   trace("\t"+my_xml.firstChild.childNodes[i].firstChild.nodeValue);
}
```

The following information is written to the log file:

using firstChild:
   morton
good&evil

using firstChild.nodeValue:
   morton
good&evil

**NOTE**
In Flash Media Server, the output of `trace()` statements appears in the application log file and Application inspector.

**See also**

`XML.nodeType`

**XML.onData**

**Availability**

Flash Media Server 2.

**Usage**

```actionscript
my_xml.onData = function(src) {}  
```

**Parameters**

`src` A string or `undefined`; the raw data, usually in XML format, that is sent by the server.
Returns
Nothing.

Description
Event handler; invoked when XML text has been completely downloaded from the server or when an error occurs when XML text downloads from a server. This handler is invoked before the XML is parsed, and you can use it to call a custom parsing routine instead of using the Flash XML parser. The src parameter is a string that contains XML text downloaded from the server, unless an error occurs during the download. In this situation, the src parameter is undefined.

By default, the XML.onData event handler invokes XML.onLoad. You can override the XML.onData event handler with custom behavior, but XML.onLoad is not called unless you call it in your XML.onData implementation.

Example
The following example shows what the XML.onData event handler looks like by default:

```javascript
XML.prototype.onData = function (src) {
    if (src == undefined) {
        this.onLoad(false);
    } else {
        this.parseXML(src);
        this.loaded = true;
        this.onLoad(true);
    }
}
```

You can override the XML.onData event handler to intercept the XML text without parsing it.

XML.onHTTPStatus

Availability
Flash Media Server 2.

Usage

```javascript
myXML.onHTTPStatus(httpStatus)
```

Parameters

httpStatus A number; the HTTP status code returned by the server. For example, a value of 404 indicates that the server has not found a match for the requested URI. HTTP status codes can be found in sections 10.4 and 10.5 of the HTTP specification.
Description
Event handler; invoked when Flash Media Server receives an HTTP status code from the server. This handler lets you capture and act on HTTP status codes.

The onHTTPStatus handler is invoked before onData, which triggers calls to onLoad with a value of undefined if the load fails. After onHTTPStatus is triggered, onData is always triggered, whether or not you override onHTTPStatus. To best use the onHTTPStatus handler, you should write a function to catch the result of the onHTTPStatus call; you can then use the result in your onData and onLoad handlers. If onHTTPStatus is not invoked, this indicates that the FMS did not try to make the URL request.

If Flash Media Server cannot get a status code from the server, or if it cannot communicate with the server, the default value of 0 is passed to your ActionScript code.

Example
The following example shows how to use onHTTPStatus to help with debugging. The example collects HTTP status codes and assigns their value and type to an instance of the XML object. (Notice that this example creates the instance members this.httpStatus and this.httpStatusType at runtime.) The onData handler uses these instance members to trace information about the HTTP response that can be useful in debugging.

```actionscript
var myXml = new XML();
myXml.onHTTPStatus = function(httpStatus) {
    this.httpStatus = httpStatus;
    if(httpStatus < 100) {
        this.httpStatusType = "flashError";
    } else if(httpStatus < 200) {
        this.httpStatusType = "informational";
    } else if(httpStatus < 300) {
        this.httpStatusType = "successful";
    } else if(httpStatus < 400) {
        this.httpStatusType = "redirection";
    } else if(httpStatus < 500) {
        this.httpStatusType = "clientError";
    } else if(httpStatus < 600) {
        this.httpStatusType = "serverError";
    }
}
myXml.onData = function(src) {
    trace(">> " + this.httpStatusType + ": " + this.httpStatus);
};
```
if(src != undefined) {
    this.parseXML(src);
    this.loaded = true;
    this.onLoad(true);
} else {
    this.onLoad(false);
}

myXml.onLoad = function(success) {
}

myXml.load("http://weblogs.macromedia.com/mxna/xml/rss.cfm?query=byMostRecent&languages=1");

See also
LoadVars.onHTTPStatus, XML.send(), XML.sendAndLoad()

XML.onLoad

Availability
Flash Media Server 2.

Usage
my_xml.onLoad = function (success) {}

Parameters
success  A Boolean value that is true if the XML object successfully loads with a XML.load() or XML.sendAndLoad() operation; false otherwise.

Returns
Nothing.

Description
Event handler; invoked by Flash Media Server when an XML document is received from the server. If the XML document is received successfully, the success parameter is true. If the document was not received, or if an error occurred in receiving the response from the server, the success parameter is false. The default implementation of this method is not active. To override the default implementation, you must assign a function that contains custom actions.
Example
The following example includes ActionScript for a simple e-commerce storefront application. The `sendAndLoad()` method transmits an XML element that contains the user's name and password and uses an XML.onLoad handler to process the reply from the server.

```actionscript
var login_str = "<login username=""+username_txt.text+""
    password=""+password_txt.text+""
    />";
var my_xml = new XML(login_str);
var myLoginReply_xml = new XML();
myLoginReply_xml.ignoreWhite = true;
myLoginReply_xml.onLoad = function(success){
    if (success) {
        if ((myLoginReply_xml.firstChild.nodeName == "packet") &&
            (myLoginReply_xml.firstChild.attributes.success == "true")) {
            gotoAndStop("loggedIn");
        } else {
            gotoAndStop("loginFailed");
        }
    } else {
        gotoAndStop("connectionFailed");
    }
};
my_xml.sendAndLoad("http://www.flash-mx.com/mm/login_xml.cfm",
    myLoginReply_xml);
```

See also
XML.load(), XML.sendAndLoad()

**XML.parentNode**

**Availability**
Flash Media Server 2.

**Usage**
`my_xml.parentNode`

**Description**
Property (read-only); an XMLNode value that references the parent node of the specified XML object or returns `null` if the node has no parent. This property cannot be used to manipulate child nodes; use the `appendChild()`, `insertBefore()`, and `removeNode()` methods instead.
Example
The following example creates an XML packet and writes the parent node of the `username` node to the log file:

```javascript
var my_xml = new XML("<login><username>morton</username>
  <password>good&amp;evil</password></login>");

// The first child is the <login /> node.
var rootNode = my_xml.firstChild;

// The first child of the root is the <username /> node.
var targetNode = rootNode.firstChild;
trace("the parent node of 'username' is: " + targetNode.nodeName + " is:
  " + targetNode.parentNode.nodeName);
trace("contents of the parent node are:
  " + targetNode.parentNode);

/* output (line breaks added for clarity):
the parent node of 'username' is: login
contents of the parent node are:
  <login>
    <username>morton</username>
    <password>good&amp;evil</password>
  </login>
*/
```

See also
XML.appendChild(), XML.insertBefore(), XML.removeNode()

**XML.parseXML()**

**Availability**
Flash Media Server 2.

**Usage**
`my_xml.parseXML(source)`

**Parameters**
`source` A string; the XML text to be parsed and passed to the specified XML object.

**Returns**
Nothing.
Description
Method: parses the XML text specified in the source parameter and populates the specified XML object with the resulting XML tree. Any existing trees in the XML object are discarded.

Example
The following example creates and parses an XML packet:

```actionscript
var xml_str = "<state name="California">
  <city>San Francisco</city></state>
```

// Defining the XML source within the XML constructor:
var my1_xml = new XML(xml_str);
trace(my1_xml.firstChild.attributes.name); // output: California

// Defining the XML source using the XML.parseXML method:
var my2_xml = new XML();
my2_xml.parseXML(xml_str);
trace(my2_xml.firstChild.attributes.name); // output: California
```

**NOTE**
In Flash Media Server, the output of `trace()` statements appears in the application log file and Application inspector.

XML.prefix

Availability
Flash Media Server 2.

Usage
`my_xml.prefix`

Description
Property (read-only): the prefix portion of the XML node name. For example, in the node `<contact:mailbox/>bob@example.com</contact:mailbox>` the prefix “contact” and the local name “mailbox”, comprise the full element name contact.mailbox.
Example

A directory contains a server-side script file and an XML file. The XML file, named "SoapSample.xml" contains the following:

```xml
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
  <soap:Body xmlns:w="http://www.example.com/weather">
    <w:GetTemperature>
      <w:City>San Francisco</w:City>
    </w:GetTemperature>
  </soap:Body>
</soap:Envelope>
```

The source for the server-side script file contains the following code (note the comments for the Output strings):

```javascript
var xmlDoc = new XML();
xmlDoc.ignoreWhite = true;
xmlDoc.load("SoapSample.xml");
xmlDoc.onLoad = function(success)
{
  var tempNode = xmlDoc.childNodes[0].childNodes[0].childNodes[0];
  trace("w:GetTemperature prefix: " + tempNode.prefix); // Output: ... w
  var soapEnvNode = xmlDoc.childNodes[0];
  trace("soap:Envelope prefix: " + soapEnvNode.prefix); // Output: ...
  soap
}
```

**XML.previousSibling**

**Availability**

Flash Media Server 2.

**Usage**

`my_xml.previousSibling`

**Description**

Property (read-only); an XMLNode value that references the previous sibling in the parent node's child list. The property has a value of `null` if the node does not have a previous sibling node. This property cannot be used to manipulate child nodes; use the `XML.appendChild()`, `XML.insertBefore()`, and `XML.removeNode()` methods instead.
Example
The following example is an excerpt from the example for the `XML.lastChild` property and shows how you can use the `XML.previousSibling` property to loop through an `XMLNode` object's child nodes:

```javascript
for (var aNode = rootNode.lastChild; aNode != null; aNode = aNode.previousSibling) {
    trace(aNode);
}
```

See also
`XML.appendChild()`, `XML.insertBefore()`, `XML.removeNode()`

**XML.removeNode()**

**Availability**
Flash Media Server 2.

**Usage**
`my_xml.removeNode()`

**Parameters**
None.

**Returns**
Nothing.

**Description**
Method; removes the specified XML object from its parent and deletes all descendants of the node.

In Flash Media Server, the output of `trace()` statements appears in the application log file and Application inspector.
Example
The following example creates an XML packet and then deletes the specified XML object and its descendant nodes:

```javascript
var xml_str = "<state name="California">
  <city>San Francisco</city>
</state>";

var my_xml = new XML(xml_str);
var cityNode = my_xml.firstChild.firstChild;
trace("before XML.removeNode():
"+my_xml);
cityNode.removeNode();
trace("*");
trace("after XML.removeNode():
"+my_xml);

/* output (line breaks added for clarity):
before XML.removeNode():
  <state name="California">
    <city>San Francisco</city>
  </state>
after XML.removeNode():
  <state name="California" />
*/
```

In Flash Media Server, the output of `trace()` statements appears in the application log file and Application inspector.

**XML.send()**

**Availability**
Flash Media Server 2.

**Usage**
`my_xml.send(url, [fileObj])`

**Parameters**
- `url` A string; the destination URL for the specified XML object.
- `fileObj` File; (optional) a File object, that is not read-only, to which the response is written. If the File object is not open, Flash Media Server opens the file, writes to the file, and closes the file. If the File object is open, Flash Media Server writes to the file and leaves the file open.
Returns
Nothing.

Description
Method; encodes the specified XML object into an XML document and sends it to the specified URL using the POST method in a browser. The Flash test environment uses only the GET method.

Example
The following example defines an XML packet and sets the content type for the XML object. The data is then sent to a server and writes the result in a File object.

```javascript
var my_xml = new XML("<highscore><name>Ernie</name><score>13045</score></highscore>" );
my_xml.contentType = "text/xml";
my_xml.send("http://www.flash-mx.com/mm/highscore.cfm", myFile);
```

See also
XML.sendAndLoad()

XML.sendAndLoad()

Media Server 2.

Usage
```javascript
my_xml.sendAndLoad(url, targetXMLobject)
```

Parameters

- `url` A string; the destination URL for the specified XML object. If the SWF file issuing this call is running in a web browser, `url` must be in the same domain as the SWF file.
- `targetXMLobject` An XML object created with the XML constructor method that will receive the return information from the server.

Returns
Nothing.

Description
Method; encodes the specified XML object into an XML document, sends it to the specified URL using the POST method, downloads the server's response, and loads it into the `targetXMLobject` object specified in the parameters. The server response loads the same as the response to the `XML.load()` method.
When `sendAndLoad()` is executed, the XML object property `loaded` is set to `false`. When the XML data finishes downloading, the `loaded` property is set to `true` if the data successfully loads and the `onLoad` event handler is invoked. The XML data is not parsed until it is completely downloaded. If the XML object previously contained any XML trees, they are discarded.

**Example**

The following example includes ActionScript for a simple e-commerce storefront application. The XML `sendAndLoad()` method transmits an XML element that contains the user's name and password and uses an `onLoad` handler to process the reply from the server.

```actionscript
var login_str = "<login username="+username_txt.text+" password="+password_txt.text+" />
var my_xml = new XML(login_str);
var myLoginReply_xml = new XML();
myLoginReply_xml.ignoreWhite = true;
myLoginReply_xml.onLoad = myOnLoad;
my_xml.sendAndLoad("http://www.flash-mx.com/mm/login_xml.cfm", myLoginReply_xml);

function myOnLoad(success) {
    if (success) {
        if ((myLoginReply_xml.firstChild.nodeName == "packet") &&
            (myLoginReply_xml.firstChild.attributes.success == "true")) {
            gotoAndStop("loggedIn");
        } else {
            gotoAndStop("loginFailed");
        }
    } else {
        gotoAndStop("connectionFailed");
    }
}
```

**See also**

XML.send(), XML.load(), XML.loaded, XML.onLoad
**XML.status**

**Availability**
Flash Media Server 2.

**Usage**

`my_xml.status`

**Description**

Property; automatically sets and returns a numeric value that indicates whether an XML document was successfully parsed into an XML object. The following list contains the numeric status codes with descriptions:

- **0**  No error; parse was completed successfully.
- **-2**  A CDATA section was not properly terminated.
- **-3**  The XML declaration was not properly terminated.
- **-4**  The DOCTYPE declaration was not properly terminated.
- **-5**  A comment was not properly terminated.
- **-6**  An XML element was malformed.
- **-7**  Out of memory.
- **-8**  An attribute value was not properly terminated.
- **-9**  A start tag was not matched with an end tag.
- **-10** An end tag was encountered without a matching start tag.

**Example**

The following example loads an XML packet into a SWF file. A status message indicates whether the XML loads and parses successfully. Add the following ActionScript to your FLA or AS file:

```actionscript
var my_xml = new XML();
my_xml.onload = function(success) {
  if (success) {
    if (my_xml.status == 0) {
      trace("XML was loaded and parsed successfully");
    } else {
      trace("XML was loaded successfully, but was unable to be parsed.");
    }
  } var errorMessage;
  switch (my_xml.status) {
  case 0:
    errorMessage = "No error; parse was completed successfully.";
    break;
  case -2:
```
errorMessage = "A CDATA section was not properly terminated."
break;
case -3 :
errorMessage = "The XML declaration was not properly terminated."
break;
case -4 :
errorMessage = "The DOCTYPE declaration was not properly terminated."
break;
case -5 :
errorMessage = "A comment was not properly terminated."
break;
case -6 :
errorMessage = "An XML element was malformed."
break;
case -7 :
errorMessage = "Out of memory."
break;
case -8 :
errorMessage = "An attribute value was not properly terminated."
break;
case -9 :
errorMessage = "A start tag was not matched with an end tag."
break;
case -10 :
errorMessage = "An end tag was encountered without a matching start tag."
break;
default :
errorMessage = "An unknown error has occurred."
break;
}
trace("status: "+my_xml.status+ ("+errorMessage+")");
} else {
trace("Unable to load/pase XML. (status: "+my_xml.status+")");
}
}
my_xml.load("http://www.flash-mx.com/mm/badxml.xml");

**NOTE**
In Flash Media Server, the output of trace() statements appears in the application log file and Application inspector.
**XML.toString()**

**Availability**
Flash Media Server 2.

**Usage**
my_xml.toString()

**Parameters**
None.

**Returns**
A string.

**Description**
Method; evaluates the specified XML object, constructs a textual representation of the XML structure, including the node, children, and attributes, and returns the result as a string.

For top-level XML objects (those created with the constructor), the `XML.toString()` method outputs the document's XML declaration (stored in the `XML.xmlDecl` property), followed by the document's DOCTYPE declaration (stored in the `XML.docTypeDecl` property), followed by the text representation of all XML nodes in the object. The XML declaration is not output if the `XML.xmlDecl` property is undefined. The DOCTYPE declaration is not output if the `XML.docTypeDecl` property is undefined.

**Example**
The following example of the `XML.toString()` method sends `<h1>test</h1>` to the log file:
```actionscript
var node = new XML('<h1>test</h1>');
trace(node.toString());
```

**NOTE**
In Flash Media Server, the output of `trace()` statements appears in the application log file and Application inspector.

**See also**
`XML.docTypeDecl`, `XML.xmlDecl`
**XML.xmlDecl**

**Availability**
Flash Media Server 2.

**Usage**

`my_xml.xmlDecl`

**Description**
Property; a string that specifies information about a document’s XML declaration. After the XML document is parsed into an XML object, this property is set to the text of the document’s XML declaration. This property is set using a string representation of the XML declaration, not an XMLNode object. If no XML declaration is encountered during a parse operation, the property is set to `undefined`. The `XML.toString()` method outputs the contents of the `XML.xmlDecl` property before any other text in the XML object. If the `XML.xmlDecl` property contains the `undefined` type, no XML declaration is output.

**Example**
The following example loads an XML file and outputs information about the file:

```javascript
var my_xml = new XML();
my_xml.ignoreWhite = true;
my_xml.onLoad = function(success)
{
  if (success)
  {
    trace("xmlDecl: "+ my_xml.xmlDecl);
    trace("contentType: "+ my_xml.contentType);
    trace("docTypeDecl: "+ my_xml.docTypeDecl);
    trace("packet: "+ my_xml.toString());
  }
  else
  {
    trace("Unable to load remote XML.");
  }
}
my_xml.load("http://foo.com/crossdomain.xml");
```

**See also**

`XML.docTypeDecl`, `XML.toString()`
XMLSocket class

Availability
Flash Media Server 2.

Description
The XMLSocket class implements client sockets that let Flash Media Server communicate with a server identified by an IP address or domain name. The XMLSocket class is useful for client-server applications that require low latency, such as real-time chat systems. A traditional HTTP-based chat solution frequently polls the server and downloads new messages using an HTTP request. In contrast, an XMLSocket chat solution maintains an open connection to the server, which lets the server immediately send incoming messages without a request from the client.

To use the XMLSocket class, the server computer must run a daemon that understands the protocol used by the XMLSocket class. The protocol is described in the following list:

- XML messages are sent over a full-duplex TCP/IP stream socket connection.
- Each XML message is a complete XML document, terminated by a zero (0) byte.
- An unlimited number of XML messages can be sent and received over a single XMLSocket connection.

The following restriction applies to how and where an XMLSocket object can connect to the server:

- The XMLSocket.connect() method can connect only to TCP port numbers greater than or equal to 1024. One consequence of this restriction is that the server daemons that communicate with the XMLSocket object must also be assigned to port numbers greater than or equal to 1024. Port numbers less than 1024 are often used by system services such as FTP, Telnet, and HTTP, which prohibits XMLSocket objects from these ports for security reasons. The port number restriction limits the possibility that these resources can be inappropriately accessed and abused.

To use the methods of the XMLSocket class, you must first use the constructor, new XMLSocket, to create an XMLSocket object.
Method summary for the XMLSocket class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMLSocket.close()</td>
<td>Closes an open socket connection.</td>
</tr>
<tr>
<td>XMLSocket.connect()</td>
<td>Establishes a connection to the specified server.</td>
</tr>
<tr>
<td>XMLSocket.send()</td>
<td>Sends an XML object to the server.</td>
</tr>
</tbody>
</table>

Event handler summary for the XMLSocket class

<table>
<thead>
<tr>
<th>Event handler</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMLSocket.onClose</td>
<td>Invoked when an XMLSocket connection is closed.</td>
</tr>
<tr>
<td>XMLSocket.onConnect</td>
<td>Invoked by Flash Media Server when a connection request initiated through XMLSocket.connect() succeeds or fails.</td>
</tr>
<tr>
<td>XMLSocket.onData</td>
<td>Invoked when an XML message has been downloaded from the server.</td>
</tr>
<tr>
<td>XMLSocket.onXML</td>
<td>Invoked when an XML object arrives from the server.</td>
</tr>
</tbody>
</table>

Constructor for the XMLSocket class

Availability
Flash Communication Server MX 1.5.

Usage
new XMLSocket(streamOrFlash)

Parameters
streamOrFlash A string indicating whether this object is an XMLSocket object or an XMLStreams object. The parameter can have one of the following two values: "flash" or "stream".

Returns
A reference to an XMLSocket object or an XMLStreams object.

Description
Constructor; creates a new XMLSocket object ("flash") or a new XMLStreams object ("stream"). The XMLSocket and XMLStreams objects are not initially connected to any server. You must call XMLSocket.connect() to connect the object to a server.
For more information about the XMLStreams class, see “XMLStreams class” on page 227.

Example
The following example creates an XMLSocket object:
var socket = new XMLSocket("flash");
The following example creates an XMLStreams object:
var stream = new XMLSocket("stream");

XMLSocket.close()

Availability
Flash Media Server 2.

Usage
myXMLSocket.close()

Parameters
None.

Returns
Nothing.

Description
Method; closes the connection specified by the XMLSocket object.

Example
The following simple example creates an XMLSocket object, attempts to connect to the server, and then closes the connection:
var socket = new XMLSocket();
socket.connect(null, 2000);
socket.close();

See also
XMLSocket.connect()
XMLSocket.connect()

Availability
Flash Media Server 2.

Usage
myXMLSocket.connect(host, port)

Parameters
host  A string; a fully qualified DNS domain name or an IP address in the form
aaa.bbb.ccc.ddd. You can also specify null to connect to the local host.
port  A number; the TCP port number on the host used to establish a connection. The port
number must be 1024 or greater.

Returns
A Boolean value; true if successful, false if not.

Description
Method; establishes a connection to the specified Internet host using the specified TCP port
(must be 1024 or higher), and returns true or false, depending on whether a connection is
successfully established. If you don’t know the port number of the Internet host computer,
contact your network administrator.
If you specify null for the host parameter, the local host is contacted.
The server-side ActionScript XMLSocket.connect() method can connect to computers that
are not in the same domain as the SWF file.

NOTE
The client-side ActionScript XMLSocket.connect() method has restrictions. For more
information, see the XMLSocket class in ActionScript 2.0 Language Reference.

If XMLSocket.connect() returns a value of true, the initial stage of the connection process is
successful; later, the XMLSocket.onConnect handler is invoked to determine whether the final
connection succeeded or failed. If XMLSocket.connect() returns false, a connection could
not be established.
Example

The following example uses `XMLSocket.connect()` to connect to the local host:

```actionscript
var socket = new XMLSocket();
socket.onConnect = function (success) {
    if (success) {
        trace("Connection succeeded!")
    } else {
        trace("Connection failed!")
    }
};
if (!socket.connect(null, 2000)) {
    trace("Connection failed!")
}
```

See also

`XMLSocket.onConnect`

### `XMLSocket.onClose`

#### Availability

Flash Media Server 2.

#### Usage

```
myXMLSocket.onClose = function() {}
```

#### Parameters

None.

#### Returns

Nothing.

#### Description

Event handler; invoked only when an open connection is closed by the server. The default implementation of this method performs no actions. To override the default implementation, you must assign a function containing custom actions.

**NOTE**

In Flash Media Server, the output of `trace()` statements appears in the application log file and Application inspector.
Example
The following example executes a `trace()` statement if an open connection is closed by the server:

```javascript
var socket = new XMLSocket();
socket.connect(null, 2000);
socket.onClose = function () {
    trace("Connection to server lost.");
}
```

**NOTE** In Flash Media Server, the output of `trace()` statements appears in the application log file and Application inspector.

See also
XMLSocket.onConnect

### XMLSocket.onConnect

**Availability**
Flash Media Server 2.

**Usage**

```javascript
myXMLSocket.onConnect = function(success) {}
```

**Parameters**

- `success` A Boolean value indicating whether a socket connection is successfully established (`true` or `false`).

**Returns**
Nothing.

**Description**
Event handler; invoked by Flash Media Server when a connection request initiated through `XMLSocket.connect()` has succeeded or failed. If the connection succeeded, the `success` parameter is `true`; `false` otherwise.

The default implementation of this method performs no actions. To override the default implementation, you must assign a function containing custom actions.
Example
The following example defines a function for the `onConnect` handler:

```javascript
socket = new XMLSocket();
socket.onConnect = myOnConnect;
socket.connect(null,2000);

function myOnConnect(success) {
  if (success) {
    trace("Connection success")
  } else {
    trace("Connection failed")
  }
}
```

See also
`XMLSocket.connect()`

**XMLSocket.onData**

**Availability**
Flash Media Server 2.

**Usage**
`myXMLSocket.onData = function(src) {}`

**Parameters**
- **src** A string containing the data sent by the server.

**Returns**
Nothing.

**Description**
Event handler; invoked when a message has been downloaded from the server, terminated by a zero (0) byte. You can override `XMLSocket.onData` to intercept the data sent by the server without parsing it as XML. This is useful if you're transmitting arbitrarily formatted data packets and you'd prefer to manipulate the data directly when it arrives, rather than have Flash Media Server parse the data as XML.

By default, the `XMLSocket.onData` method invokes the `XMLSocket.onXML` method. If you override `XMLSocket.onData` with custom behavior, `XMLSocket.onXML` is not called unless you call it in your implementation of `XMLSocket.onData`. 
Example
In the following example, the `src` parameter is a string containing XML text downloaded from the server. The zero (0) byte terminator is not included in the string.

```javascript
XMLSocket.prototype.onData = function (src) {
    this.onXML(new XML(src));
}
```

**XMLSocket.onXML**

**Availability**
Flash Media Server 2.

**Usage**
```javascript
myXMLSocket.onXML = function(object) {};
```

**Parameters**
- `object` An XML object that contains a parsed XML document received from a server.

**Returns**
Nothing.

**Description**
Event handler; invoked by Flash Media Server when the specified XML object containing an XML document arrives through an open XMLSocket connection. An XMLSocket connection can be used to transfer an unlimited number of XML documents between the client and the server. Each document is terminated with a zero (0) byte. When Flash Media Server receives the zero byte, it parses all the XML received since the previous zero byte or since the connection was established if this is the first message received. Each batch of parsed XML is treated as a single XML document and passed to the `onXML` event.

The default implementation of this method performs no actions. To override the default implementation, you must assign a function containing actions that you define.

**Example**
The following function overrides the default implementation of the `onXML` method in a simple chat application. The function `myOnXML` instructs the chat application to recognize a single XML element, `MESSAGE`, in the following format:

```xml
<Message User="John" Text="Hello, my name is John!"/>
```

```javascript
var socket = new XMLSocket();
```
The following `displayMessage()` function is assumed to be a user-defined function that shows the message that the user receives:

```javascript
socket.onXML = function (doc) {
    var e = doc.firstChild;
    if (e != null && e.nodeName == "MESSAGE") {
        displayMessage(e.attributes.user, e.attributes.text);
    }
}
```

### XMLSocket.send()

**Availability**
Flash Media Server 2.

**Usage**

`myXMLSocket.send(object)`

**Parameters**

`object` An XML object or other data to transmit to the server.

**Returns**

Nothing.

**Description**

Method; converts the XML object or data specified in the `object` parameter to a string and transmits it to the server, followed by a zero (0) byte. If `object` is an XML object, the string is the XML textual representation of the XML object. The send operation is asynchronous; it returns immediately, but the data may be transmitted later. The `XMLSocket.send()` method does not return a value that indicates whether the data is successfully transmitted.

If the `myXMLSocket` object is not connected to the server (using `XMLSocket.connect()`), the `XMLSocket.send()` operation fails.

**Example**

The following example shows how you can specify a user name and password to send the XML object `my_xml` to the server:

```javascript
var myXMLSocket = new XMLSocket();
var my_xml = new XML();
var myLogin = my_xml.createElement("login");
myLogin.attributes.username = usernameTextField;
myLogin.attributes.password = passwordTextField;
my_xml.appendChild(myLogin);
myXMLSocket.send(my_xml);
```
XMLStreams class

Availability
Flash Media Server 2.

Description
The XMLStreams class is a variation of the XMLSocket class—it has all the same methods, properties, and events, but it transmits and receives data in fragments. To create an XMLStreams object, use the XMLSocket constructor and pass "stream" as the parameter. See “Constructor for the XMLSocket class” on page 219.

The Flash Media Server can transmit XML data in stream format (for example, as needed by a Jabber server or IM applications). Streaming XML data passes over a normal XMLSocket connection, but it begins with a stream:stream tag, contains fragments of XML content, and concludes with a /stream:stream closing tag.

The onData event is invoked and returns complete XML tags whenever it receives them. The /stream:stream tag closes the stream. There is an asynchronous call to onData whenever a complete tag has been received by the stream.

NOTE As a security precaution, if 4 KB of data arrives before a closing XML tag, the incoming data is thrown away. This should not be a common concern—at least one complete XML tag should arrive within 4 KB.

Example
If you want your FMS application to communicate with a Jabber server, which uses XML streaming, create an XMLStreams object. The XMLStreams object connects to a remote XML streaming server, and the onData event is called as complete sections of XML occur in the stream.

myXMLStreams = new XMLSocket("stream");
The Application, NetConnection, and Stream classes provide an onStatus event handler that uses an information object for providing information, status, or error messages. To respond to this event handler, you must create a function to process the information object, and you must know the format and contents of the information object returned.

You can define the following global function at the top of your main.asc file to display all the status messages for the parameters that you pass to the function. You need to place this code in the main.asc file only once.

```javascript
function showStatusForClass(){
    for (var i=0;i<arguments.length;i++){
        arguments[i].prototype.onStatus = function(infoObj){
            trace(infoObj.code + " (level:" + infoObj.level + ")");
        }
    }
}
showStatusForClass(NetConnection, Stream);
```

For more information about information objects, see the appendix of the Client-Side Communication ActionScript Language Reference for Flash Media 2.

An information object has the following properties: level, code, description, and details. All information objects have level and code properties, but only some have the description and/or details properties. The following tables list the code and level properties as well as the meaning of each information object.
## Application information objects

The following table lists the information objects for the Application object.

<table>
<thead>
<tr>
<th>Code</th>
<th>Level</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application.Script.Error</td>
<td>Error</td>
<td>The ActionScript engine has encountered a runtime error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In addition to the standard infoObject properties, the following properties are set:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>filename: name of the offending ASC file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lineno: line number where the error occurred.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>linebuf: source code of the offending line.</td>
</tr>
<tr>
<td>Application.Script.Warning</td>
<td>Warning</td>
<td>The ActionScript engine has encountered a runtime warning.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In addition to the standard infoObject properties, the following properties are set:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>filename: name of the offending ASC file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lineno: line number where the error occurred.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>linebuf: source code of the offending line.</td>
</tr>
<tr>
<td>Application.Resource.LowMemory</td>
<td>Warning</td>
<td>The ActionScript engine is low on runtime memory. This provides an opportunity for the application instance to free some resources or take suitable action.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the application instance runs out of memory, it is unloaded and all users are disconnected. In this state, the server will not invoke the Application.onDisconnect event handler or the Application.onAppStop event handler.</td>
</tr>
<tr>
<td>Application.Shutdown</td>
<td>Status</td>
<td>This information object is passed to the onAppStop handler when the application is being shut down.</td>
</tr>
<tr>
<td>Application.GC</td>
<td>Status</td>
<td>This information object is passed to the onAppStop event handler when the application instance is about to be destroyed by the server.</td>
</tr>
</tbody>
</table>
NetConnection information objects

The NetConnection class has the same information objects as the client-side NetConnection class.

<table>
<thead>
<tr>
<th>Code</th>
<th>Level</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetConnection.Call.Failed</td>
<td>Error</td>
<td>The NetConnection.call method was not able to invoke the server-side method or command.*</td>
</tr>
<tr>
<td>NetConnection.Connect.AppShutdown</td>
<td>Error</td>
<td>The application has been shut down (for example, if the application is out of memory resources and must shut down to prevent the server from crashing) or the server has shut down.</td>
</tr>
<tr>
<td>NetConnection.call.BadVersion</td>
<td>Error</td>
<td>The URI specified in the NetConnection.connect method did not specify &quot;rtmp&quot; as the protocol. &quot;rtmp&quot; must be specified when connecting to Flash Communication Server.</td>
</tr>
<tr>
<td>NetConnection.Connect.Closed</td>
<td>Status</td>
<td>The connection was closed successfully.</td>
</tr>
<tr>
<td>NetConnection.Connect.Rejected</td>
<td>Error</td>
<td>The client does not have permission to connect to the application, the application expected different parameters from those that were passed, or the application name specified during the connection attempt was not found on the server.†</td>
</tr>
<tr>
<td>NetConnection.Connect.Success</td>
<td>Status</td>
<td>The connection attempt succeeded.</td>
</tr>
</tbody>
</table>

* This information object also has a description property, which is a string that provides a specific reason for the failure.
† This information object also has an application property, which contains the value that the application.rejectConnection server-side method returns.
Stream information objects

The information objects of the Stream class are similar to those of the client-side NetStream class.

<table>
<thead>
<tr>
<th>Code</th>
<th>Level</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NetStream.Clear.Success</td>
<td>Status</td>
<td>A recorded stream was deleted successfully.</td>
</tr>
<tr>
<td>NetStream.Publish.Start</td>
<td>Status</td>
<td>An attempt to publish was successful.</td>
</tr>
<tr>
<td>NetStream.Publish.BadName</td>
<td>Error</td>
<td>An attempt was made to publish a stream that is already being published by someone else.</td>
</tr>
<tr>
<td>NetStream.Failed</td>
<td>Error</td>
<td>An attempt to use a Stream method failed.*</td>
</tr>
<tr>
<td>NetStream.Unpublish.Success</td>
<td>Status</td>
<td>An attempt to unpublish was successful.</td>
</tr>
<tr>
<td>NetStream.Record.Start</td>
<td>Status</td>
<td>Recording was started.</td>
</tr>
<tr>
<td>NetStream.Record.NoAccess</td>
<td>Error</td>
<td>An attempt was made to record a read-only stream.</td>
</tr>
<tr>
<td>NetStream.Record.Stop</td>
<td>Status</td>
<td>Recording was stopped.</td>
</tr>
<tr>
<td>NetStream.Record.Failed</td>
<td>Error</td>
<td>An attempt to record a stream failed.</td>
</tr>
<tr>
<td>NetStream.Play.InsufficientBW</td>
<td>Warning</td>
<td>Data is playing behind the normal speed.***</td>
</tr>
<tr>
<td>NetStream.Play.Start</td>
<td>Status</td>
<td>Play was started.†</td>
</tr>
<tr>
<td>NetStream.Play.StreamNotFound</td>
<td>Error</td>
<td>An attempt was made to play a stream that does not exist.</td>
</tr>
<tr>
<td>NetStream.Play.Stop</td>
<td>Status</td>
<td>Play was stopped.</td>
</tr>
<tr>
<td>NetStream.Play.Failed</td>
<td>Error</td>
<td>An attempt to play back a stream failed.**</td>
</tr>
<tr>
<td>NetStream.Play.Reset</td>
<td>Status</td>
<td>A playlist was reset.</td>
</tr>
<tr>
<td>NetStream.Play.PublishNotify</td>
<td>Status</td>
<td>The initial publish to a stream was successful. This message is sent to all subscribers.</td>
</tr>
<tr>
<td>NetStream.Play.UnpublishNotify</td>
<td>Status</td>
<td>An unpublish from a stream was successful. This message is sent to all subscribers.</td>
</tr>
</tbody>
</table>

* This information object also has a description property, which is a string that provides a specific reason for the failure.
† This information object also has a details property, which is a string that provides the name of the streams being played. This is useful for multiple plays. The details property shows the name of the stream when switching from one element in the playlist to the next element.

*** By default, outgoing messages are monitored at 5 second intervals; if data is behind, the client is notified within 5 seconds. The interval for sampling can be configured in the Application.xml file in the Client tag.