



CUEScript Developer's Guide

CLEO

RESTRICTED RIGHTS

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (C)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Manufacturer is:

Cleo Communications

4203 Galleria Drive
Rockford, IL 61111 USA
Phone: 815.654.8110
Fax: 815.654.8294
Email: sales@cleo.com
www.cleo.com

Support: 1.866.444.2536 or support@cleo.com

Cleo Communications reserves the right to, without notice, modify or revise all or part of this document and/or change product features or specifications and shall not be responsible for any loss, cost or damage, including consequential damage, caused by reliance on these materials.

This document may not be reproduced, stored in a retrieval system, or transmitted, in whole or in part, in any form or by any means (electronic, mechanical, photo-copied or otherwise) without the prior written permission of Cleo Communications.

©2008 CLEO COMMUNICATIONS ALL RIGHTS RESERVED. CLEO IS A REGISTERED TRADEMARK OF CLEO COMMUNICATIONS. ALL OTHER BRAND NAMES USED ARE TRADEMARKS OR REGISTERED TRADEMARKS OF THEIR RESPECTIVE COMPANIES.

©2008 CUESCRIPT IS A REGISTERED TRADEMARK OF CUESOFT.COM, INC.

Table of Contents

Overview	5
Basics	5
Script Language	6
Program Structure.....	6
Multiple Script Sections	6
Operators	6
Built-in Functions.....	7
Variables and Scope	9
Expressions.....	10
Translate Text.....	10
Conditional Expressions	10
Arrays	11
Built-In Objects	11
System Object	11
Document Object	12
Output Object	12
Scripting Elements	14
Scripting Run Elements.....	14
Script (<script>)	14
Stop (<stop>).....	14
External Resources	14
Assembly (<assembly>).....	14
Library (<library>).....	15
Variables and Expressions	16
Variable (<var>).....	16
Expression (<expr>).....	16
Conditional Elements	16
If (<if>).....	16
Else (<else>)	17
Switch, Case, Default (<switch>,<case>,<default>).....	17
Looping.....	18
For (<for>)	18
Foreach (<foreach>)	18
While (<while>)	18
Repeat Until (<repeat>).....	19
Continue (<continue>)	19
Break (<break>).....	19
Exception Handling.....	19
Try (<try>).....	20
Catch (<catch>)	20
Finally (<finally>)	20
Raise (<raise>).....	21
Reraise (<reraise>)	21
Functions	21

Function (<function>).....	22
Parameter (<parameter>)	22
Return (<return>).....	22
Events	23
Event (<event>).....	23
Document Manipulation	24
Inline (<inline>)	24
Merge (<merge>)	24
Database Scripting Extension.....	25
DbConnection (<dbconnection>).....	25
DbQuery (<dbquery>)	26
DbExec (<dbexec>).....	27
Using the Script Engine	29
Extending the Script Engine	30

Overview

Basics

CUEScript is an XML-based scripting language. CUEScript supports all the standard features of programming languages: variables with scope, looping, functions, events, and exception handling. In addition, CUEScript can invoke .NET runtime objects. The XML tags that constitute CUEScript can be extended by simply adding .NET assemblies. CUEScript is more powerful and easier to use than XSLT because it is declarative and linear. CUEScript also supports XSLT transforms through a special tag, so you can mix and match based on your requirements.

CUEScript is parsed and interpreted at runtime and calls object members on a late-binding basis. *Liaison* is the stand-alone command line script processing application. CUEScript can also be embedded directly in an application by creating an instance of the *ScriptEngine* class and executing the script programmatically.

Script Language

Program Structure

A CUEScript is a simple XML document. The root element is `<script>`.

```
<?xml version="1.0" encoding="utf-8" ?>
<script xmlns='http://www.cuesoft.com/CUEScript/1.0'>
  <!-- Your application goes here. -->
</script>
```

The script tag declares the default namespace this script. CUEScript sections are defined with the namespace `http://www.cuesoft.com/CUEScript/1.0`. Many XML editors can provide help by accessing the schema online. Since CUEScript is defined by its own namespace, script sections can be embedded within other XML documents.

```
<authors xmlns:cs='http://www.cuesoft.com/CUEScript/1.0'>
  <author first="Edgar" last="Poe"></author>
  <cs:script name='main'>
    <cs:output>Hello from the main script section.</cs:output>
  </cs:script>
  <author first="John" last="Steinbeck"></author>
  <cs:script>
    <cs:output>This section does not execute.</cs:output>
  </cs:script>
</authors>
```

The document element of this script is not part of any namespace, but it declares the CUEScript namespace and assigns a 'cs' prefix to it. The script tags are then included within the document.

Multiple Script Sections

A CUEScript can contain multiple script sections. If no script section has been marked as *main*, then each section will execute in document order. When a section is marked as *main* (using the name attribute), then only that section will execute. You may override this behavior by selecting a section to execute on the liaison command line (see section Using Liaison) or `ScriptEngine.Execute()` member (See section Programmer's Guide).

Operators

CUEScript supports a small number of language operators. CUEScript supports native XPath syntax, so its operators are different from XPath operators.

Operator	Description	Example
\$	Variable reference. The dollar sign is used to indicate the next token is a variable.	\$index := 0
:=	Assignment (colon equals).	\$index := 0
.	The dot operator functions like it does in .NET. It is	\$directory.Name()

	used invoke a member on an object.	
^	String concatenation	<code>\$fullName := \$firstName ^ \$lastName;</code>
{{ }}	Embedded expression. The nested expression is evaluated and substituted in place.	<code>Arg1={{\$0}}</code>
;	Statement terminator.	CUEScript expression tags can support multiple statements. The semicolon is used to terminate a single statement.
div, mod	Integer division and modulus.	<code>\$dollars := \$total div 100; \$cents := \$total mod 100;</code>
and, or	Logical operations	
XPath operators	CUEScript is built in over XPath. All xpath operators are valid in CUEScript.	<code>\$nodes := /authors/author[@last='Poe'];</code>

Built-in Functions

CUEScript adds functions to the basic XPath function set.

Function	Description	Example
<code>chr()</code>	Returns the character for the passed numeric reference.	<code>\$newline := chr(10);</code>
<code>contains()</code>	Returns true if the second argument contains the first argument.	<code><if test="contains('Steve', 'eve')"> </if></code>
<code>datetime()</code>	Convert the passed string value to a date time object. Returns <code>DateTime.Now</code> if nothing is passed or the value fails to parse.	<code>\$now := datetime(); \$then := datetime('3/6/2008');</code>
<code>decimal()</code>	Converts the passed value to a decimal value. Useful for converting a decimal value to a string value.	<code>\$pi := '3.1416'; \$piValue := decimal(\$pi);</code>
<code>decrypt()</code>	Decrypts the passed string using a built-in key. Callers may optionally provide a key as a second parameter.	<code>\$node.InnerText := decrypt(\$node.InnerText);</code>
<code>encrypt()</code>	Encrypts the passed string using a built-in key. Callers may optionally provide a key as a second parameter.	<code>\$node.InnerText := encrypt(\$node.InnerText);</code>
<code>enum()</code>	Resolves a .NET enumerated type name into an enumerated value.	<code>\$enumValue := Enum("DialogResult.Cancel")</code>
<code>format()</code>	Produces formatted output like the .NET <code>String.Format()</code> method.	<code>\$element.SetAttribute('fullName', format('{0}{1}', \$first, \$last));</code>
<code>hex()</code>	Converts the passed argument to a hexadecimal representation. The resulting value is a string.	<code>\$value := 128; \$hexValue := hex(\$value);</code>
<code>isnull()</code>	Returns true if the passed variable reference is null.	<code><var name='node' expr="tonode('\doc/foo')"/> <if test='isnull(\$node)' </if></code>
<code>isnullorempty()</code>	Returns true if the passed string reference is null or	<code><function name='ShowString'></code>

	has zero length.	<pre> <parameter name='str' /> <if test='isnullorempty(\$str)'> <output>String is null or empty.</output> </if> <else> <output>{{\$str}}</output> </else> </function> </pre>
isnumber()	Returns true if the passed object is numeric.	<pre> \$count := 37; \$strCount := '37'; <if test='isnumber(\$count)'> <output>count is a number.</output> </if> <if test='not(isnumber(\$strCount)) '> <output>strCount is a string that looks like a number.</output> </if> </pre>
isvariable()	Returns true if the passed string is the name of a defined variable.	isvariable("index")
new()	Creates a new .NET runtime object of the passed type name. The first argument to new is the class name. Any subsequent arguments become arguments to the object's constructor.	<pre> \$event := new('System.Threading.ManualReset Event') </pre>
null()	Returns the null constant value.	<pre> \$node := tonode('/doc/foo'); <if test='\$node=null()'> </if> </pre>
rgb()	Creates an RGB triple from the passed color numbers.	\$red := rgb(255,0,0);
starts-with()	Returns true if the first string parameter starts with the second passed parameter. This is case sensitive.	\$steve := starts-with(\$name, 'Steve');
timespan()	Creates a timespan from the passed string or returns TimeSpan.Zero.	<pre> \$now := timespan(); \$then := timespan('0.00:03:00'); </pre>
tonode()	Returns the first node from a node list. Since The xpath expression could return multiple nodes, you should use this function to return first node. If there are no nodes in the list, this function returns null.	\$edward := tonode(/authors/author[@last='Poe'])
type(typeName, object)	Convert an object from one type to another type. This should only be used on primary types (i.e. integers, string, characters, etc). The first parameter is the .NET type name to convert to. The second parameter is the object to convert. The result is the converted object or null on failure.	\$char := type("System.Char", "X");

Variables and Scope

CUEScript variable names follow the standard programming naming conventions (alphanumeric, not starting with digits). Variables can either be declared or simply used. Variables are scoped based on where they are declared or first used. Variable names are case sensitive.

To declare a variable, use the `<var>` tag:

```
<var name="index" expr="0"/>
<var name="author" expr="' Jones'"/>
```

The variable declaration may contain the variable scope: global, session, or application. Global scope makes the variable available to any function in the script or at any scope level. Session scope makes the variable available to other scripts running in the same session (availability of session scope depends on CUEScript implementation). Application scope makes the variable available to other scripts for the lifetime of the application (availability of application scope depends on the CUEScript implementation). If session or application scopes are not supported in the implementation, the result of selecting those scopes will be global.

```
<var name='index' expr='0' scope='global' />
```

The programmer should be careful of using global variables and local variables with the same name as unexpected and difficult to debug behavior may result. See the `<var>` element reference for more information.

```
<script>
  <var name="firstName" expr="'Steve'"/>
  <if test='true()'>
    <var name="firstName" expr="'Alex'"/>
    <expr>
      $lastName := 'Smith';
    </expr>
  </if>
  <if test="$firstName='Alex'">
  </if>
</script>
```

Any script tag that has child tags creates a new variable context (with the exception of the `<expr>` tag). Any variables that are then declared or created live within that context. When the child elements are finished executing, the context is deleted and those variables are lost. Variables that are defined by initial use within an `<expr>` tag have the scope of the `<expr>` tag's parent tag. In the above example, the variable 'firstName' gets declared in the 'script' context. Within the 'if' tag, a new copy of 'firstName' gets created. When the script leaves the 'if', the variable instance `firstName='Alex'` is removed leaving `firstName='Steve'`. The variable 'lastName' is created without a declaration and will only live while the 'if' statement executes.

This scoping feature is very powerful and allows you to reuse index variables within nested loops. It can, however, cause some confusion as to which variable instance you are modifying.

Expressions

Expressions consist of a combination of operators to perform some function. These operators consist of a mixture of XPath operators along with CUEScript native operators. The semi-colon is used to separate expressions where they can occur. The table below lists some different types of expressions.

Type	Examples
XPath	<code>//authors</code> <code>//authors[@lastName='Poe']</code> <code>//authors[@lastName='Poe']/@lastName</code> <code>count(//authors)</code>
Expression	<code>2 + 2</code> <code>\$lastName ^ ", " ^ \$firstname</code> <code>tonode(//authors[@lastName='Poe']).SetAttribute("firstName", "Edgar")</code>
Assignment	<code>\$myauthor := tonode(//authors[@lastName='Poe'])</code> <code>\$fullName := \$lastName ^ ", " ^ \$firstname</code>
Function Call	<code>factorial(\$somenumber)</code> <code>rgb(255, 0, 0);</code>

Translate Text

Several of the CUEScript element attributes and text entries support a feature called “translate text”. This allows CUEScript expressions to be inserted within expressions or text by surrounding the expression with `{{}}`. This is a useful way of inserting the results of expressions within other text or even other expressions. If occurring within another expression, sub-expressions within `{{}}` are evaluated first and then become part of the outer expression.

For example:

```
<var name="msg">'Mismatched appliance ids, expected {{${security.JobApplianceId}}, has {{${security.ApplianceId}}}'</var>

<expr>
  $mynodes := /document/datanodes[ @{{${relevantAttribute}}};
</expr>
```

Conditional Expressions

CUEScript support conditional shorthand expressions that are useful for conditional assignments, tests, or simple if/else statements. The notation for this conditional expressions is:

`condition ? (expression), (expression)`

If the condition is true, then the first expression is run, otherwise the second expression is run.

For example:

```
<expr>
  $message := $isImportant ? ( "This message is important" ), ( "Not so important" );
  $output.Write( $message );
  $person = 'George Bush' ? ( $music := presidentialTheme() ), ( $music := standardTheme() );
  play( $music );
</expr>
```

Arrays

CUEScript allows you to define arrays using the [] notation.

```
$someVariable := [ expression1, expression2, expression3, ... ];
```

For example:

```
<expr>
    $days := [ "Sun", "Mon", "Tue", "Wed", "Thur", "Fri", "Sat" ];
</expr>
```

Once defined, arrays can be enumerated using the "foreach" tag or directly indexed.

For example:

```
<expr>
    $thursAbbr := $days[4];
</expr>
```

For adding and removing objects to an array within script, we recommend using the .NET ArrayList or List<T> object implementations. Here is an example of using an ArrayList to add elements to the list and then convert that to an array object.

```
<module xmlns="http://www.cuesoft.com/CUEScript/1.0">
    <script name="main">
        <expr>
            $list := new( "System.Collections.ArrayList" );
            $list.Add( "hello" );
            $list.Add( "world" );
            $output.WriteLine( $list.Count );

            $listAsArray := $list.ToArray();
            $output.WriteLine( $listAsArray[0] );
        </expr>
    </script>
</module>
```

Built-In Objects

Every script has access to a group of built-in objects. These objects provide common functionality and access to other system resources.

System Object

The system object is a global variable within every script. You access it using the lowercase name: *\$system*.

PlaySound(string soundFile)	Plays the sound file passed as an argument
PlaySound(int defaultSound)	Plays the sound associated with the message box button.
Sleep(int milliseconds)	Causes the current thread to sleep for the indicated number of milliseconds.

<pre>Wait(int timeout) Wait(int timeout, string eventName)</pre>	<p>Waits for the specified amount of time for any event to be fired. Returns true if an event fired, false otherwise. The second version waits for a specific event name.</p>
<pre>SubscribeToEvent(object sourceObject, string clrEventName, string scriptEventName)</pre>	<p>Subscribes to a .NET event on an object. See the description of the event element for more details.</p>
<pre>UnsubscribeToEvent(object sourceObject, string clrEventName)</pre>	<p>Removes a subscription to a .NET event.</p>
<pre>RaiseEvent(string name, EventArgs e)</pre>	<p>Manually raises an event of a particular name.</p>
<pre>CreateNetObject(string typeName) CreateNetObject(string typeName, params object[] args)</pre>	<p>Creates a .NET object of the given type name. The arguments are passed to the constructor.</p>
<pre>CreateComObject(string objectId)</pre>	<p>Creates a Component Object Model (COM) object. The objectId can be a CLSID or ProgId.</p>
<pre>CreateObject(string name)</pre>	<p>Creates internal CUEScript objects or .NET runtime objects with parameterless constructors. Pass "document" to create a new CUEScript document, for instance.</p>

Document Object

CUEScript supports self modifying scripting. In order for a script to access itself, it uses either \$document or \$thisscript. The returned object is a script document derived from the .NET XmlDocument class. You can use the .NET XPath implementation (SelectSingleNode/SelectNodes), or you can use the CUESoft XPath implementation and simply apply XPath notation to expressions. CUEScript understands XPath inherently.

.NET way:

```
$poeNode := $document.SelectSingleNode( '/authors/author[@lastName='Poe'] )
```

Or the CUEScript way:

```
$poeNode := tonode(/authors/author[@lastName='Poe'])
```

You have full access to the other Xml document members. The CUESoft Script document class has the following additions.

<pre>MergeXml(string textToParse, XmlNode refNode)</pre>	<p>MergeXml parses new Xml markup into an existing document, appending the result to the reference node. It is the programmer's responsibility to ensure that the result is well formed.</p>
<pre>Save()</pre>	<p>Save changes to the document using the same file name it was loaded with.</p>

Output Object

The \$output object allows you to access an output stream. In liaison, this output stream is the console by default, but it can be redirected to a file, or any other output stream. When CUEScript is used for web applications, the output can be set as the Http response stream making anything written to it available in the client browser. The \$output object is always a .NET StreamWriter.

Scripting Elements

The following sections define the XML scripting elements in the CUEScript namespace. Use the following key to determine the value type of the field:

(lit) = Literal text value (i.e. someValue)

(ex) = Expression (i.e. 'someValue')

(tt) = Translate text (i.e. some{{ \$value}})

(ttex) = Translate text expression (i.e. 'some{{ \$value}}')

Scripting Run Elements

The *script* element is the root of every CUEScript script section. The *stop* element will quit script execution.

Script (<script>)

Description	Defines a script section.
Attributes	
name	<i>(lit)</i> Optional name of the script section. The only name that is reserved is 'main'. If a section is marked main, it is the only section that executes. This behavior can be overridden on the liaison command line or ScriptEngine.Execute() method call.
Example	<pre><script name='main' xmlns='http://www.cuesoft.com/CUEScript/1.0'> <!-- Other CUEScript goes here --> </script></pre>

Stop (<stop>)

Description	Stops script execution. If the script executes a stop, the script will terminate with a Stop status. The stop will not override an exception. If the script engine has raised an exception that is not caught, the script will terminate with an Exception status even if it hits a stop element.
Example	<pre><if test='\$fatalError = true()'> <!-- Stop script processing immediately --> <stop/> </if></pre>

External Resources

CUEScript's extensibility allows outside resources to be used in scripts. The *assembly* command is used to reference external .NET assemblies for use in script. The *library* command allows other script document resources to be included in the script.

Assembly (<assembly>)

Description	Loads the indicated assembly and makes the script objects and tags available to the script
--------------------	--------------------------------------------------------------------------------------------

	engine. If the file name is passed, then the assembly is loaded for tags and objects. This operation occurs when the script document is parsed making the tags available for the remainder of the document. If the reference attribute is supplied, then the assembly is loaded when the script executes.
Attributes	
file	<i>(lit)</i> The full path name of the assembly file. (optional) The file attribute causes the assembly to be loaded when the script is parsed.
reference	<i>(lit)</i> The full path name of the assembly file. (optional). The reference attribute causes the assembly to be loaded at runtime.
scope	<i>(lit)</i> Optionally indicates the scope where the script objects in the assembly should be created. By default, the objects are created in the current context. Specify 'global' to make those objects global. See the Programmer's Reference for more details on extending CUEScript.
namespace	<i>(lit)</i> Optionally define a namespace name for the objects in this assembly.
Example	<pre><script name='main' xmlns='http://www.cuesoft.com/CUEScript/1.0'> <assembly file="MyAssembly.dll"/> <expr> \$myObject := new("CUESoft.MyAssy.MyObject"); \$myObject.DoMyThing(); </expr> </script></pre>

Library (<library>)

Description	The library tag allows one script to include a reference to a second script. Libraries are generally declared at the top of the including script. Libraries can be preloaded or loaded on demand meaning they are only loaded if you call a function that resides in the library.
Attributes	
file	<i>(lit)</i> The file name for the library. The script resolver uses this name to load the file. The default resolver attempts to load this file from the current directory.
name	<i>(lit)</i> The library's symbolic name. The name is used as a prefix when calling a function within the library. A pound sign separates the library name from the function name when calling an external function.
cache	<i>(lit)</i> Optionally indicates if this library can be cached. If this attribute is 'false', then the library will always be reloaded and never pulled from a local cache. When true, then the file can be pulled from a cache depending on the expiration rules.
preload	<i>(lit)</i> Optionally indicates if you want the script load when the library element executes or to wait until it is needed. The default is false.
expires	<i>(lit)</i> Optional. This applies to libraries that are cached. If a library is cached, the 'expires' attribute is used to specify how long a library is valid before it is reloaded. This value is parsed as a .NET TimeSpan.
Example	<pre><script name='main'> <library file="Connections.xml" name="Connections" preload="true" expires="08:00:00"/> <expr> \$myObject := Connections#GetObject(); </expr> </script></pre>

Variables and Expressions

Variable (<var>)

Description	Defines a variable in the current or specified context. The value of the variable retrieved from the 'expr' attribute or the inner text of the element.
Attributes	
name	<i>(tt)</i> The name of the variable.
expr	<i>(ttex)</i> The expression to store in the variable.
scope	<i>(tt)</i> The scope of where to store the variable. Valid values for scope are "global", "application", and "session". If no scope is specified, the variable is added in the local scope.
revoke	<i>(lit)</i> Set to 'true' to remove the variable from the scope.
Example	<pre><var name='index' expr='100' /> or <var name='index'>100</var></pre>

Expression (<expr>)

Description	Used to include expressions. The expression itself is either the inner text (<i>ex</i>) of the 'expr' element or specified using the 'value' attribute.
Attributes	
value	<i>(ex)</i> The expression to evaluate.
Example	<pre><if test="\$outputReport"> <expr> \$output.WriteLine("Total Hours = " ^ \$hours); \$output.WriteLine("Total Dollars = " ^ \$totalAmount); <expr> </if></pre>

Conditional Elements

CUEScript supports the conditional statements, *if*, *else* and *switch*.

If (<if>)

Description	<p>Starts a conditional block. The child elements of the if element are executed if the test condition is true.</p> <p>Note that you will have to escape the "<" when you want to represent less than. For XML this is &lt;. See the example below.</p>
--------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Attributes	
test	(<i>ttex</i>) The condition used to determine if the child elements should execute. This attribute is required.
Example	<pre> <var name='\$index' expr='100' /> <if test='\$index >= 10'> <output>Index is big</output> </if> <if test='\$index &lt; 10'> <output>Index is small.</output> </if> </pre>

Else (<else>)

Description	Else is a sibling block to the if statement. It can contain a test condition making it an else/if. If no condition is specified, then it will execute when the if condition evaluates false.
Attributes	
test	(<i>ttex</i>) The condition used to determine if the child elements should execute. This attribute is required.
Example	<pre> <var name='\$index' expr='100' /> <if test='\$index > 50'> <output> Index is big</output> </if> <else test='\$index > 30'> <output>Index is medium</output> </else> <else> <output>Index is small</output> </else> </pre>

Switch, Case, Default (<switch>,<case>,<default>)

Description	When coding a series of conditions, you can use if/else if/else if/else, or combine everything into a switch element. Unlike C-based languages, the case condition in CUEScript does not have to be a constant.
Attributes	
test	(<i>ttex</i>) The condition used to determine which case to execute. This attribute is required.
value (case)	(<i>ttex</i>) The value attribute is required on the case element. This expression is compared against the value of the test attribute on the switch element.
Example	<pre> <var name='\$index' expr='100' /> <switch test='\$index'> <case value='100'> <output>Perfect</output> </case> <case value='0'> <output>Imperfect</output> </case> <default> <output>Somewhere in between</output> </default> </pre>

Looping

CUEScript supports *for*, *foreach*, *while*, and *repeat* loops. Loops can be short circuited by either *continue* or *break* statements.

For (<for>)

Description	Starts a for loop. Generally used to iterate through a known number of items.
Attributes	
init	(<i>ttex</i>) The initial expression
test	(<i>ttex</i>) The condition used to determine if the loop should continue.
step	(<i>ttex</i>) The expression used at the end of the loop. Typically, this expression increments a counter or some other condition which may make the test condition return false.
Example	<pre><for init=' \$index = 0' test=' \$index &lt; 10' step=' \$index := \$index + 1'> <expr> \$output.WriteLine(\$index); <expr> </for></pre>

Foreach (<foreach>)

Description	Used to loop through a collection of items.
Attributes	
var	(<i>tt</i>) The name of the variable to hold each item in the collection.
in	(<i>ttex</i>) The collection to iterate through.
Example	<pre><foreach var='authorElement' in='//authors'> <expr> \$output.WriteLine(\$authorElement.GetAttribute("name")); <expr> </foreach></pre>

While (<while>)

Description	Starts a while loop. Since a while loop evaluates its test condition first, it is possible that the loop will not execute.
Attributes	
test	(<i>ttex</i>) The condition used to determine if the loop should continue.
Example	<pre><var name=' \$index' expr='0' /> <while test=' \$index &lt; 10'> <expr> \$output.WriteLine(\$index); \$index := \$index + 1; <expr> </while></pre>

Repeat Until (<repeat>)

Description	Starts a repeat/until loop. Since a while loop evaluates its test condition last, the loop will always execute at least once.
Attributes	
until	(<i>ttex</i>) The condition used to determine if the loop should continue.
Example	<pre><var name=' \$index' expr='0' /> <repeat until=' \$index = 10'> <expr> \$output.WriteLine(\$index); \$index := \$index + 1; </expr> </repeat></pre>

Continue (<continue>)

Description	The continue element causes the loop to skip the remainder of its body and reevaluate its test condition. The example shows a loop that will print out all the values of index except 5.
Example	<pre><for init=' \$index = 0' test=' \$index &lt; 10' step=' \$index := \$index + 1'> <if test=' \$index=5'> <continue/> </if> <expr>\$output.WriteLine(\$index)</expr> </for></pre>

Break (<break>)

Description	The break element breaks out of one or more loops. In the example, when index equals 5, both the for and while loops will break. If the break element did not have the level attribute, then only the for loop would break.
Attributes	
level	(<i>tt</i>) Optionally specifies how many loops to break out of.
Example	<pre><while test='true()'> <for init=' \$index = 0' test=' \$index &lt; 10' step=' \$index := \$index + 1'> <if test=' \$index=5'> <break level='2' /> </if> <expr>\$output.WriteLine(\$index)</expr> </for> </while></pre>

Exception Handling

CUEScript supports structured exception handling. Exceptions from the operating system, script engine, or scripts can be caught and handled. There are five tags that implement exception handling: *try*, *catch*, *finally*, *raise*, and *reraise*.

Try (<try>)

Description	This element starts a block of code where an exception could occur. In the example below, if the object cannot be created, then an exception will be raised. If the exception is not caught, it will cause the script engine to terminate the script. Try blocks can be nested.
Example	<pre><try> <expr> \$system.CreateObject('Something'); </expr> </try></pre>

Catch (<catch>)

Description	This element catches an exception. The attributes for this element specify which exceptions to catch. If no parameters are specified, then all exceptions are caught.
Attributes	
name	<i>(tt)</i> The optional name of the exception.
code	<i>(tt)</i> The optional numeric code of the exception.
Example	<pre><try> <expr> \$system.ThisThrows(); </expr> </try> <catch name='ScriptSyntax'> <output>Script syntax error.</output> </catch> <output>Some other exception.</output> </catch></pre>

Finally (<finally>)

Description	This element creates a block of code that runs unconditionally at the end of a try block. Use the finally block to close or release resources you have created within the try block. Since finally blocks always execute, you do not have to explicitly release resources in each catch block.
Example	<pre><var name='myFile' expr="\$system.Invoke(\System.IO.File.OpenRead('myFile.txt')"/> <try> <expr> \$lines := \$myFile.ReadAllLines(); </expr> </try> <catch name='ScriptSyntax'> <output>Script syntax error.</output> </catch> <output>Some other exception.</output> </catch></pre>

```
<finally>
  <expr>$myFile.Close()</expr>
</finally>
```

Raise (<raise>)

Description	This element raises an exception from the script.
Attributes	
name	<i>(tt)</i> The name of the exception.
code	<i>(tt)</i> The optional numeric code of the exception.
explanation	<i>(tt)</i> The error text.
Example	<pre><if test='\$index > 10'> <raise name='IndexOutOfRange' explanation='Index is greater than 10.'/> </if></pre>

Reraise (<reraise>)

Description	This element re-raises an exception from a catch block. This allows you to catch and process an exception in more than one place. In the given example, the expression will result in a syntax exception. The first catch block will then execute. Within that block, the exception is reraised. The catch all block that follows WILL NOT execute. Reraising the exception will jump you to the next outer try block; no further catch processing will occur in the current try. The finally element will always execute, however.
Example	<pre><try> <expr> \$system.ThisWillThrow(); </expr> </try> <catch name='ScriptSyntax'> <output>Script syntax error.</output> <reraise/> </catch> <output>Some other exception.</output> </catch> <finally> <output>Moving On</output> </finally></pre>

Functions

CUEScript supports functions and procedures with the function tag. A function is a shared piece of programming logic that can be called multiple times from multiple places. Functions can receive named parameters and optionally return values to the caller. See the library tag for information on using external function libraries.

Function parameters are automatically assigned numeric variables starting with \$0, \$1, \$2, etc. You can refer to these variables inside your function. To assign more reader-friendly names to your parameters, use the <parameter> tag. This tag also allows you to define default values for parameters if they are omitted.

For example, defining a function like:

```
<function name="logFile">
  <parameter name="file"/>
  <parameter name="path" default="'c:\temp'"/>
  <expr>
    $log.Debug( "Logging file " ^ $path ^ "\" ^ $file );
  </expr>
</function>
```

would allow you to call it like:

```
logFile( "myfile.xml", "c:\foo" );
or
logFile( "myfile.xml" );
```

Function (<function>)

Description	This element creates a script function. Script functions can be called from expressions.
Attributes	
name	<i>(lit)</i> The name of the function. This is required.
Example	<pre><function name='CreateNewUser'> </function></pre>

Parameter (<parameter>)

Description	This element declares a function parameter. Parameter declarations may included a default value and are then not required to be passed. As in C++, however, only trailing parameters can be left defaulted. No parameter can be skipped over.
Attributes	
name	<i>(tt)</i> The name of the parameter. This is required.
default	<i>(ttex)</i> Optional. An expression to be used when the parameter is not passed.
Example	<pre><function name='CreateNewUser'> <parameter name='first' /> <parameter name='last' /> <parameter name='permission' default="'guest'"/> <expr> \$systemLogins.Create(\$first, \$last, \$permission); </expr> </function> ... <expr> CreateNewUser('Fred', 'Smith', 'administrator'); CreateNewUser('Tom', 'Jones'); </expr></pre>

Return (<return>)

Description	The return element exits the function. If an expression is specified, the result is returned
--------------------	----------------------------------------------------------------------------------------------

	to the caller. If no return value is specified, the function returns null.
Attributes	
expr	<i>(ttex)</i> An optional expression that specifies the value to return to the caller.
	<i>(ttex)</i> The inner text of the return can be used instead of the 'expr' attribute.
Example	<pre> <function name='GetPi'> <return expr='3.1416' /> </function> ... <expr> \$pi := GetPi(); </expr> </pre>

Events

CUEScript supports event callbacks. Events are special functions that are invoked by objects asynchronously. The syntax for events is similar to functions, but the parameter list is always the same: the sending object and the event arguments. You can invoke members in both the sender and event arguments. They will be the full objects.

Event parameters are numbered like function parameters (\$0 and \$1). The second parameter is also assigned to the variable \$eventArgs, since that is the event argument parameter. You can also assign the <parameter> tag to name the arguments any way you want.

Event (<event>)

Description	This element declares an event handler. Parameter tags are optional.
Attributes	
name	<i>(lit)</i> The name of the parameter. This is required.
Example	<pre> <event name='OnTimerTick'> <parameter name='sender' /> <parameter name='e' /> <expr> \$output.WriteLine("Got tick"); </expr> </name> ... <expr> \$myTimer := new("System.Windows.Forms.Timer"); \$myTimer.Interval := 30000; \$system.SubscribeToEvent(\$myTimer, "OnTick", "OnTimerTick"); \$myTimer.Start(); </expr> </pre>

Document Manipulation

CUEScript supports self-modifying scripting. You can write script that generates or changes itself based on runtime conditions. In addition to XPath and the full document object mode (DOM), CUEScript supports 2 special tags that manipulate the document.

Inline (<inline>)

Description	This element declares an xml inline expression. The inline element is replaced by the expression composed by its child elements. The example shows how you can create XML on-the-fly.
Example	<pre><parent> <cs:inline> <child name="{getName()}" /> </cs:inline> </parent></pre>

Merge (<merge>)

Description	This element converts child text to XML elements.
Example	<pre><parent> <cs:merge> &lt;child/> </cs:merge> </parent></pre>

Database Scripting Extension

CUEScript can be extended programmatically. See the section 'Extending the Script Engine' for details. CUEScript comes with one extension already created for you: SQL database access. The sample below demonstrates the SQL tags and options.

IMPORTANT: Because this is an extension, you'll need to add the assembly reference to the CUEScript SQL assembly (<assembly file="CUESoft.CUEScript.Sql"/>).

```
<script xmlns="http://www.cuesoft.com/CUEScript/1.0">
  <dbconnection name="RMCS"
    source="Server=ARCHIE;Database=RMCS;Integrated Security=true">
    <dbconnection
      source="Server=ARCHIE;Database=Coinstar;Integrated Security=true">
      <dbexec query="SELECT COUNT(*) FROM Country" var="countryCount"/>
      <expr>
        $output.WriteLine( format( "Total number of countries: {0}",
          $countryCount ) )
      </expr>
      <dbexec connection="RMCS" query="SELECT COUNT(*) FROM Scripts"
        var="scriptCount"/>
      <expr>
        $output.WriteLine( format("Total number of scripts: {0}", $scriptCount ) )
      </expr>
      <dbquery connection="RMCS" select="SELECT * FROM Scripts" var="script">
        <expr>
          $output.WriteLine( format( "Name:{0} Id:{1}",
            $script.Name, $script.ScriptId ) )
        </expr>
      </dbquery>
      <dbquery connection="RMCS" select="SelectScripts" var="matchingScripts">
        <param name="@scriptName" value="Interactive"/>
        <expr>
          $output.WriteLine(format( "Matching: {0}", $matchingScripts.Name ) )
        </expr>
      </dbquery>
      <dbexec connection="RMCS" query="ClearJobInstances" var="result"/>
      <expr>
        $output.WriteLine( format( "Result of stored procedure: {0}", $result ) )
      </expr>
    </dbconnection>
  </dbconnection>
</script>
```

DbConnection (<dbconnection>)

Description	This element declares and creates a SQL database connection. The connection is opened when the connection tag executes. The connection closes at the end tag.
Attributes	

source	(<i>tt</i>) The database connection string. The attribute is required.
name	(<i>tt</i>) The connection name. This attribute provides a name for the connection object. When working with multiple connections, you must provide a name on the connection, and specify the connection attribute on each <dbquery> or <dbexec> element. When working with a single connection, no names are required.
Example	<pre><dbconnection source="Server=ARCHIE;Database=RMCS;Integrated Security=true"> <dbquery select="SELECT * FROM Appliances"> </dbquery> </dbconnection></pre>

DbQuery (<dbquery>)

Description	This element declares and a query returning a recordset. The child elements are executed in a foreach style loop.
Attributes	
select	(<i>tt</i>) The database query to execute. The attribute is required. The query can be either a standard SQL query or stored procedure call.
var	(<i>tt</i>) The name of the variable instance for the row object. Each column in the resulting recordset has a corresponding property in this object. For instance, if a returned recordset has columns: Id, Name, and LastRun, and the 'var' attribute is 'row', then you can access the column data using \$row.Name, \$row.Id, and \$row.LastRun. Since these fields are coming from a data reader, they are read only. You would need to create a dbexec element to update the database.
connection	(<i>tt</i>) The optional connection name. This attribute provides a name for the connection object on which this query should execute. This attribute is required if the connection element has a name, or if multiple connections are being used.
timeout	(<i>tt</i>) The timeout in milliseconds for the statement to execute. If not specified, the system default is used.
Child Elements	
<param>	<p>If the query is a stored procedure that requires arguments, they are specified as a series of child <param> elements. Each parameter has 'name' and 'value' attributes.</p> <p>Attributes: name (<i>tt</i>), value (<i>ex</i>), type (<i>tt</i>).</p> <p>You must also specify the 'type' attribute with the parameter type. Valid types are listed below:</p> <ul style="list-style-type: none"> AnsiString Binary Byte Boolean Currency Date DateTime Decimal Guid

	<p>Int16 Int32 Int64 Object SByte Single String Time UInt16 UInt32 UInt64 VarNumeric AnsiStringFixedLength Xml</p>
Example	<pre><!-- This example demonstrates calling a stored procedure that requires a parameter and returns a recordset --> <dbquery select="SelectScripts" var="matchingScript"> <param name="@scriptName" value="Interactive"/> <expr> \$output.WriteLine(format("Matching: {0}", \$matchingScript.Name)) </expr> </dbquery></pre>

DbExec (<dbexec>)

Description	This element executes a sql command. The command can be any style of SQL. Specify the 'var' attribute to save off the return value. If the query is not specified as an attribute, it can be the inner text of the element.
Attributes	
query	(<i>tt</i>) The database query to execute. The attribute is optional. The query can be either a standard SQL query or stored procedure call. If the attribute is not specified, then the element inner text supplies the query.
var	(<i>tt</i>) The optional name of the variable instance for the result object. When specified, the dbexec performs an ExecuteScalar(), otherwise, it performs an ExecuteNonQuery.
connection	(<i>tt</i>) The optional connection name. This attribute provides a name for the connection object on which this query should execute. This attribute is required if the connection element has a name, or if multiple connections are being used.
timeout	(<i>tt</i>) The timeout in milliseconds for the statement to execute. If not specified, the system default is used.
Child Elements	
<param>	<p>If the query is a stored procedure that requires arguments, they are specified as a series of child <param> elements. Each parameter has 'name' and 'value' attributes.</p> <p>Attributes: name (<i>tt</i>), value (<i>ex</i>), type (<i>tt</i>).</p> <p>You must also specify the 'type' attribute with the parameter type. Valid types are listed below:</p>

	<p>AnsiString Binary Byte Boolean Currency Date DateTime Decimal Guid Int16 Int32 Int64 Object SByte Single String Time UInt16 UInt32 UInt64 VarNumeric AnsiStringFixedLength Xml</p>
<p>Example</p>	<pre><!-- This example demonstrates executing a scalar query --> <dbexec query="SELECT COUNT(*) FROM Scripts" var="scriptCount"/> <expr>\$output.WriteLine(\$scriptCount)</expr> <dbexec> INSERT INTO [MyTable] VALUES ('First', {{\$secondValue}}, 'Third') </dbexec></pre>

Using the Script Engine

The CUEScript engine can be embedded directly within .NET applications.

- 1) Make a reference to the CUESoft.CUEScript.dll
- 2) Include a reference the CUESoft.CUEScript namespace.
- 3) Declare a ScriptEngine instance.
- 4) Declare a ScriptDocument instance. Load the XML document using one of the methods provided.
- 5) Pass the document into the engine and invoke the Execute() method.

```
// Start the script engine
ScriptDocument scriptDoc = new ScriptDocument();
scriptDoc.LoadXml( data.ToString() );
engine = new ScriptEngine( scriptDoc );

if ( connection != null )
    engine.Context.SetVariable( "connection", connection,
                               ContextType.Global );

engine.Context.SetVariable("command", sendCommand, ContextType.Global);
engine.Context.SetVariable( "session",
                             new SessionInstance( this ), ContextType.Global );

engine.Context.SetVariable( "fileTransfer", fileTransferProcessor,
                             ContextType.Global );

ExitState scriptExit = engine.Execute();
```

The code sample above demonstrates the steps necessary to instantiate and executes a script. Note, the programmer can inject objects into the script context as object variables that can then be accessed within the script.

Extending the Script Engine

Programmers can add new functionality to the CUEScript language programmatically. CUEScript supports two types of programmatic extension: objects and tags.

IMPORTANT: CUEScript will reflect against all assemblies in the execution domain with the name "CUEScript" in the assembly name. Therefore, any CUEScript extension DLLs must have "CUEScript" in the name.

To inject an object into CUEScript:

- 1) Create a public class.
- 2) Mark the class with the [ScriptObject] attribute. ScriptObjectAttribute requires you to pass a name for this object. This is the name that the script will use to access it. Names must be globally unique, so choose names carefully. There is an optional Boolean parameter in the attribute: UsesContext. When true, the script engine will construct this object passing the ScriptContext instance to the constructor. If UsesContext=false, the script engine will call the default (no arguments) constructor.
- 3) Create public members that the script can call.
- 4) Build the assembly. Place it where it can be loaded by the script.
- 5) Make an <assembly> reference within the script where you want the object to be inserted.
- 6) Use the named variable to access the object.

```
// Show a script extension
Using CUESoft.CUEScript;

[ScriptObject( "greeting", UsesContext=true )]
public class MyGreeting
{
    public MyGreeting( ScriptContext context )
    {
        // I can create multiple objects and store them in the context
    }

    public string Hello()
    {
        return "Hello World!";
    }
}
```

In my script, I need to make an assembly reference.

```
<script name='main'>
  <assembly file='Greetings.dll' />
  <expr>
    $output( $greeting.Hello() );
  </expr>
</script>
```

Programmers can also extend the script language by creating new XML tags.

- 1) Create a public class for your element derived from CUESoft.CUEScript.ScriptElement
- 2) Mark the class with a [ScriptCommand(tagName)] attribute. Pass the XML tag name you are providing.

3) Create the required constructor:

```
public MyScriptElement( string prefix, string localname,
                      string nsURI, XmlDocument doc )
    : base( prefix, localname, nsURI, doc )
{ }
```

- 4) Override the Execute() member to perform the actions of the element. The ScriptElement class and ScriptContext class have a number of useful helper functions to enable you to implement the element's functionality.
- 5) Build the assembly and make it available to the script.
- 6) Make an assembly reference in the script.
- 7) Use the new tag in the script anywhere after the assembly tag.

Below is the Execute() member for the <expr> CUEScript element:

```
public override bool Execute( ScriptContext context )
{
    using ( new ScriptStackManager( context, this ) )
    {
        if ( context.State != ExitState.Normal )
            return false;

        string expression;
        if ( HasAttribute( "value" ) )
            expression = this.GetAttribute( "value" );
        else
            expression = this.InnerText; //expression of inner text

        object result = context.EvaluateExpression( expression.Trim(), this );
        if ( result is Exception )
            return RaiseException( context, (Exception)result );

        return true;
    }
}
```