Proposed JavaScript Extensions
Borland International

Preliminary draft

22 Nov 1996
Abstract:
This document describes the major enhancements to the JavaScript language as currently implemented
in the “IntraBuilder” product. A more detailed Specification Extension proposal will be forthcoming
shortly.

CORE LANGUAGE EXTENSIONS..............................................................................................................
CLASS DEFINITION (CLASS, EXTENDS):.................................................................................................
EXCEPTION HANDLING (TRY, CATCH, THROW, FINALLY):........................................................................
SWITCH STATEMENT:.................................................................................................................................
OTHER..........................................................................................................................................................

extern..........................................................................................................................................................
CodeBlocks..................................................................................................................................................
Literal array syntax..................................................................................................................................

PRE-PROCESSOR...........................................................................................................................................
#DEFINE..........................................................................................................................................................
#if.................................................................................................................................................................
#ifndef...........................................................................................................................................................
#else..............................................................................................................................................................
#else ...............................................................................................................................................................

OBJECT EXTENSIONS..................................................................................................................................
CLASS EXCEPTION........................................................................................................................................
CLASS OBJECT...........................................................................................................................................
CLASS ASSOCARRAY...................................................................................................................................
CLASS FILE...................................................................................................................................................
CLASS NETINFO...........................................................................................................................................
CLASS ACTIVEX...........................................................................................................................................
CLASS JAVAAPPLET....................................................................................................................................
CLASS OLEAUTOCLIENT.................................................................................................................................
CLASS TIMER.............................................................................................................................................

IMPLEMENTATION OF OBJECTS IN NS SPEC...........................................................................................
CLASS STRING.............................................................................................................................................
CLASS ARRAY..............................................................................................................................................
Core Language Extensions

The proposed keyword extensions that are described in this section are: **class, extends, try, catch, throw, finally, switch, case, default**

The extensions are based on the Java language.

**Class Definition (class, extends):**

**Syntax**

```
class <class name> [extends <superclass name> [custom]]{
    [constructor code]
    [<methods>]
}
```

*<class name>*
The name of the class.

*extends <superclass name>*
Indicates that the class is a derived class which inherits the methods defined in the superclass. The superclass constructor is called before the *<constructor code>* in the class is called, which means that any properties created in the superclass are inherited by the class.

*custom*
Identifies the class as a custom component class, so that its predefined properties are not streamed out by the Form or Report Designer.

*<constructor code>*
The code that is called when a new instance of the class is created with the new operator. The constructor consists of all the code at the top of the class declaration up to the first method.

*<methods>*
Any number of functions designed for the class.

**Description**

A class declaration formalizes the creation of an object and its methods. Although you can always add properties to an object and assign methods dynamically, a class simplifies the task and allows you to build a clear class hierarchy.

Another benefit is polymorphism. Every function defined in the class becomes a method of the class. An object of that class automatically has a property with the same name as each function and which contains a reference to that function. Because a method is part of the class, different functions may use the same name as long as they are methods of different classes. For example, you can have multiple `copy()` functions in different classes, with each one applying to objects of that class. Without
classes, you would have to name the functions differently even if they performed the same task conceptually.

Before the first statement in the constructor is executed, if the class extends another class, the constructor for that superclass has already been executed, so the object contains all the superclass properties. Any properties that refer to methods, as described in the previous paragraph, are assigned. This means that if the class contains a method with the same name as a method in a superclass, the method in the class overrides the method in the superclass. The class constructor, if any, then executes.

In the constructor, the reserved word this refers to the object being created. Typically, the constructor creates properties by assigning them to this with dot notation. However, the constructor may contain any code at all, except another class—you can’t nest classes—or a function, since that function would become a method of the class and indicate the end of the constructor.

**Exception Handling (try, catch, throw, finally):**

Control statements used to handle exceptions and other deviations of program flow, generally in response to error conditions.

**Syntax**

```
try
  { <statement block 1> }
[ catch( <exception type1> <exception oRef1> )
    { <statement block 2> } ]
[ catch( <exception type2> <exception oRef2> )
    { <statement block 3> } ]
[ catch ... ]
[ finally
    { <statement block 4> } ]
```

A statement block inside curly braces for which the following catch or finally block—or both—will be used if an exception occurs during execution. A try block must be followed by either a catch block, a finally block, or both.

```
catch { <statement block 2> }
```

A statement block inside curly braces that is executed when an exception occurs.

```
<exception type>
```

The class name of the exception to look for—usually, Exception.

```
<exception oRef>
```

A formal parameter to receive the Exception object passed to the catch block.

```
catch...
```

Catch blocks for other types of exceptions.

```
finally { <statement block 2> }
```
A statement block inside curly braces which is always executed after the try block, even if an exception or other deviation of program flow occurs. If there is both a catch and a finally, the finally block executes after the catch block.

Note: There is no semicolon between the closing brace (}) and the keyword catch or finally.

**Description**

An exception is a condition that is either generated by the scripting environment, usually in response to an error, or by the programmer. By default, the scripting environment handles an exception by displaying an error dialog and terminating the currently executing script. You can use finally to make sure some code gets executed even if there is an exception, and catch to handle the exception yourself, in the following combinations:

For a block of code that may generate an exception, place the code inside a try block. To prevent the exception from generating a standard error dialog and terminating execution, place exception handling code in a catch block after the try. If an exception occurs, execution immediately jumps to the catch block; no more statements in the try block are executed. If no exception occurs, the catch block is not executed.

If there’s some code that should always be executed at the end of a process, whether or not the process completes successfully, place that code in a finally block. With try and finally but no catch, if an exception occurs during the try block, execution immediately jumps to the finally block; no more statements in the try block are executed. Since there was no catch, you would still have an exception, which if not handled by a higher-level catch as described later, the scripting environment would handle as usual, after executing the finally block. If no exception occurs, the finally block is executed after the try.

If you have all three—try, catch, and finally—if an exception occurs, execution immediately jumps to the catch block; after the catch block executes, the finally block is executed. If there is no exception during the try, then the catch block is skipped, and the finally block is executed.

The code that is covered by try doesn’t have to be inside the statement block physically; the coverage exists until that entire block of code is executed. For example, you may have a function call inside a try block, and if an exception occurs while that function is executing—even if that function is defined in another script file—execution jumps back to the corresponding catch or finally.

A try block may be followed by multiple catch blocks, each with its own <exception type>. When an exception occurs, the scripting environment compares the <exception type> with the className property of the Exception object. If they match, that catch block is executed and all others are skipped. If the className does not match, the script environment searches the class hierarchy of that object to find a match. If no match is found, the next catch block is tested. For example, the DbException class is a subclass of the Exception class. If the blocks are arranged like this:

```java
try {
    // Statements
}
catch ( DbException e ) {
    // Block 1
}
catch ( Exception e ) {
    // Block 2
```
then if a DbException occurs, execution goes to Block 1, because that’s a match. If an Exception occurs, execution goes to Block 2, because Block 1 doesn’t match, but Block 2 does. If the blocks are arranged the other way around, like this:

```java
try {
    // Statements
} catch (Exception e) {
    // Block 1
} catch (DbException e) {
    // Block 2
}
```

then all exceptions always go to Block 1, because all Exceptions are derived from the Exception class. Therefore, when using multiple catch blocks, list the most specific exception classes first.

You can generate exceptions on purpose with the throw statement to control program flow. For example, if you enter deeply nested control structures or subroutines from a try block, you can throw an exception from anywhere in the nested code. This would cause execution to jump back to the corresponding catch or finally, instead of having to exit each control structure or subroutine one-by-one.

You may also nest try structures. An exception inside the try block causes execution to jump to the corresponding catch or finally, but an exception in a catch or finally is simply treated as an exception. Also, if you have a try and finally but no catch, that leaves you with an unhandled exception. If the try/catch/finally is itself inside a try block, then that exception would be handled at that next higher level, as illustrated in the following code skeleton:

```java
try {
    // exception level 1
    try {
        // exception level 2
    } catch (Exception e) {
        // handler for level 2
        // but exception level 1
    } finally {
        // level 2
    }
} catch (Exception e) {
    // handler for level 1
}
```

Note that if an exception occurs in the level 2 catch, the level 2 finally is still executed before going to the level 1 catch, because a finally block is always executed after a try block.

In addition to exceptions, other program flow deviations—specifically break, continue, and return—are also caught by try. If there is a corresponding finally block, it’s executed before control is transferred to the expected destination. (catch catches only exceptions.)
Switch Statement:

Conditionally executes statements based on the value of a numeric expression.

Syntax

switch ( <expN> ) {
    [case <constant expN1>:
        [<statement block>]
        [break;]
    [case <constant expN2>:
        [<statement block>]
        [break;]
    [case ...]]]
    [default:
        [<statement block>]]
}

<expN>

A numeric expression to evaluate.

case <constant expN>

Where execution goes if <expN> equals <constant expN>. <constant expN> should be a non-negative integer constant. Each integer may be used in only one case statement in a switch block.

<statement block>

Any number of statements that are executed when <expN> equals <constant expN> in the preceding case statement.

break

Exits switch block and continues execution with the next statement after the switch block.

default

If present, executes the <statement block> that follows if the <expN> does not match any <constant expN>.

Description

Use switch when testing a numeric expression that might evaluate to a number of known values. Using switch is more efficient than using if with a compound logical expression to test the same expression, because the expression is evaluated only once. switch works by diverting execution to the case that contains the matching numeric value. Once the jump has been made into a case, the remaining case and default statements are ignored, and the rest of the statements in the switch block are executed from the top down. There is usually a break at the end of each <statement block> so that only the statements for that case are executed, but you can construct a switch block to take advantage of the top-down execution.

A switch block must contain at least one case or the default.
**Other:**

These items are not based on the Java language….

**extern**

Allow use of existing applications from the scripting environment through importing methods.

Example:

```plaintext```
extern int myFunc(int) MYAPP;
```plaintext```

**CodeBlocks**

A codeblock is code that can be stored in a variable or property, i.e.:

**Syntax**

```plaintext```
identifier = {; expression list }
```plaintext```

or

```plaintext```
identifier = {|| expression list }
```plaintext```

**Literal array syntax**

Example:

```plaintext```
a = {"A", "B", "C"};
```plaintext```
Pre-Processor

The usage of a preprocessor allows you to:

- Replace constants and “magic numbers” in your code with easy-to-read and easy-to-change identifiers
- Create macro-functions to replace complex expressions with parameters
- Use collections of constant identifiers and macro-functions in multiple script files
- Maintain separate versions of your scripts, for example debug and production versions, in the same script files through conditional compilation

The script environments preprocessor is similar to the preprocessor used in the C language. It uses a handful of preprocessor directives to control its activities. All preprocessor directives start with the # character and each one must be on its own, single line in the script file. They do not use a line termination character.

The macros added are: `#define`, `#else`, `#endif`, `#if`, `#ifdef`, `#ifndef`, `#include`, `#undef`.

### `#define`

**Syntax**

```
#define <identifier> [<replacement text>]
#define <identifier>(<parameter list>) <replacement text with parameters>
```

**<identifier>**

A name. It identifies the text to replace if `<replacement text>` is supplied. The name must start with an alphabetic character and can contain any combination of alphabetic or numeric characters, uppercase or lowercase letters. The identifier is case-sensitive.

**(<parameter list>)**

Parameter names that correspond to arguments passed to a macro-function that you create with `#define <identifier>(<parameter list>) <replacement text>`. If you specify multiple parameters, separate each with a comma. There cannot be any spaces between the `<identifier>` and the opening parenthesis of `<parameter list>`, or after any of the parameter names in the `<parameter list>`.

**<replacement text>**

The text you want to use to replace all occurrences of `<identifier>`. If you specify `<replacement text>`, the preprocessor scans each source code line for identifiers and replaces each one it encounters with the specified replacement text. `<replacement text>` can be any text that is part of a JavaScript script, such as a string, numeric expression, or series of statements.

### `#if`

Controls conditional compilation of code based on the value of an identifier assigned with `#define`.

**Syntax**

```
#if <expression>
```

The value of `<expression>` is evaluated, and if the result is true, the code enclosed in the #if block is compiled. If the result is false, the code is not compiled. The `<expression>` can be any boolean expression that evaluates to true or false. For example,

```
#if DEBUG
  // Code only for debug version
#endif
```
#if <condition>
<statements 1>
#else
<statements 2>
#endif

A logical expression, using an identifier you’ve defined, that evaluates to true or false.

<statements 1>

Any number of statements and preprocessor directives. These lines are compiled if <condition> evaluates to true.

#else <statements 2>

Specifies the lines you want to compile if <condition> evaluates to false.

Description

Use the #if directive to conditionally compile sections of source code based on the value of <identifier>. Two other directives, #ifdef and #ifndef, are also used to conditionally include or exclude code for compilation. Unlike the #if directive, however, they test only for the existence of an identifier, not for its value.

The <condition> must be a simple logical condition; that is, a single test using one basic comparison operator (==, <, >, <=, >=, !=). You may nest conditional compilation directives.

Conditional compilation is useful when maintaining different versions of the same script, for debugging, and for managing the use of #include files. Using #if for conditional compilation is different than conditionally executing code with an if statement. With if, the code still gets compiled into the resulting byte code file, even if it is never executed. By using #if to exclude code you don’t want for a particular version of your script, the code is never compiled into byte code.

#ifdef

Controls conditional compilation of code based on the existence of an identifier created with #define.

Syntax

#ifdef <identifier>
<statements 1>
#else
<statements 2>
#endif

<identifier>

The identifier you want to test for. <identifier> is defined with the #define directive.

<statements 1>

Any number of statements and preprocessor directives. These lines are compiled if <identifier> has been defined.

#else <statements 2>
Specifies the lines to compile if <identifier> has not been defined.

Description

Use the #ifdef directive to conditionally compile sections of source code. If you’ve defined <identifier> with #define, the code you specify with <statements 1> is compiled; otherwise, the code following #else, if any, is compiled.

You may nest conditional compilation directives.

Conditional compilation is useful when maintaining different versions of the same script, for debugging purposes, and for managing the use of #include files. Using #ifdef for conditional compilation is different than not executing code with an if statement. With if, the code still gets compiled into the resulting byte code file, even if it is never executed. By using #ifdef to exclude code you don’t want for a particular version of your script, the code is never compiled into byte code.

 ifndef

Controls conditional compilation of code based on the existence of an identifier assigned with #define.

Syntax

 ifndef <identifier>
 <statements 1>
 [#else
 <statements 2>]
 endif

<identifier>

The identifier you want to test for. <identifier> is defined with the #define directive.

<statements 1>

Any number of statements and preprocessor directives. These lines are compiled if <identifier> has not been defined.

#else <statements 2>

Specifies the lines to compile if <identifier> has been defined.

Description

Use the ifndef directive to conditionally compile sections of source code. If you haven’t defined <identifier> with #define, the code you specify with <statements 1> is compiled; otherwise, the code following #else, if any, is compiled.

Use ifndef if you want to include code only if the identifier is not defined. Otherwise, you can use ifdef to include code only if the identifier is defined, and ifndef with its #else option to include different sets of code depending on the existence of the identifier.

You may nest conditional compilation directives.

Conditional compilation is useful when maintaining different versions of the same script, for debugging purposes, and for managing the use of #include files. Using ifndef for conditional compilation is different than not executing code with an if statement. With if, the code still gets compiled into the resulting byte code file, even if it is never executed. By using ifndef to exclude code you don’t want for a particular version of your script, the code is never compiled into byte code.
#include

Inserts the contents of the specified source file (known as an include or header file) into the current script file at the location of the #include statement.

Syntax

```c
#include <filename> | "<filename>"
```

`<filename> | "<filename>"`

The name of the file, optionally including a full or partial path, whose contents are to be inserted into the current script file. You can specify the file name within or without quotes. An include file typically has an .h file-name extension.

If you specify `<filename>` without a path, the preprocessor uses the following search order:

1. It searches the current directory for the file exactly as you’ve specified it.
2. If you omitted the .h file-name extension, it adds the extension and searches the current directory.
3. If it can’t find the file in the current directory, it looks in `<home directory>\INCLUDE`. (The home directory is the one returned by `_sys.env.home()`.)
4. If it can’t find the file in the current directory or `<home directory>\INCLUDE`, it looks in the directory you specify with the DOS environment variable INCLUDE.

Description

The effect of #include is as if the contents of the specified file were typed into the current script file at the location of the #include statement. The specified file is called an include file. #include is used primarily for files which have #define directives.

Identifiers are available only to the script in which they are defined. To use a single set of identifiers in multiple scripts, save the #define statements in a file, then use the #include directive to define the identifiers in additional scripts.

An advantage of having all the #define statements in one file is the ease of maintenance. If you need to modify any of the #define statements, you need only change the include file; the script files that use the #define statements remain unchanged. After you modify the include file, recompile your script file for the changes to take effect.

#undef

Removes the current definition of the specified identifier previously defined with #define.

Syntax

```c
#undef <identifier>
```

`<identifier>`

The identifier whose definition you want to remove.

Description

The #undef directive removes the definition of an identifier previously defined with the #define directive. If you use #define with `<replacement text>`, the preprocessor replaces all instances of the identifier with the replacement text from the point it encounters that #define
until it encounters an #undef specifying the same identifier. Therefore, to replace an identifier only in parts of a script, insert #undef <identifier> into your script where you want the search-and-replace process to stop.

#undef is also required if you want to change the <replacement text> for an identifier. You cannot use #define for an identifier that is already defined. You must #undef the identifier first, then specify a new #define directive. Attempting to #undef an identifier that is not defined has no effect; no error is generated.
class Exception

Properties

The following table lists the properties of the Exception class. (No events or methods are associated with this class.)

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>className</td>
<td>Exception</td>
<td>Identifies the object as an instance of the Exception class</td>
</tr>
<tr>
<td>code</td>
<td></td>
<td>A numeric code to identify the type of exception</td>
</tr>
<tr>
<td>message</td>
<td>Empty string</td>
<td>Text to describe the exception</td>
</tr>
</tbody>
</table>

Description

An Exception object is automatically generated by the script environment whenever an error occurs. The object’s properties contain information about the error.

You can also create an Exception object manually, which you can fill with information and throw to manage execution or to jump out of deeply nested statements.

You may subclass the Exception class to create your own custom exception objects. A try block may be followed by multiple catch blocks, each one looking for a different exception class.

class Object

Properties

An object of the Object class has no initial properties, events, or methods.

Description

Use the Object class to create your own simple objects. Once the new object is created, you may add properties and methods through assignment. You cannot add events.

class AssocArray

Properties

The following tables list the properties and methods of the AssocArray class. (No events are associated with this class.)

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>className</td>
<td>AssocArray</td>
<td>Identifies the object as an instance of the AssocArray class</td>
</tr>
<tr>
<td>firstKey</td>
<td></td>
<td>Character string assigned as the subscript of the first element of an associative array</td>
</tr>
</tbody>
</table>
Description

In an associative array, elements are associated with arbitrary character strings, which act as key values. The keys may be of any length, and are case-sensitive. An AssocArray is a one-dimensional array.

New elements are created simply by assigning a value to a key. If the key does not exist, a new element is created. If the key already exists, then the old value is replaced. For example,

```java
aTest = new AssocArray();

aTest[ "alpha" ] = 1  // Create element with key "alpha" value 1
```

class File

Properties

The following tables list the properties and methods of the File class. (No events are associated with this class.)

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>className</td>
<td>File</td>
<td>Identifies the object as an instance of the File class</td>
</tr>
<tr>
<td>handle</td>
<td>–1</td>
<td>Operating system file handle</td>
</tr>
<tr>
<td>path</td>
<td></td>
<td>Full path and file name for open file</td>
</tr>
<tr>
<td>position</td>
<td>0</td>
<td>Current position of file pointer, relative to the start of the file</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessDate()</td>
<td></td>
<td>Returns the last date a file was opened</td>
</tr>
<tr>
<td>close()</td>
<td></td>
<td>Closes the currently open file</td>
</tr>
<tr>
<td>copy()</td>
<td>&lt;filename expC&gt;, &lt;new name expC&gt;</td>
<td>Makes a copy of the specified file</td>
</tr>
<tr>
<td>create()</td>
<td>&lt;filename expC&gt;</td>
<td>Creates a new file with optional access attributes [&lt;access rights&gt;]</td>
</tr>
<tr>
<td>createDate()</td>
<td>&lt;filename expC&gt;</td>
<td>Returns the date when the file was created</td>
</tr>
<tr>
<td>createTime()</td>
<td>&lt;filename expC&gt;</td>
<td>Returns the time a file was created as a string</td>
</tr>
<tr>
<td>date()</td>
<td>&lt;filename expC&gt;</td>
<td>Returns the date the file was last modified</td>
</tr>
<tr>
<td>delete()</td>
<td>&lt;filename expC&gt;</td>
<td>Deletes the specified file</td>
</tr>
<tr>
<td>eof()</td>
<td></td>
<td>Returns true or false indicating if the file pointer is positioned past the end of the currently open file</td>
</tr>
<tr>
<td>error()</td>
<td></td>
<td>Returns a number indicating the last error encountered</td>
</tr>
<tr>
<td>exists()</td>
<td>&lt;filename expC&gt;</td>
<td>Returns true or false to indicate whether the specified disk file exists</td>
</tr>
<tr>
<td>flush()</td>
<td></td>
<td>Writes current data in the file buffer to disk and keeps file open</td>
</tr>
</tbody>
</table>
gets()       [<chars read expN>]
             [, <eol expC>]  Reads and returns a line from a file, leaving the file
pointer at
             the beginning of the next line. Same as readln()
open()      <filename expC> Opens an existing file with optional access attributes
             [,<access rights>]  
puts()      <input string expC>
             [, <max chars expN>
             [, <eol expC>]  Writes a character string and end-of-line character(s) to a
file.
             Same as writeln()
read()      <characters expN>
             Reads and returns the specified number of characters from
             the
             file starting from the current file pointer position; leaving
             the
             file pointer at the character after the last one read
readln()    [<chars read expN>]
             [, <eol expC>]  Reads and returns a line from a file, leaving the file
pointer at
             the beginning of the next line. Same as gets().
rename()    <filename expC> Changes the name of the specified file to a new name
             [, <new name expC>
seek()       <offset expN>
             [, 0 | 1 | 2 ]  Moves the file pointer the specified number of bytes
             within a
             file, optionally allowing the movement to be
             from the
             beginning (0), end(2), or current file position (1)
shortName() <filename expC> Returns the short (8.3) name for a file
size()      <filename expC> Returns the number of bytes in the specified file
time()      <filename expC> Returns the time the file was last modified as a string
write()     <expC>
             [, <max chars expN>]
             [, <eol expC>]  Writes the specified string into the file at the current file
position, overwriting any existing data and leaving the file
pointer at the character after the last character written
writeln()   <input string expC>
             [, <max chars expN>
             [, <eol expC>]  Writes a character string and end-of-line character(s) to a
file.
             Same as puts().

Description

Use a File object for direct byte-level access to files. A File object may access only one file at a time.

class NetInfo

Properties

The following tables list the properties of the NetInfo class. (No events or methods are associated with this class.)

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>className</td>
<td>NetInfo</td>
<td>Identifies the object as an instance of the NetInfo class</td>
</tr>
<tr>
<td>IPAddress</td>
<td>-1</td>
<td>IP address of client browser</td>
</tr>
<tr>
<td>referrer</td>
<td></td>
<td>URL of document that contained link to current location</td>
</tr>
<tr>
<td>remoteHost</td>
<td></td>
<td>Host name of client browser</td>
</tr>
</tbody>
</table>
serverName 
sessionID: -1 
userAgent: 

**class ActiveX**

Properties

The following tables list the properties, events, and methods of the ActiveX class.

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt</td>
<td></td>
<td>An alternate string that is displayed if the client browser does not support ActiveX controls</td>
</tr>
<tr>
<td>classId</td>
<td></td>
<td>The ID string that identifies the ActiveX control</td>
</tr>
<tr>
<td>className</td>
<td>ActiveX</td>
<td>Identifies the object as an instance of the ActiveX class</td>
</tr>
<tr>
<td>codeBase</td>
<td></td>
<td>The URL for the ActiveX control</td>
</tr>
<tr>
<td>form</td>
<td></td>
<td>The form that contains the ActiveX object</td>
</tr>
<tr>
<td>height</td>
<td></td>
<td>Height in characters</td>
</tr>
<tr>
<td>left</td>
<td></td>
<td>The location of the left edge of the ActiveX object</td>
</tr>
<tr>
<td>object in</td>
<td></td>
<td>characters, relative to the left edge of the form</td>
</tr>
<tr>
<td>name</td>
<td></td>
<td>The name of the ActiveX object</td>
</tr>
<tr>
<td>pageno</td>
<td>1</td>
<td>The page of the form on which the ActiveX object appears</td>
</tr>
<tr>
<td>params</td>
<td></td>
<td>Parameters passed to the ActiveX control</td>
</tr>
<tr>
<td>parent</td>
<td></td>
<td>The form that contains the ActiveX object</td>
</tr>
<tr>
<td>top</td>
<td></td>
<td>The location of the top edge of the ActiveX object</td>
</tr>
<tr>
<td>object in</td>
<td></td>
<td>characters, relative to the top edge of the form</td>
</tr>
<tr>
<td>width</td>
<td></td>
<td>Width in characters</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>onDesignLoad</td>
<td>&lt;from palette expL&gt;</td>
<td>After the ActiveX object is first added from the palette and then every time the form is opened in the Form Designer</td>
</tr>
<tr>
<td>onServerLoad</td>
<td></td>
<td>After the form containing the ActiveX object is loaded, but before it is rendered into HTML</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>release()</td>
<td>memory</td>
<td>Explicitly releases the ActiveX object from memory</td>
</tr>
</tbody>
</table>

**class JavaApplet**

Properties

The following tables list the properties, events, and methods of the JavaApplet class.

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt</td>
<td></td>
<td>An alternate string that is displayed if the client browser does not support Java applets</td>
</tr>
<tr>
<td>className</td>
<td>JavaApplet</td>
<td>Identifies the object as an instance of the JavaApplet</td>
</tr>
<tr>
<td>JavaApplet</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
class OleAutoClient

Properties

The properties, events, and methods of each instance of the OleAutoClient class depend on the attached OLE automation server.

Description

OLE automation allows you to control another application, an OLE automation server, through an OLE automation client.

class Timer

Properties

The following tables list the properties and events of the Timer class. (No methods are associated with this class.) For details on each property, click on the property below.

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>className</td>
<td>Timer</td>
<td>Identifies the object as an instance of the Timer class</td>
</tr>
<tr>
<td>enabled</td>
<td>false</td>
<td>Whether the Timer is active</td>
</tr>
<tr>
<td>interval</td>
<td>10</td>
<td>The interval between actions, in seconds</td>
</tr>
<tr>
<td>Event</td>
<td>Parameters</td>
<td>Description</td>
</tr>
</tbody>
</table>
onTimer
Action to take when interval expires

Description
An object that initiates a recurring action at preset intervals.

Implementation of Objects in NS Spec

**class String**

Current Implementation of the String

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>className</td>
<td>String</td>
<td>Identifies the object as an instance of the String class</td>
</tr>
<tr>
<td>length</td>
<td></td>
<td>The number of characters in the string</td>
</tr>
<tr>
<td>string</td>
<td></td>
<td>The value of the String object</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anchor()</td>
<td>&lt;expC&gt;</td>
<td>Tags the string as an anchor &lt;A NAME&gt;</td>
</tr>
<tr>
<td>big()</td>
<td></td>
<td>Tags the string as big &lt;BIG&gt;</td>
</tr>
<tr>
<td>blink()</td>
<td></td>
<td>Tags the string as blinking &lt;BLINK&gt;</td>
</tr>
<tr>
<td>bold()</td>
<td></td>
<td>Tags the string as bold &lt;BOLD&gt;</td>
</tr>
<tr>
<td>charAt()</td>
<td>&lt;index expN&gt;</td>
<td>Returns the character in the string at the designated position</td>
</tr>
<tr>
<td>fixed()</td>
<td></td>
<td>Tags the string as fixed font &lt;TT&gt;</td>
</tr>
<tr>
<td>fontcolor()</td>
<td>&lt;expC&gt;</td>
<td>Tags the string as the designated color</td>
</tr>
<tr>
<td>fontsize()</td>
<td>&lt;expN&gt;</td>
<td>Tags the string as the designated font size</td>
</tr>
<tr>
<td>indexOf()</td>
<td>&lt;expC&gt;</td>
<td>Returns the position of the search string inside the string</td>
</tr>
<tr>
<td>link()</td>
<td>&lt;expC&gt;</td>
<td>Tags the string as a link &lt;A HREF&gt;</td>
</tr>
<tr>
<td>small()</td>
<td></td>
<td>Tags the string as small &lt;SMALL&gt;</td>
</tr>
<tr>
<td>strike()</td>
<td></td>
<td>Tags the string as strikethrough &lt;STRIKE&gt;</td>
</tr>
<tr>
<td>sub()</td>
<td></td>
<td>Tags the string as subscript &lt;SUB&gt;</td>
</tr>
<tr>
<td>substring()</td>
<td>&lt;start index expN&gt;</td>
<td>Returns a substring derived from the string</td>
</tr>
<tr>
<td>sup()</td>
<td></td>
<td>Tags the string as superscript &lt;SUP&gt;</td>
</tr>
<tr>
<td>toLowerCase()</td>
<td></td>
<td>Returns the string in all lowercase</td>
</tr>
<tr>
<td>toUpperCase()</td>
<td></td>
<td>Returns the string in all uppercase</td>
</tr>
</tbody>
</table>

**class Array**

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>className</td>
<td>Array</td>
<td>Identifies the object as an instance of the Array class</td>
</tr>
<tr>
<td>dimensions</td>
<td></td>
<td>The number of dimensions in the array</td>
</tr>
<tr>
<td>length</td>
<td>0</td>
<td>The number of elements in the array</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
</table>
| add()       | <exp>      | Increases the size of a one-dimensional array by one and
assigns the passed value to the new element.
delete()  
<position expN> Deletes an element from a one-dimensional array, or deletes a  
[.1 | 2] row (1) or column (2) of elements from a two-dimensional array, without changing the size of the array.
dir()  
[<filespec expC>]Stores in the array five characteristics of specified files: name, size, modified date, modified time, and DOS attribute(s). Returns the number of files whose characteristics are stored.
dirExt()  
[<filespec expC>] Same as dir() method, but adds short (8.3) file name,  
create  
<filespec expC> date, create time, and access date.
element()  
<row expN> Returns the element number for the element at the specified  
[,<col expN>] row and column.
fill()  
<exp> Stores a specified value into one or more elements of the array.  
[, <start expN>  
[, <count expN>]]
grow()  
1 | 2 When passed 1, adds a single element to a one-dimensional array or a row to a two-dimensional array; when passed 2, adds a column to the array.
insert()  
<element expN> Inserts an element, row (1), or column (2) into an array  
[.1 | 2] without changing the size of the array (the last element, row, or column is lost).
resize()  
<rows expN> Increases or decreases the size of an array. First passed  
[,<cols expN>] parameter indicates the new number of rows,  
[, <retain values>]] the second parameter indicates the new number of columns.  
if  
nonzero, they are retained in their old positions.
scan()  
<exp> Searches an array for the specified expression; returns the  
[, <start expN>] element number of the first element that matches the expression, or –1 if the search is unsuccessful.
sort()  
<start expN> Sorts the elements in a one-dimensional array or the rows in a  
[,<count expN>] two-dimensional array in ascending (0) or descending (1) order.  
subscript()  
<element expN> Returns the row (1) or column (2) subscript for the specified  
1 | 2 element number;