[MS-ES3EX]:

Microsoft JScript Extensions to the ECMAScript Language Specification Third Edition

---

Intellectual Property Rights Notice for Open Specifications Documentation

- **Technical Documentation.** Microsoft publishes Open Specifications documentation ("this documentation") for protocols, file formats, data portability, computer languages, and standards support. Additionally, overview documents cover inter-protocol relationships and interactions.
- **Copyrights**. This documentation is covered by Microsoft copyrights. Regardless of any other terms that are contained in the terms of use for the Microsoft website that hosts this documentation, you can make copies of it in order to develop implementations of the technologies that are described in this documentation and can distribute portions of it in your implementations that use these technologies or in your documentation as necessary to properly document the implementation. You can also distribute in your implementation, with or without modification, any schemas, IDLs, or code samples that are included in the documentation. This permission also applies to any documents that are referenced in the Open Specifications documentation.
- **No Trade Secrets**. Microsoft does not claim any trade secret rights in this documentation.
- **Patents**. Microsoft has patents that might cover your implementations of the technologies described in the Open Specifications documentation. Neither this notice nor Microsoft's delivery of this documentation grants any licenses under those patents or any other Microsoft patents. However, a given Open Specifications document might be covered by the Microsoft Open Specifications Promise or the Microsoft Community Promise. If you would prefer a written license, or if the technologies described in this documentation are not covered by the Open Specifications Promise or Community Promise, as applicable, patent licenses are available by contacting iplg@microsoft.com.
- **License Programs**. To see all of the protocols in scope under a specific license program and the associated patents, visit the Patent Map.
- **Trademarks**. The names of companies and products contained in this documentation might be covered by trademarks or similar intellectual property rights. This notice does not grant any licenses under those rights. For a list of Microsoft trademarks, visit www.microsoft.com/trademarks.
- **Fictitious Names**. The example companies, organizations, products, domain names, email addresses, logos, people, places, and events that are depicted in this documentation are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

Reservation of Rights. All other rights are reserved, and this notice does not grant any rights other than as specifically described above, whether by implication, estoppel, or otherwise.

Tools. The Open Specifications documentation does not require the use of Microsoft programming tools or programming environments in order for you to develop an implementation. If you have access to Microsoft programming tools and environments, you are free to take advantage of them. Certain Open Specifications documents are intended for use in conjunction with publicly available standards specifications and network programming art and, as such, assume that the reader either is familiar with the aforementioned material or has immediate access to it.

Support. For questions and support, please contact dochelp@microsoft.com.

Revision Summary

Date	Revision History	Revision Class	Comments
3/26/2010	1.0	New	Released new document.
4/16/2010	1.1	Minor	Clarified the meaning of the technical content.
5/26/2010	1.2	None	Introduced no new technical or language changes.
9/8/2010	1.3	Major	Significantly changed the technical content.
10/13/2010	1.4	Minor	Clarified the meaning of the technical content.
2/10/2011	2.0	Minor	Clarified the meaning of the technical content.
2/22/2012	3.0	Major	Significantly changed the technical content.
7/25/2012	3.1	Minor	Clarified the meaning of the technical content.
3/31/2014	3.1	None	No changes to the meaning, language, or formatting of the technical content.
1/22/2015	4.0	Major	Updated for new product version.
7/7/2015	4.1	Minor	Clarified the meaning of the technical content.
11/2/2015	4.2	Minor	Clarified the meaning of the technical content.
1/20/2016	4.3	Minor	Clarified the meaning of the technical content.
3/22/2016	4.4	Minor	Clarified the meaning of the technical content.
11/2/2016	4.4	None	No changes to the meaning, language, or formatting of the technical content.
3/14/2017	4.4	None	No changes to the meaning, language, or formatting of the technical content.
10/3/2017	4.4	None	No changes to the meaning, language, or formatting of the technical content.

Table of Contents

1 Introduction 6
1.1 Glossary 6
1.2 References 6
1.2.1 Normative References 6
1.2.2 Informative References 6
1.3 Extension Overview (Synopsis) 7
1.3.1 Organization of This Documentation 8
1.4 Relationship to Standards and Other Extensions 8
1.5 Applicability Statement 8

2 Extensions 9
2.1 Conditional Source Text Processing 9
2.1.1 Global State 9
2.1.2 Conditional Processing Algorithm 10
2.2 Extensions to Types 19
2.2.1 SafeArray Type 19
2.2.2 VarDate Type 19
2.3 Extensions to Statements 19
2.3.1 debugger Statement 19
2.4 Extensions to Native ECMAScript Objects 20
2.4.1 Function Properties of the Global Object 20
2.4.1.1 ScriptEngine 20
2.4.1.2 ScriptEngineBuildVersion 20
2.4.1.3 ScriptEngineMajorVersion 20
2.4.1.4 ScriptEngineMinorVersion 20
2.4.1.5 CollectGarbage 20
2.4.1.6 RuntimeObject 21
2.4.1.7 GetObject 22
2.4.2 Constructor Properties of the Global Object 22
2.4.3 Object Functions in JScript 5.8 23
2.4.3.1 Object.getOwnPropertyDescriptor (O, P) 23
2.4.3.2 Object.defineProperty (O, P, Attributes) 24
2.4.4 Properties of Function Instances 27
2.4.4.1 The arguments Property 27
2.4.4.2 The caller Property 27
2.4.4.3 The [[Get]] (P) Method of a Function Object 27
2.4.5 String.prototype HTML Wrapper Properties 28
2.4.5.1 String.prototype.anchor(name) 28
2.4.5.2 String.prototype.big( ) 28
2.4.5.3 String.prototype.blink( ) 28
2.4.5.4 String.prototype.bold( ) 28
2.4.5.5 String.prototype.fixed( ) 28
2.4.5.6 String.prototype.fontcolor(color) 29
2.4.5.7 String.prototype.fontsize(size) 29
2.4.5.8 String.prototype.italics( ) 29
2.4.5.9 String.prototype.link(url) 29
2.4.5.10 String.prototype.small( ) 29
2.4.5.11 String.prototype.strike( ) 29
2.4.5.12 String.prototype.sub( ) 29
2.4.5.13 String.prototype.sup( ) 29
2.4.6 Date Time String Format for JSON 29
2.4.6.1 Extended Years 30
2.4.6.2 Date.prototype.getVarDate ( ) 31
2.4.6.3 Date.prototype.toJSON ( ) 31
2.4.7 Properties of the RegExp Constructor 31

2.4.7.1 RegExp.index 31
2.4.7.2 RegExp.input 31
2.4.7.3 RegExp.lastIndex 31
2.4.7.4 RegExp.lastMatch 31
2.4.7.5 RegExp.lastParen 32
2.4.7.6 RegExp.leftContext 32
2.4.7.7 RegExp.rightContext 32
2.4.7.8 RegExp.$1 - RegExp.$9 32
2.4.7.9 RegExp.$_ 32
2.4.7.10 RegExp['$&'] 32
2.4.7.11 RegExp['$+'] 32
2.4.7.12 RegExp["$`"] 32
2.4.7.13 RegExp["$'"] 33
2.4.8 Properties of the RegExp Prototype Object 33
2.4.8.1 RegExp.prototype.compile(pattern, flags) 33
2.4.9 Properties of the RegExp Instances 33
2.4.9.1 options 34
2.4.10 The Error Constructor 34
2.4.10.1 new Error () 34
2.4.10.2 new Error(number, message) 34
2.4.11 Properties of Error Instances 34
2.4.11.1 description 34
2.4.11.2 number 34
2.4.12 Native Error Types Used in This Standard 35
2.4.12.1 RegExpError 35
2.4.12.2 ConversionError 35
2.4.13 Properties of NativeError Instances 35
2.4.13.1 description 35
2.4.13.2 number 35
2.4.14 The JSON Object 35
2.4.14.1 The JSON Grammar 36
2.4.14.1.1 The JSON Lexical Grammar 36
2.4.14.1.2 The JSON Syntactic Grammar 37
2.4.14.2 parse (text [, reviver]) 37
2.4.14.3 stringify (value [, replacer [, space]]) 39
2.4.15 The Debug Object 45
2.4.15.1 Function Properties of the Debug Object 46
2.4.15.1.1 write ([item1 [, item2 [, …]]]) 46
2.4.15.1.2 writeln ([item1 [, item2 [, …]]]) 46
2.4.16 Enumerator Objects 46
2.4.16.1 The Enumerator Constructor Called as a Function 46
2.4.16.2 The Enumerator Constructor 46
2.4.16.2.1 new Enumerator ([collection]) 46
2.4.16.3 Properties of the Enumerator Constructor 47
2.4.16.3.1 Enumerator.prototype 47
2.4.16.4 Properties of the Enumerator Prototype Object 47
2.4.16.4.1 Enumerator.prototype.constructor 47
2.4.16.4.2 Enumerator.prototype.atEnd ( ) 47
2.4.16.4.3 Enumerator.prototype.item ( ) 47
2.4.16.4.4 Enumerator.prototype.moveFirst ( ) 48
2.4.16.4.5 Enumerator.prototype.moveNext ( ) 48
2.4.16.5 Properties of Enumerator Instances 48
2.4.17 VBArray Objects 48
2.4.17.1 The VBArray Constructor Called as a Function 48
2.4.17.1.1 VBArray (value) 48
2.4.17.2 The VBArray Constructor 49
2.4.17.2.1 new VBArray (value) 49
2.4.17.3 Properties of the VBArray Constructor 49

2.4.17.3.1 VBArray.prototype 49
2.4.17.4 Properties of the VBArray Prototype Object 49
2.4.17.4.1 VBArray.prototype.constructor 49
2.4.17.4.2 VBArray.prototype.dimensions ( ) 49
2.4.17.4.3 VBArray.prototype.getItem (dim1 [, dim2, [dim3, …]]) 50
2.4.17.4.4 VBArray.prototype.lbound ([dimension]) 50
2.4.17.4.5 VBArray.prototype.toArray ( ) 50
2.4.17.4.6 VBArray.prototype.ubound ([dimension]) 51
2.4.17.4.7 VBArray.prototype.valueOf ( ) 51
2.4.17.5 Properties of VBArray Instances 51
2.4.18 ActiveXObject Objects 51
2.4.18.1 The ActiveXObject Constructor Called as a Function 51
2.4.18.1.1 ActiveXObject (name [, location])) 52
2.4.18.2 The ActiveXObject Constructor 52
2.4.18.2.1 new ActiveXObject ((name [, location])) 52
2.4.18.3 Properties of the ActiveXObject Constructor 53
2.4.18.3.1 ActiveXObject.prototype 53
2.4.18.4 Properties of the ActiveXObject Prototype Object 53
2.4.18.4.1 ActiveXObject.prototype.constructor 53
2.4.18.5 Properties of ActiveXObject Instances 53
3 Security Considerations 54
4 Appendix A: Product Behavior 55
5 Change Tracking 56
6 Index 57

1 Introduction

This document describes extensions provided by JScript 5.x in these modes of Windows Internet Explorer: Quirks Mode, IE7 Mode, and IE8 Mode. The JScript 5.x dialects of ECMAScript are based on the *ECMAScript Language Specification 3rd Edition* [ECMA-262-1999], published in 1999.

Section 2 of this specification is normative. All other sections and examples in this specification are informative.

1.1 Glossary

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as defined in [RFC2119]. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

1.2 References

Links to a document in the Microsoft Open Specifications library point to the correct section in the most recently published version of the referenced document. However, because individual documents in the library are not updated at the same time, the section numbers in the documents may not match. You can confirm the correct section numbering by checking the Errata.

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information.

[ECMA-262-1999] Ecma International, "ECMAScript Language Specification", Standard ECMA-262 3rd Edition - December 1999, http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf

[ECMA-262/5] Ecma International, "ECMAScript Language Specification", Standard ECMA-262 5th Edition / December 2009, http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262%205th%20edition%20December%202009.pdf

[ISO-8601] International Organization for Standardization, "Data Elements and Interchange Formats - Information Interchange - Representation of Dates and Times", ISO/IEC 8601:2004, December 2004, http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=40874&ICS1=1&ICS2=140&ICS3=30

Note There is a charge to download the specification.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.rfc-editor.org/rfc/rfc2119.txt

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006, http://www.rfc-editor.org/rfc/rfc4627.txt

1.2.2 Informative References

[MS-ES3EX] Microsoft Corporation, "Microsoft JScript Extensions to the ECMAScript Language Specification Third Edition".

[MS-ES3] Microsoft Corporation, "Microsoft JScript ECMA-262-1999 ECMAScript Language Specification Standards Support Document".

[MS-ES5EX] Microsoft Corporation, "Internet Explorer Extensions to the ECMA-262 ECMAScript Language Specification (Fifth Edition)".

[MS-ES5] Microsoft Corporation, "Internet Explorer ECMA-262 ECMAScript Language Specification (Fifth Edition) Standards Support Document".

1.3 Extension Overview (Synopsis)

The extensions described in this document were selected for their applicability to [ECMA-262-1999]. Portions of this document also refer to [ECMA-262/5], the ECMAScript Language Specification 5th Edition, December 2009.

These extensions are organized based on sections of [ECMA-262-1999] as follows.

Section 2.1, Lexical Conventions

- Global State
- Conditional Processing Algorithm

Section 2.2, Types

Section 2.3, Statements

Section 2.4, Native ECMAScript Objects

- Function Properties of the Global Object
- Constructor Properties of the Global Object
- Object Functions in JScript 2.4.3
- Properties of Function Instances
- String.prototype HTML Wrapper Properties
- Date Time String Format for JSON
- Properties of the RegExp Constructor
- Properties of the RegExp Prototype Object
- Properties of the RegExp Instances
- The Error Constructor
- Properties of Error Instances
- Native Error Types Used in This Standard
- Properties of NativeError Instances
- The JSON Object
- The Debug Object
- Enumerator Objects
- VBArray Objects
- ActiveXObject Objects

1.3.1 Organization of This Documentation

This document is organized as follows:

- **Conditional Source Text Processing**: Processing of source text by JScript 5.x.
- **Extensions to Types:** Types defined by JScript 5.x that supplement types of [ECMA-262-1999].
- **Extensions to Statements:** A statement defined by JScript 5.x that supplements statements of [ECMA-262-1999].
- **Extensions to Native ECMAScript Objects**: Object extensions defined by JScript 5.x are listed according to object at the highest level.
- **Properties:** The object properties defined by JScript 5.x, typically functions, methods, or data formats, are described at the next levels.

1.4 Relationship to Standards and Other Extensions

This document defines extensions to [ECMA-262-1999]. Variations from [ECMA-262-1999] are defined in [MS-ES3].

The following documents describe variations and extensions from versions 3 and 5 of the ECMAScript Language:

Document Type	Reference	Title
Variations	[MS-ES3]	Internet Explorer ECMA-262 ECMAScript Language Specification Standards Support Document
Variations	[MS-ES5]	Internet Explorer ECMA-262 ECMAScript Language Specification (Fifth Edition) Standards Support Document
Extensions	[MS-ES3EX]	Microsoft JScript Extensions to the ECMAScript Language Specification Third Edition
Extensions	[MS-ES5EX]	Internet Explorer Extensions to the ECMA-262 ECMAScript Language Specification (Fifth Edition)

1.5 Applicability Statement

This document specifies a set of extensions to the [ECMA-262-1999] specifications. The extensions in this document provide access to some features that are unique to Internet Explorer when it loads a document in Quirks Mode, IE7 Mode, or IE8 Mode.

2 Extensions

This section specifies extensions to [ECMA-262-1999] that are available in Windows Internet Explorer 7, Windows Internet Explorer 8, Windows Internet Explorer 9, Windows Internet Explorer 10, Internet Explorer 11, and Internet Explorer 11 for Windows 10.

The extensions are as follows:

- Conditional Source Text Processing
- Extensions to Types
- Extensions to Statements
- Extensions to Native ECMAScript Objects

2.1 Conditional Source Text Processing

When converting source text into input elements, JScript 5.x first does the processing necessary to remove or replace any conditional text spans and then does the input element conversion using the results of that processing as the actual source text input to the identification of lexical input elements.

Each Program (see [ECMA-262-1999] section 14), whether presented as either a discrete source text or as the argument to the eval built-in function, and each FunctionBody (see [ECMA-262-1999] section 13) processed by the standard built-in Function constructor ([ECMA-262-1999] section 15.3.2.1) has conditional source text processing performed independently upon it.

NOTE

This specification defines conditional source text processing as if it were performed over an entire source text prior to any input element identification. It is an unobservable implementation detail whether this processing is actually performed in that manner or whether it is performed incrementally interweaved with input element identification.

2.1.1 Global State

The following state is shared by the conditional source text processing of all independent source texts that make up an ECMAScript program (see [ECMA-262-1999] section 14). The state is initialized prior to the first such processing as follows:

- *SubstitutionEnabled* Boolean flag with an initial value of **false**.
- *CCvariables* A set of association between string valued keys and values. The keys are strings. The values may be either ECMAScript Number ([ECMA-262-1999] section 8.5) or Boolean ([ECMA-262-1999] section 8.3) values. The initial associations are defined in the following table.

Key	Initial Value
"_win32"	Defined as **true** if this JScript 5.x implementation is a Microsoft 32-bit–based implementation. Otherwise, this association is not initially defined.
"_win64"	Defined as **true** if this JScript 5.x implementation is a Microsoft 64-bit–based implementation. Otherwise, this association is not initially defined.
"_x86"	Defined as **true** when running on a processor using the x86-based architecture. Otherwise, this association is not initially defined.

Key	Initial Value
"_ia64"	Defined as **true** when running on a processor using the Itanium 64-bit architecture. Otherwise, this association is not initially defined.
"_amd64"	Defined as **true** when running on a processor using the x64 architecture. Otherwise, this association is not initially defined.
"_jscript"	**true**.
"_jscript_build"	Number value that identifies the specific build of the JScript 5.x implementation that is running.
"_jscript_version"	Number value representing the version of the JScript 5.x language implementation. The value 5.7 indicates that the implementation only supports features of the JScript 5.7 language. The value 5.8 indicates that the implementation supports both 5.7 and 5.8 language features.
"_microsoft"	Defined as **true** when running on a JScript 5.x implementation provided by Microsoft. Otherwise, this association is not initially defined.

2.1.2 Conditional Processing Algorithm

For each source text to be processed, let source be the original source text (a sequence of Unicode characters) and let output initially be an empty sequence of Unicode characters. Let IfNestingLevel be 0.

Processing of *source* proceeds by recognizing specific input elements from *source* and then taking specified actions. The processing is organized into several states. The specific input elements that are recognized and the subsequent semantic action that is taken varies among states. The semantic action taken for a recognized input element may include transitioning to a different state. Processing of a source text begins by recognizing *CCInputElementState0* if *SubstitutionEnabled* is **false** and *CCInputElementState1* if *SubstitutionEnabled* is **true**.

The input elements for conditional processing are defined by the following grammar, which has Unicode characters as terminal symbols. Some rules of the grammar are defined using rules of the ECMAScript lexical grammar.

Syntax

NOTE:

CCInputElementState0 is recognized during top-level conditional processing when *SubstitutionEnabled* is false. When recognizing a *RegularExpressionLiteral* in this state, the contextual distinction between *RegularExpressionLiteral* and *DivPunctuator* (see [ECMA-262] section 7) must be respected.

CCInputElementState0 ::

RegularExpressionLiteral

StringLiteral

CCOn

CCSet0

CCIf0

CCMultiLineComment0

CCSingleLinecomment0

SourceCharacter

CCOn ::

@ CCOnId

/@ CCOnId*

//@ CCOnId

CCOnId ::

cc_on [lookahead ∉ IdentifierPart]

CCSet0 ::

@set [lookahead ∉ IdentifierPart]

CCIf0 ::

@if [lookahead ∉ IdentifierPart]

CCMultiLineComment0 ::

/ [lookahead ≠ CCOnId] MultiLineCommentCharsopt */*

SingleLineComment0 ::

// [lookahead ≠ CCOnId] SingleLineCommentCharsopt

Semantics

If *CCInputElementState0* cannot be recognized because there are no remaining characters in source, then Conditional Source processing is completed and the characters of the output supply the Unicode characters for subsequent input element processing. If *CCInputElementState0* cannot be recognized and there are characters in source a SyntaxError exception is raised.

The productions *CCInputElementState0* :: *RegularExpressionLiteral*, *CCInputElementState0* :: *StringLiteral*, *CCInputElementState0* :: *CCMultiLineComment0*, *CCInputElementState0* :: *CCSingleLinecomment0*, and *CCInputElementState0* :: *SourceCharacter* upon recognition perform the following actions:

0. Append to the end of output, in left-to-right sequence, the Unicode characters from source that were recognized by the production. Remove the recognized characters from source.

1. Use CCInputElementState0 to recognize the next input element from source.

The production *CCInputElementState0* :: *CCOn* upon recognition performs the following actions:

1. Set SubstitutionEnable to true.

2. Append a <SP> character to the end of output. Remove the recognized characters from source.

3. Use CCInputElementState1 to recognize the next input element from source.

The production *CCInputElementState0* :: *CCSet0* upon recognition performs the following actions:

1. Set SubstitutionEnable to true.

2. Append a <SP> character to the end of output. Remove the recognized characters from source.

3. Use CCInputElementStateSetLHS to recognize the next input element from source.

The production *CCInputElementState0* :: *CCIf0* upon recognition performs the following actions:

1. Set SubstitutionEnable to true.

2. Append a <SP> character to the end of output. Remove the recognized characters from source.

3. Increment the value of IfNestingLevel by 1.

4. Use CCInputElementStateIfPredicate to recognize the next input element from source.

Syntax

NOTE:

CCInputElementState1 is recognized during active conditional processing when *SubstitutionEnabled* is true. This may be at the top level or in the clause of an **@if** statement that represents the "true" condition. When recognizing a *RegularExpressionLiteral* in this state the contextual distinction between *RegularExpressionLiteral* and *DivPunctuator* (see [ECMA-262] section 7) must be respected.

CCInputElementState1 ::

> *RegularExpressionLiteralStringLiteralCCOnCCSet1CCIf1CCElif1CCElse1CCEnd1CCSubstitution1 CCStartMarkerCCEndMarkerCCMultiLineComment1CCSingleLinecomment1SourceCharacter*

CCSet1 ::

> *@set [lookahead ∉ IdentifierPart]/*@set [lookahead ∉ IdentifierPart]//@set [lookahead ∉ IdentifierPart]*

CCIf1 ::

> *@if [lookahead ∉ IdentifierPart]/*@if [lookahead ∉ IdentifierPart]//@if [lookahead ∉ IdentifierPart]*

CCElif1 ::

> *@elif [lookahead ∉ IdentifierPart]/*@elif [lookahead ∉ IdentifierPart]//@elif [lookahead ∉ IdentifierPart]*

CCElse1 ::

> *@else [lookahead ∉ IdentifierPart]/*@else [lookahead ∉ IdentifierPart]//@else [lookahead ∉ IdentifierPart]*

CCEnd1 ::

> *@end [lookahead ∉ IdentifierPart]/*@end [lookahead ∉ IdentifierPart]//@end [lookahead ∉ IdentifierPart]*

CCSubstitution1 ::

> *@ CCSubIdentifier/*@ CCSubIdentifier//@ CCSubIdentifier*

CCStartMarker ::

> */*@ //@*

CCEndMarker ::

@/*

CCMultiLineComment1 ::

/ [lookahead ≠ @] MultiLineCommentCharsopt */*

SingleLineComment1 ::

// [lookahead ≠ @] SingleLineCommentCharsopt

CCSubIdentifer ::

[lookahead ∉ CCKeyword] IdentifierName

CCKeyword ::

cc_on setifelif elseend

Semantics

If *CCInputElementState1* cannot be recognized because there are no remaining characters in source then Conditional Source processing is completed and the characters of the output supply the Unicode characters for subsequent input element processing. If *CCInputElementState1* cannot be recognized and there are characters in source a SyntaxError exception is raised.

The productions *CCInputElementState1* :: *RegularExpressionLiteral*, *CCInputElementState1* :: *StringLiteral*, *CCInputElementState1* :: *CCMultiLineComment1*, *CCInputElementState1* :: *CCSingleLinecomment1*, and *CCInputElementState1* :: *SourceCharacter* upon recognition perform the following actions:

1. Append to the end of output, in left-to-right sequence, the Unicode characters from source that were recognized by the production. Remove the recognized characters from source.

2. Use *CCInputElementState1* to recognize the next input element from source.

The productions *CCInputElementState1* :: *CCOn*, *CCInputElementState1* :: *CCStartMarker*, *CCInputElementState1* :: *CCEndMarker* upon recognition perform the following actions:

1. Append a <SP> character to the end of output. Remove the recognized characters from source.

2. Use *CCInputElementState1* to recognize the next input element from source.

The production *CCInputElementState1* :: *CCSet1* upon recognition performs the following actions:

1. Append a <SP> character to the end of output. Remove the recognized characters from source.

2. Use *CCInputElementStateSetLHS* to recognize the next input element from source.

The production *CCInputElementState1* :: *CCIf1* upon recognition performs the following actions:

1. Append a <SP> character to the end of output. Remove the recognized characters from source.

2. Increment the value of *IfNestingLevel* by 1.

3. Use *CCInputElementStateIfPredicate* to recognize the next input element from source.

The production *CCInputElementState1* :: *CCElif1* upon recognition performs the following actions:

1. Remove the recognized characters from source.

2. If *IfNestingLevel* is 0, raise a SyntaxError exception.

3. Use *CCInputElementStateFalseIfTail* to recognize the next input element from source.

The production *CCInputElementState1* :: *CCElse1* upon recognition performs the following actions:

1. Remove the recognized characters from source.
2. If *IfNestingLevel* is 0, raise a SyntaxError exception.
3. Use *CCInputElementStateFalseIfTail* to recognize the next input element from source.

The production *CCInputElementState1* :: *CCEnd* upon recognition performs the following actions:

1. Append a <SP> character to the end of output. Remove the recognized characters from source.
2. If *IfNestingLevel* is 0, raise a SyntaxError exception.
3. Decrement the value of *IfNestingLevel* by 1.
4. Use *CCInputElementState1* to recognize the next input element from source.

The production *CCInputElementState1* :: *CCSubstitution1* upon recognition performs the following actions:

1. Let var be the string of characters recognized as the *CCSubIdentifier* element of *CCSubstitution1*.
2. If the value of var is a key of *CCVariables*, then let the value be the associated value. Otherwise, let value be the string "NaN"
3. Let value be *ToString(value)*
4. Append the characters of the string value of value to the end of output.
5. Remove the recognized characters from source.
6. Use *CCInputElementStateIfPredicate* to recognize the next input element from source.

Syntax

NOTE:

CCInputElementStateSetLHS is recognized during active conditional processing of the body of an **@set** statement.

CCInputElementStateSetLHS ::

WhiteSpaceopt @ IdentifierName WhiteSpaceopt = CCExpression

Semantics

If *CCInputElementStateSetLHS* cannot be recognized a SyntaxError exception is raised.

The production *CCInputElementStateSetLHS* :: *WhiteSpaceopt* @ *IdentifierName WhiteSpaceopt* = *CCExpression* upon recognition performs the following actions:

1. Let *setName* be the string of characters recognized as the *IdentifierName* element of *CCSubstitution1*.
2. Let *value* be the result of evaluating *CCExpression*.
3. Create an association within *CCVariables* where the key is the string value of *setName* and where the value is *value*. If an association with that key already exists, replace it.
4. Remove the recognized characters from source.

5. Use *CCInputElementState1* to recognize the next input element from source.

Syntax

NOTE:

CCInputElementStateIfPredicate is recognized during active conditional processing of the predicate portion of an **@if** or **@elif** statement.

CCInputElementStateIfPredicate ::

WhiteSpaceopt (CCExpression WhiteSpaceopt)

Semantics

If *CCInputElementStateIfPredicate* cannot be recognized a SyntaxError exception is raised.

The production *CCInputElementStateSetIfPredicate* :: *WhiteSpaceopt* (*CCExpression WhiteSpaceopt*) upon recognition performs the following actions:

1. Let predicate be the result of evaluating *CCExpression*.
2. Increment the value of *IfNestingLevel* by 1.
3. Set *SkippedIfNestingLevel* to 0.
4. Remove the recognized characters from source.
5. If *ToBoolean*(predicate) is true, then use *CCInputElementState1* to recognize the next input element from source.
6. Otherwise, use *CCInputElementStateFalseThen* to recognize the next input element from source.

Syntax

NOTE:

CCInputElementStateFalseThen is recognized during processing of false clauses of an **@if** statement for which the true clause has not yet been processed. The current clause may be a "then" clause, an **@elif** clause, or an **@else** clause.

CCInputElementStateFalseThen ::

@if [lookahead ∉ IdentifierPart]@elif [lookahead ∉ IdentifierPart]@else [lookahead ∉ IdentifierPart]@end [lookahead ∉ IdentifierPart]SourceCharacter

Semantics

If *CCInputElementStateFalseThen* cannot be recognized a SyntaxError exception is raised.

The production *CCInputElementStateFalseThen* :: *@if* [lookahead ∉ IdentifierPart] upon recognition performs the following actions:

1. Increment the value of *SkippedIfNestingLevel* by 1.
2. Remove the recognized characters from source.
3. Use *CCInputElementStateFalseThen* to recognize the next input element from source.

The production *CCInputElementStateFalseThen* :: *@elif* [lookahead ∉ IdentifierPart] upon recognition performs the following actions:

1. Remove the recognized characters from source.

2. If *SkippedIfNestingLevel* > 0, then use *CCInputElementStateFalseThen* to recognize the next input element from source.

3. Otherwise, use *CCInputElementStateIfPredicate* to recognize the next input element from source.

The production *CCInputElementStateFalseThen* :: *@else* [lookahead ∉ IdentifierPart] upon recognition performs the following actions:

1. Remove the recognized characters from source.

2. If *SkippedIfNestingLevel* > 0, then use *CCInputElementStateFalseThen* to recognize the next input element from source.

3. Otherwise, use *CCInputElementState1* to recognize the next input element from source.

The production *CCInputElementStateFalseThen* :: *@end* [lookahead ∉ IdentifierPart] upon recognition performs the following actions:

1. Remove the recognized characters from source.

2. If *SkippedIfNestingLevel* is 0, then go to step 6.

3. Decrement the value of *SkippedIfNestingLevel* by 1.

4. Use *CCInputElementStateFalseThen* to recognize the next input element from source.

5. Return.

6. Decrement the value of *IfNestingLevel* by 1.

7. Use *CCInputElementState1* to recognize the next input element from source.

The production *CCInputElementStateFalseThen* :: *SourceCharacter* upon recognition performs the following actions:

1. Remove the recognized characters from source.

2. Use *CCInputElementStateFalseThen* to recognize the next input element from source.

Syntax

NOTE:

CCInputElementStateFalseThen is recognized during processing of false clauses of an **@if** statement for which the true clause has already been processed. It is also used during processing of all clauses of a @if statement that is nested within a false clause of an enclosing **@if** statement. The current clause may be a "then" clause, an **@elif** clause or an **@else** clause.

CCInputElementStateFalseIfTail ::

> *@if [lookahead ∉ IdentifierPart]@elif [lookahead ∉ IdentifierPart]@else [lookahead ∉ IdentifierPart]@end [lookahead ∉ IdentifierPart]SourceCharacter*

Semantics

If *CCInputElementStateFalseIfTail* cannot be recognized a SyntaxError exception is raised.

The production *CCInputElementStateFalseIfTail* :: *@if* [lookahead ∉ IdentifierPart] upon recognition performs the following actions:

1. Increment the value of *SkippedIfNestingLevel* by 1.

2. Remove the recognized characters from source.

3. Use *CCInputElementStateFalseIfTail* to recognize the next input element from source.

The productions *CCInputElementStateFalseIfTail* :: *@elif* [lookahead ∉ IdentifierPart] and *CCInputElementStateFalseIfTail* :: *@else* [lookahead ∉ IdentifierPart] upon recognition perform the following actions:

1. Remove the recognized characters from source.

2. Use *CCInputElementStateFalseIfTail* to recognize the next input element from source.

The production *CCInputElementStateFalseIfTail* :: *@end* [lookahead ∉ IdentifierPart] upon recognition performs the following actions:

1. Remove the recognized characters from source.

2. If *SkippedIfNestingLevel* is 0, then go to step 6.

3. Decrement the value of `SkippedIfNestingLevel` by 1.

4. Use *CCInputElementStateFalseIfTail* to recognize the next input element from source.

5. Return.

6. Decrement the value of *IfNestingLevel* by 1.

7. Use *CCInputElementState1* to recognize the next input element from source.

The production *CCInputElementStateFalseIfTail* :: *SourceCharacter* upon recognition performs the following actions:

1. Remove the recognized characters from source.

2. Use *CCInputElementStateFalseIfTail* to recognize the next input element from source.

Syntax

CCExpression ::

CCLogicalANDExpression

CExpression WhiteSpaceopt || *CCLogicalANDExpression*

CCLogicalANDExpression ::

CCBitwiseORExpressionCCcLogicalANDExpression WhiteSpaceopt && CCBitwiseORExpression

CCBitwiseORExpression ::

CCBitwiseXORExpressionCCBitwiseORExpression WhiteSpaceopt | CCBitwiseXORExpression

CCBitwiseXORExpression ::

CCBitwiseANDExpressionCCBitwiseXORExpression WhiteSpaceopt ^ CCBitwiseANDExpression

CCBitwiseANDExpression ::

CCEqualityExpressionCCBitwiseANDExpression WhiteSpaceopt & CCEqualityExpression

CCEqualityExpression ::

CCRelationalExpressionCCEqualityExpression WhiteSpaceopt ==
CCRelationalExpressionCCEqualityExpression WhiteSpaceopt!=
CCRelationalExpressionCCEqualityExpression WhiteSpaceopt ===
CCRelationalExpressionCCEqualityExpression WhiteSpaceopt !== CCRelationalExpression

CCRelationalExpression ::

CCShiftExpressionCCRelationalExpression WhiteSpaceopt <
CCShiftExpressionCCRelationalExpression WhiteSpaceopt >
CCShiftExpressionCCRelationalExpression WhiteSpaceopt <=
CCShiftExpressionCCRelationalExpression WhiteSpaceopt >= CCShiftExpression

CCShiftExpression ::

CCAdditiveExpressionCCShiftExpression WhiteSpaceopt <<
CCAdditiveExpressionCCShiftExpression WhiteSpaceopt >>
CCAdditiveExpressionCCShiftExpression WhiteSpaceopt >>> CCAdditiveExpression

CCAdditiveExpression ::

CCMultiplicativeExpressionCCAdditiveExpression WhiteSpaceopt +
CCMultiplicativeExpressionCCAdditiveExpression WhiteSpaceopt – CCMultiplicativeExpression

CCMultiplicativeExpression ::

*CCUnaryExpressionCCMultiplicativeExpression WhiteSpaceopt **
CCUnaryExpressionCCMultiplicativeExpression WhiteSpaceopt /
CCUnaryExpressionCCMultiplicativeExpression WhiteSpaceopt % CCUnaryExpression

UnaryExpression ::

CCPrimaryExpressionWhiteSpaceopt + CCUnaryExpressionWhiteSpaceopt –
CCUnaryExpressionWhiteSpaceopt ~ CCUnaryExpressionWhiteSpaceopt! CCUnaryExpression

CCPrimaryExpression ::

CCVariableCCLiteralWhiteSpaceopt (Expression)

CCLiteral ::

WhiteSpaceopt true [lookahead ∉ IdentifierPart]WhiteSpaceopt false [lookahead ∉ IdentifierPart]WhiteSpaceopt Infinity [lookahead ∉ IdentifierPart]WhiteSpaceopt NumericLiteral

CCVariable ::

WhiteSpaceopt @ IdentifierName

Semantics

Unless otherwise specified in this section, the productions of *CCExpression* are evaluated using the same semantic rules as the analogous productions of the ECMAScript syntactic grammar for Expression in [ECMA-262] section 11. However, only values of types Number and Boolean can occur during the evaluation of *CCExpression* productions so any semantic steps that are relative to other types of values are not relevant.

The production *CCLitera*l :: *WhiteSpaceopt* true [lookahead ∉ IdentifierPart] is evaluated by returning the value true.

The production *CCLiteral* :: *WhiteSpaceopt* false [lookahead ∉ IdentifierPart] is evaluated by returning the value false.

The production *CCLiteral* :: *WhiteSpaceopt* Infinity [lookahead ∉ IdentifierPart] is evaluated by returning the value +∞.

The production *CCVariable* :: WhiteSpaceopt @ *IdentifierName* is evaluated by performing the following steps:

1. Let var be the string of characters recognized as the *IdentifierName* element of *CCVariable*.
2. If the value of var is a key of *CCVariables*, then let value be the associated value. Otherwise, let value be NaN.
3. Return value.

2.2 Extensions to Types

JScript 5.x defines extensions to types of [ECMA-262-1999] that are described in the following sections.

2.2.1 SafeArray Type

The `SafeArray` type is the set of all references to Microsoft COM SAFEARRAY data structures.

SafeArray values can be created only by host objects and host functions. `SafeArray` values can be manipulated similarly to other ECMAScript data types.

2.2.2 VarDate Type

The `VarDate` type is the set of all references to Microsoft COM VARIANT data structures that have a VARTYPE enumeration value of VT_DATE.

`VarDate` values can be created only by host objects and host functions, or by calling the **getVarDate** method by using the **prototype** property of the **Date** object: **Date**.**prototype**.**getVarDate**. `VarDate` values can be manipulated similarly to other ECMAScript data types.

2.3 Extensions to Statements

JScript 5.x defines an extension to statements of [ECMA-262-1999] that is described in the following section.

2.3.1 debugger Statement

The **debugger** statement causes a breakpoint to be entered if a debugger is available. If a debugger does not exist or is not active, this statement has no observable effect.

Semantics

In JScript 5.x implementations, the **debugger** statement is evaluated as follows:

- If a debugger is not available or is not active for this statement, return (**normal**, **empty**, **empty**).
- Otherwise, suspend execution and enter the debugger.
- When the debugging action is complete, if the debugger supplies a completion result, return that result; otherwise, return (**normal**, **empty**, **empty**).

2.4 Extensions to Native ECMAScript Objects

JScript 5.x defines extensions to the native ECMAScript objects of [ECMA-262-1999]. These extensions are described in the following sections.

2.4.1 Function Properties of the Global Object

JScript 5.x defines additional properties of the **Global** object of [ECMA-262-1999]. These properties are described in the following sections.

2.4.1.1 ScriptEngine

When the **ScriptEngine** function is called, it returns a string value that specifies the implementation-defined name of the ECMAScript implementation that is executing the call. The JScript 5.x implementations within Internet Explorer 7 and Internet Explorer 8 always return the string 'JScript.'

2.4.1.2 ScriptEngineBuildVersion

When the **ScriptEngineBuildVersion** function is called, it returns a value that uniquely identifies the specific build of the ECMAScript implementation that is executing the call.

2.4.1.3 ScriptEngineMajorVersion

When the **ScriptEngineMajorVersion** function is called, it returns a value that identifies the major revision level of the implementation, not the revision level of the ECMAScript or JScript language specification that is currently supported by the implementation. This return value cannot be used as a reliable indicator of the availability or lack of availability of specific language features.

The JScript 5.x implementations within Internet Explorer 7 and Internet Explorer 8 always return a value of 5.

2.4.1.4 ScriptEngineMinorVersion

When the **ScriptEngineMinorVersion** function is called, it returns a value that identifies the minor revision level of the implementation, not the revision level of the ECMAScript or JScript language specification that is currently supported by the implementation. An implementation of JScript 5.x that supports distinct modes that separately implement JScript 5.7 and JScript 5.8 functionality may return a single value that does not vary among modes and that does not reflect the language level implemented by the current mode. This return value cannot be used as a reliable indicator of the availability or lack of availability of specific language features.

The JScript 5.x implementation within Microsoft Internet Explorer 7 always returns a value of 7. The JScript 5.x implementation within Microsoft Internet Explorer 8 always returns a value of 8, even when Internet Explorer 8 is operating in IE7 compatibility mode.

2.4.1.5 CollectGarbage

When the **CollectGarbage** function is called, the JScript 5.x implementation may attempt to reclaim unused or unneeded resources that are associated with the currently running application. Whether or not any action is actually taken depends on the current state of the execution environment and the resource management strategies and heuristics used by the implementation. An application may call this function to request that any such pending reclamation activities be completed immediately. However, a JScript 5.x implementation is not required to honor such a request.

2.4.1.6 RuntimeObject

The **RuntimeObject** function is used to search a global object for properties with names that match a specified pattern. The function only locates properties of the global object that were explicitly created by **VariableStatement** or **FunctionDeclaration** functions, or that were implicitly created by appearing as an identifier on the left side of an assignment operator. The function does not locate properties that were created by means of explicit property access on the global object.

When the **RuntimeObject** function is called, the following steps are taken:

0. If *pattern* is present, set *name* to "*" and go to step 6.
1. Call the function **toPrimitive**(pattern, hint Number).
2. If the type of Result(2) is not **String**, raise a **TypeError** exception.
3. If Result(2) is the empty string, set *name* to "*" and go to step 6.
4. Set *name* = *pattern*.
5. Set the values of both *leftWild* and *rightWild* to **false**.
6. If the first character of *name* is "*", let *leftWild* be **true**, and remove the first character from *name*.
7. If the last character of *name* is "*", let *rightWild* be **true**, and remove the last character from *name*.
8. Let *obj* be a new ECMAScript object created as if by the expression `new Object()`, where **Object** is the original built-in constructor with that name.
9. Let *enum* be an enumeration of the names of the properties of the global object.
10. Let *n* be the next element of enum. If there are no more elements, return *obj*.
11. If *n* is the name of a built-in property defined by [ECMA-262-1999] Section 15.1, or by the implementation or the host environment, go to step 11.
12. If *n* was not created by variable instantiation ([ECMA-262-1999] Section 10.1.3), or by an assignment operator in which the left side was the identifier *n*, go to step 11.
13. If *name* is the empty string, go to step 19.
14. Search for the first substring *name* within *n*, and let *left* be the position within *n* of the first character of the matched substring, and let *right* be the position within *n* of the last character of the matched substring.
15. If a substring match was not found, go to step 11.
16. If *leftWild* is **false** and left is not 1, go to step 11.
17. If *rightWild* is **false** and right is not the last character position of *n*, go to step 11.
18. Let value be the result of calling the [[Get]] property of the global object, passing n as the argument.
19. If *value* is **undefined**, go to step 11.
20. Call the [[Put]] method of *obj*, passing *n* and *value* as arguments.
21. Go to step 11.

The **length** property of the **RuntimeObject** function has a value of 1.

2.4.1.7 GetObject

The **GetObject** function is similar to the **ActiveXObject** constructor in that it provides a mechanism for creating and interacting with host objects provided by Microsoft Windows ActiveX automation servers. **GetObject** is used when a current automation object is already active, or if an automation object is to be retrieved from a file. When the **GetObject** constructor is called with one or more arguments, the following steps are taken:

1. Call **toPrimitive**(nameOrPath, hint Number).
2. If the type of Result(1) is not **String**, raise a **TypeError** exception.
3. If Result(1) is the empty string, raise a **TypeError** exception.
4. If name is not present, go to step 7.
5. Call the function **toPrimitive**(name, hint Number).
6. If the type of Result(5) is not **String**, raise a **TypeError** exception.
7. If only one argument was passed to this function, the string value of Result(1) may be an implementation-dependent file locator or an implementation-dependent automation object name. If two arguments were passed, Result(1) is a file locator, and Result(5) is the automation object name. If only one argument was passed, Step 8 first attempts to interpret Result(1) as a file path; if not successful, Step 8 attempts to interpret Result(1) as an automation object name.
8. Attempt to create or retrieve a host object that can be used to communicate with the application and application-specific object identified by Result(1) and Result(5).
9. If any error occurs during Step(8) such that the host object cannot be created or retrieved, raise an **Error** exception.
10. Return Result(8).

The format of the string values passed as arguments to this function are defined by the host operating system.

The **length** property of the **GetObject** function has a value of 1.

2.4.2 Constructor Properties of the Global Object

JScript 5.x defines the following additional constructor properties of the **Global** object:

- RegExpError
- ConversionError
- JSON
- Debug
- Enumerator
- VBArray
- ActiveXObject

2.4.3 Object Functions in JScript 5.8

The following two functions implement functionality similar to that of the like-named functions defined in the ECMAScript, 5th Edition Specification ([ECMA-262/5]). In the definition of these functions, the term "data property" means a normal ECMAScript 3rd Edition property as defined in [ECMA-262-1999] section 4.3.3. The term "accessor property" means a property that has two function objects associated with it, such that accessing the property using its object's **[[Get]]** and **[[Put]]** internal methods cause one of the functions to be implicitly invoked. The associated function that is invoked when the **[[Get]]** method is called is known as the "get" function of the accessor property. The value that the get function returns is used as the return value of the [[Get]] method. The associated function that is invoked when the **[[Put]]** method is called is known as the "set" function of the accessor property. The second argument of the **[[Put]]** method is passed as the argument to the set function.

2.4.3.1 Object.getOwnPropertyDescriptor (O, P)

This function is not defined for JScript 5.7. It exists only in JScript 5.8.

1. When the **getOwnProperty** function is called, the following steps are taken:
2. If the Type(*O*) is not Object, raise a **TypeError** exception.
3. If the *O* is not a host object that supports property access using this function, raise a **TypeError** exception.
4. Let *name* be ToString(*P*)
5. If *O* does not have an own property named *name*, return a new object created as if by evaluating the ECMAScript expressions: **{configurable:true,enumerable: true,value: undefined, writable: true}**
6. Let *desc* be a new object created as by evaluating the expression **{ }**.
7. If the own property named *name* of *O* has the DontEnum attribute, let *flag* be **true**; if it does not have the DontEnum attribute, let *flag* be **false**.
8. Call the **[[Put]]** method of *desc* passing **"enumerable"** and *flag* as arguments.
9. If the own property named *name* of *O* has the **DontDelete** attribute, let *flag* be **false**; if it does not, have the **DontEnum** attribute let *flag* be **true**.
10. Call the **[[Put]]** method of *desc* passing **"configurable"** and *flag* as arguments.
11. If the own property named *name* of *O* is an accessor property, go to step 16.
12. Let *value* be the current value of the own property named *name* of *O.*
13. Call the **[[Put]]** method of *desc* passing **"value"** and *value* as arguments.
14. If the own property named *name* of *O* has the ReadOnly attribute, let *flag* be **false**; if it does not have the ReadOnly attribute, let *flag* be **true**.
15. Call the **[[Put]]** method of *desc* passing **"writable"** and *flag* as arguments.
16. Return *desc*.
17. If the own accessor property named *name* of *O* has a defined get function, let *func* be that function object; otherwise, let *func* be **undefined**.
18. Call the **[[Put]]** method of *desc* passing **"get"** and *func* as arguments.

19. If the own accessor property named *name* of *O* has a defined set function, let *func* be that function object; otherwise, let *func* be **undefined**.

20. Call the **[[Put]]** method of *desc* passing **"set"** and *func* as arguments.

21. Return *desc*.

2.4.3.2 Object.defineProperty (O, P, Attributes)

This function is not defined for JScript 5.7. It exists only in JScript 5.8.

When the **defineProperty** function is called, the following steps are taken:

1. If the Type(*O*) is not Object, raise a **TypeError** exception.

2. If the *O* is not a host object that supports property creation using this function, raise a **TypeError** exception.

3. Let *name* be ToString(*P*).

4. Let *attrs* be ToObject(*Attributes*).

5. Let *enumerable* be **undefined**.

6. If the result of calling the **[[HasProperty]]** internal method of *O* with argument **"enumerable"** is **false**, go to step 9.

7. Let *val* be the result of calling the **[[Get]]** internal method of *O* with **"enumerable"**.

8. Let *enumerable* be ToBoolean(*val*).

9. Let *configurable* be **undefined**.

10. If the result of calling the **[[HasProperty]]** internal method of *O* with argument **"configurable"** is **false**, go to step 13.

11. Let *val* be the result of calling the **[[Get]]** internal method of *O* with **"configurable"**.

12. Let *configurable* be ToBoolean(*val*).

13. Let *valuePresent* be **false**.

14. If the result of calling the **[[HasProperty]]** internal method of *O* with argument **"value"** is **false**, go to step 17.

15. Let *value* be the result of calling the **[[Get]]** internal method of *O* with **"value"**.

16. Let *valuePresent* be **true**.

17. Let *writable* be **undefined**.

18. If the result of calling the **[[HasProperty]]** internal method of *O* with argument **"writable"** is **false**, go to step 21.

19. Let *val* be the result of calling the **[[Get]]** internal method of *O* with **"writable"**.

20. Let *writable* be ToBoolean(*val*).

21. Let *getPresent* be **false**.

22. If the result of calling the **[[HasProperty]]** internal method of *O* with argument **"get"** is **false**, go to step 27.

23. Let *getter* be the result of calling the **[[Get]]** internal method of *O* with **"get"**.
24. Let *getPresent* be **true**.
25. If *getter* is **undefined**, go to step 27.
26. If *getter* is not a function, raise a **TypeError** exception.
27. Let *setPresent* be **false**.
28. If the result of calling the **[[HasProperty]]** internal method of *O* with argument **"set"** is **false**, go to step 33.
29. Let *setter* be the result of calling the **[[Get]]** internal method of *O* with **"set"**.
30. Let *setPresent* be **true**.
31. If *setter* is **undefined**, go to step 33.
32. If *setter* is not a function, raise a **TypeError** exception.
33. If *getPresent* is **false**, let *setter* be **undefined**.
34. If *setPresent* is **false**, let *setter* be **undefined**.
35. If *O* does not have an own property named *name*, go to step 50.
36. If either *getPresent* or *setPresent* is true, go to step 44.
37. If *valuePresent* is **false**, return *O*.
38. If the own property named *name* of *O* is an accessor property, go to step 42.
39. If *writable* is **false**, raise a **TypeError** exception.
40. If *configurable* is **false**, raise a **TypeError** exception.
41. If *enumerable* is **false**, raise a **TypeError** exception.
42. Create a data property of *O* named *name* that has a value of *value* and with no attributes.
43. Return *O*.
44. If *configurable* is **false**, raise a **TypeError** exception.
45. If *enumerable* is **true**, raise a **TypeError** exception.
46. If *writable* is not **undefined**, raise a **TypeError** exception.
47. If *valuePresent* is **true**, raise a **TypeError** exception.
48. Create an accessor property of *O* named *name* that has a set function of *setter*, a get function of *getter*, and that has the **DontEnum** attribute.
49. Return *O*.
50. If the own property named *name* of *O* is an accessor property, go to step 65.
51. If either *getPresent* or *setPresent* is **true**, go to step 59.
52. If *valuePresent* is **false**, return *O*.
53. Call the **[[Put]]** method of *desc*, passing **"value"** and *value* as arguments.

54. If *configurable* is **false**, raise a **TypeError** exception.

55. If *writable* is **false**, raise a **TypeError** exception.

56. If *enumerable* is **false**, raise a **TypeError** exception.

57. Set the value of the data property of *O* named *name* to *value*.

58. Return *O*.

59. If *configurable* is **false**, raise a **TypeError** exception.

60. If *enumerable* is **true**, raise a **TypeError** exception.

61. If *writable* is not **undefined**, raise a **TypeError** exception.

62. If *valuePresent* is **true**, raise a **TypeError** exception.

63. Convert the own property of *O* named *name* into an accessor property that has a set function of *setter*, a get function of *getter*, and that has the DontEnum attribute.

64. Return *O*.

65. If *valuePresent* is **true**, go to step 73.

66. If neither *getPresent* nor *setPresent* is **true**, return *O*.

67. If *configurable* is **false**, raise a **TypeError** exception.

68. If *enumerable* is **true**, raise a **TypeError** exception.

69. If *writable* is not **undefined**, raise a **TypeError** exception.

70. If *setPresent* is **true**, set the set function of the accessor property of *O* named *name* to *setter*.

71. If *getPresent* is **true**, set the get function of the accessor property of *O* named *name* to *getter*.

72. Return *O*.

73. If either *getPresent* or *setPresent* is **true**, go to step 79.

74. If *configurable* is **false**, raise a **TypeError** exception.

75. If *writable* is **false**, raise a **TypeError** exception.

76. If *enumerable* is **false**, raise a **TypeError** exception.

77. Call the **[[Put]]** method of *O* passing *name* and *value* as arguments.

78. Return *O*.

79. If *configurable* is **false**, raise a **TypeError** exception.

80. If *enumerable* is **true**, raise a **TypeError** exception.

81. If *writable* is not **undefined**, raise a **TypeError** exception.

82. Raise a **TypeError** exception.

2.4.4 Properties of Function Instances

JScript 5.x defines additional properties of **Function** instances of [ECMA-262-1999]. These properties are described in the following sections.

2.4.4.1 The arguments Property

The value of the **arguments** property of a function instance is null. This property has the attributes { DontDelete, ReadOnly, DontEnum }. However, function instances also have a special **[[Get]]** internal method which in certain circumstances will return a value other than null when accessing the **arguments** property.

2.4.4.2 The caller Property

The value of the **caller** property of a function instance is null. This property has the attributes { DontDelete, ReadOnly, DontEnum }. However,function instances also have a special **[[Get]]** internal method which in certain circumstances will return a value other than null when accessing the **caller** property.

2.4.4.3 The [[Get]] (P) Method of a Function Object

Assume *F* is a Function object.

When the **[[Get]]** method of *F* is called with value *P*, the following steps are taken:

1. If *P* is not the string **'arguments'** then go to step 6.
2. If an active execution context for *F* does not exist, go to step 13.
3. Let *X* be the most recently created active execution context for *F*.
4. If *X* is marked as having a partially accessible arguments object, let *A* be the original arguments object for *X*; otherwise, let *A* be the value of the property named **'arguments'** of *X's* variable object.
5. Return *A*.
6. If *P* is not the string **'caller'**, go to step 13.
7. Let *X* be the most recently created active execution context for *F*.
8. If *X* does not have an execution context to which it could normally exit, return null.
9. Let *R* be the execution context which would become the current execution context if *X* exited normally (not via an exception).
10. If *R* is an execution context for a built-in function or a host object function, return null.
11. If *R* is an execution context for global code or for eval code, return null.
12. *R* must be an execution context for function code, so return the function object with the call that caused *R* to be created.
13. Return the result of calling the default **[[Get]]** method ([ECMA-262-1999] section 8.6.2.1) passing *P* as the argument.

Note: JScript 5.x under Internet Explorer 9 marks the current execution context as having a partially accessible arguments object when the function's *FormalParameterList* contains the name 'arguments' or the function's *FunctionBody* contains a direct reference to the function's original arguments object or the function's *FunctionBody* contains a direct call to **eval**.

JScript 5.x under Internet Explorer 7 or 8 marks the current execution context as having a partially accessible arguments object when the function's *FormalParameterList* contains the name 'arguments'.

2.4.5 String.prototype HTML Wrapper Properties

JScript 5.x defines **String**.**prototype** functions that wrap the string value of a **this** value with an HTML tag. The following abstraction is used to specify the behavior of these functions.

The abstract operation *WrapWithHTML* is called with arguments *body*, *tag*, *attribute*, and *data*. The *tag* and *attribute* arguments must be strings; *attribute* and *data* may be obmitted. The following steps are performed:

1. Append the character "<" to the characters of tag.
2. If attribute is not present, go to Step 7.
3. Append to Result(1) a single-space character followed by the characters of attribute.
4. Append to Result(3) the characters "=" and """.
5. Append to Result(4) the characters of the string returned by ToString(data).
6. Append to Result(5) the character """.
7. If attribute is present, use Result(6); otherwise, use Result(1).
8. Append to Result(7) the character ">".
9. Append to Result(8) the characters of the string returned by ToString(body).
10. Append to Result(9) the characters "<" and "/".
11. Append to Result(10) the characters of tag.
12. Append to Result(11) the character ">".
13. Return the string value of the characters from Result(12).

2.4.5.1 String.prototype.anchor(name)

Return the result of *WrapWithHTML*(**this** value, "A", "NAME", *name*).

2.4.5.2 String.prototype.big( )

Return the result of *WrapWithHTML*(**this** value, "BIG").

2.4.5.3 String.prototype.blink( )

Return the result of *WrapWithHTML*(**this** value, "BLINK").

2.4.5.4 String.prototype.bold( )

Return the result of *WrapWithHTML*(**this** value, "B").

2.4.5.5 String.prototype.fixed( )

Return the result of *WrapWithHTML*(**this** value, "TT").

2.4.5.6 String.prototype.fontcolor(color)

Return the result of *WrapWithHTML*(**this** value, "FONT", "COLOR", *color*).

2.4.5.7 String.prototype.fontsize(size)

Return the result of *WrapWithHTML*(**this** value, "FONT", "SIZE", *size*).

2.4.5.8 String.prototype.italics( )

Return the result of *WrapWithHTML*(**this** value, "I").

2.4.5.9 String.prototype.link(url)

Return the result of *WrapWithHTML*(**this** value, "A", "HREF", *url*).

2.4.5.10 String.prototype.small( )

Return the result of *WrapWithHTML*(**this** value, "SMALL").

2.4.5.11 String.prototype.strike( )

Return the result of *WrapWithHTML*(**this** value, "STRIKE").

2.4.5.12 String.prototype.sub( )

Return the result of *WrapWithHTML*(**this** value, "SUB").

2.4.5.13 String.prototype.sup( )

Return the result of *WrapWithHTML*(**this** value, "SUP").

2.4.6 Date Time String Format for JSON

This section is based upon the ECMAScript 5th Edition Specification, [ECMA-262/5]. The format defined here is used only by JScript 5.8 for the **Date.prototype.toJSON** method.

ECMAScript defines a string interchange format for date-times based upon a simplification of the [ISO-8601] Extended Format, which is YYYY-MM-DDTHH:mm:ss.sssZ

These fields are defined in the following table:

Field	Definition
YYYY	Decimal digits of the year in the Gregorian calendar.
-	The character "-" (hyphen) appears literally twice in the string.
MM	Month of the year from 01 (January) to 12 (December).
DD	Day of the month from 01 to 31.
T	The character "T" appears literally in the string, to indicate the beginning of the time element.

Field	Definition
HH	Number of complete hours that have passed since midnight as two decimal digits.
:	The character ":" (colon) appears literally twice in the string.
mm	Number of complete minutes since the start of the hour as two decimal digits.
Ss	Number of complete seconds since the start of the minute as two decimal digits.
.	The character "." (dot) appears literally in the string. The "." field may be omitted.
sss	Number of complete milliseconds since the start of the second as three decimal digits. The milliseconds field may be omitted.
Z	Time zone offset is specified as "Z" (for UTC), or either "+" or "-" followed by a time expression hh:mm

This format includes date-only forms:

YYYY

YYYY-MM

YYYY-MM-DD

It also includes time-only forms with an optional time zone offset appended:

THH:mm

THH:mm:ss

THH:mm:ss.sss

Also included are "date-times," which may be any combination of the above.

All numbers must be decimal (base 10).

Illegal values (out-of-bounds as well as syntax errors) in a format string means that the format string is not a valid instance of this format.

Because each day both starts and ends with midnight, the two notations 00:00 and 24:00 are available to distinguish the two midnights that can be associated with one date. This means that the following two notations refer to exactly the same moment in time: 1995-02-04T24:00 and 1995-02-05T00:00

There exists no international standard that specifies abbreviations for civil time zones such as CET, EST, PDT, and so on. Sometimes the same abbreviation is even used for two very different time zones. For this reason, [ISO-8601] and this format specify entirely numeric representations of date and time.

2.4.6.1 Extended Years

The ECMAScript 3rd Edition Specification [ECMA-262-1999] requires the ability to specify 6-digit years (extended years). This amounts to approximately 285,616 years, either forward or backward, from 01 January, 1970 UTC. To represent years before 0 or after 9999, [ISO-8601] permits the expansion of the year representation, but only by prior agreement between the sender and the receiver. In the simplified ECMAScript format, such an expanded year representation shall have 2 extra year digits and

is always prefixed with a plus (+) or minus (–) sign. The year 0 is considered positive and therefore is prefixed with a plus (+) sign.

2.4.6.2 Date.prototype.getVarDate ( )

The **getVarDate** method is implemented as follows:

1. Let *t* be the time value.
2. It *t* is **NaN**, return a date value in VT_DATE format for which the value of **ToNumber** is **NaN**.
3. Otherwise, return a date value in VT_DATE format that corresponds to the time value *t*.

2.4.6.3 Date.prototype.toJSON ( )

The **toJSON** method returns a **String** value that represents the instance in time that corresponds to the current **Date** object. All fields are present in the **String**. The time zone is always specified in UTC, denoted by the suffix **Z**. If this time value is not finite, `null` is returned.

This method is only defined for JScript 5.8.

2.4.7 Properties of the RegExp Constructor

JScript 5.x defines additional properties of the **RegExp** constructor of [ECMA-262-1999]. These properties are described in the following sections.

2.4.7.1 RegExp.index

The initial value of the **RegExp.index** property is the number −1. This property shall have the attributes { **DontEnum**, **DontDelete**, **ReadOnly** }. Even though this is a ReadOnly property, its value may be modified by calls to **RegExp.prototype.exec**.

2.4.7.2 RegExp.input

The initial value of **RegExp.input** is the empty string. This property shall have the attributes { **DontEnum**, **DontDelete** }. The value of this property may be modified by calls to **RegExp.prototype.exec**. The properties **RegExp.input** and **RegExp.$_** always have the same value. When one is set to some value, the other is automatically also set to that same value. Unlike most other **RegExp** constructor properties, this property is does not have the ReadOnly attribute.

2.4.7.3 RegExp.lastIndex

The initial value of **RegExp.lastIndex** is the number −1. This property shall have the attributes { **DontEnum**, **DontDelete**, **ReadOnly** }. Even though this is a ReadOnly property, its value may be modified by calls to **RegExp.prototype.exec**.

2.4.7.4 RegExp.lastMatch

The initial value of **RegExp.lastMatch** is the empty string. This property shall have the attributes { **DontEnum**, **DontDelete**, **ReadOnly** }. Even though this is a ReadOnly property, its value may be modified by calls to **RegExp.prototype.exec**.

2.4.7.5 RegExp.lastParen

The initial value of **RegExp.lastParen** is the empty string. This property shall have the attributes { **DontEnum**, **DontDelete**, **ReadOnly** }. Even though this is a ReadOnly property, its value may be modified by calls to **RegExp.prototype.exec**.

2.4.7.6 RegExp.leftContext

The initial value of **RegExp.leftContext** is the empty string. This property shall have the attributes { **DontEnum**, **DontDelete**, **ReadOnly** }. Even though this is a ReadOnly property, its value may be modified by calls to **RegExp.prototype.exec**.

2.4.7.7 RegExp.rightContext

The initial value of **RegExp.rightContext** is the empty string. This property shall have the attributes { **DontEnum**, **DontDelete**, **ReadOnly** }. Even though this is a ReadOnly property, its value may be modified by calls to **RegExp.prototype.exec**.

2.4.7.8 RegExp.$1 - RegExp.$9

The initial value of **RegExp.rightContext** is the empty string. This property shall have the attributes { **DontEnum**, **DontDelete**, **ReadOnly** }. Even though these are ReadOnly properties, their values may be modified by calls to **RegExp.prototype.exec**.

2.4.7.9 RegExp.$_

The initial value of each of the properties **RegExp.$1**, **RegExp.$2**, **RegExp.$3**, **RegExp.$4**, **RegExp.$5**, **RegExp.$6**, **RegExp.$7**, **RegExp.$8**, and **RegExp.$9** is the empty string. These properties shall have the attributes { **DontEnum**, **DontDelete**, **ReadOnly**}. The value of this property may be modified by calls to **RegExp.prototype.exec**. The properties **RegExp.input** and **RegExp.$_** always have the same value. When one of these properties is set to some value, the other is automatically also set to that same value. Unlike most other **RegExp** constructor properties, this property does not have the ReadOnly attribute.

2.4.7.10 RegExp['$&']

The initial value of **RegExp['$&']** is the empty string. This property shall have the attributes { **DontEnum**, **DontDelete**, **ReadOnly** }. Even though this is a ReadOnly property, its value may be modified by calls to **RegExp.prototype.exec**.

2.4.7.11 RegExp['$+']

The initial value of **RegExp['$+']** is the empty string. This property shall have the attributes { **DontEnum**, **DontDelete**, **ReadOnly** }. Even though this is a ReadOnly property, its value may be modified by calls to **RegExp.prototype.exec**.

2.4.7.12 RegExp["$`"]

The initial value of **RegExp["$`"**] is the empty string. This property shall have the attributes { **DontEnum**, **DontDelete**, **ReadOnly** }. Even though this is a ReadOnly property, its value may be modified by calls to **RegExp.prototype.exec**.

2.4.7.13 RegExp["$'"]

The initial value of **RegExp["$'"]** is the empty string. This property shall have the attributes { **DontEnum**, **DontDelete**, **ReadOnly** }. Even though this is a ReadOnly property, its value may be modified by calls to **RegExp.prototype.exec**.

2.4.8 Properties of the RegExp Prototype Object

JScript 5.x defines additional properties of the **RegExp** Prototype Object of [ECMA-262-1999]. These properties are described in the following sections.

2.4.8.1 RegExp.prototype.compile(pattern, flags)

If *pattern* is an object *R* that has a **[[Class]]** property **"RegExp"** and *flags* is **undefined**, let *P* be the *pattern* used to construct *R* and let *F* be the flags used to construct *R*. If *pattern* is an object *R* that has a **[[Class]]** property **"RegExp"** and *flags* is not **undefined**, raise a **RegExpError** exception. Otherwise, let *P* be the empty string if *pattern* is **undefined** and ToString(*pattern*) otherwise, and let *F* be the empty string if *flags* is **undefined** and ToString(*flags*) otherwise.

The **global** property of this **RegExp** object is set to a Boolean value that is **true** if *F* contains the character **"g"** and that is **false** otherwise.

The **ignoreCase** property of this **RegExp** object is set to a Boolean value that is **true** if *F* contains the character **"i"** and that is **false** otherwise.

The **multiline** property of this **RegExp** object is set to a Boolean value that is **true** if *F* contains the character **"m"** and that is **false** otherwise.

If *F* contains any character other than **"g"**, **"i"**, or **"m"**, raise a **RegExpError** exception.

If *P*'s characters do not have the form *Pattern,* raise a **RegExpError** exception. Otherwise, let the newly constructed object have a **[[Match]]** property obtained by evaluating ("compiling") *Pattern*. Note that evaluating *Pattern* may raise a **RegExpError** exception. (Note: if *pattern* is a *StringLiteral*, the usual escape sequence substitutions are performed before the string is processed by **RegExp**. If *pattern* must contain an escape sequence to be recognized by **RegExp**, the "****" character must be escaped within the *StringLiteral* to prevent its being removed when the contents of the *StringLiteral* are formed.)

The **source** property of this **RegExp** object is set as follows:

When *pattern* is an object *R* that has a **[[Class]]** property of **"RegExp"**, this **RegExp** object is set to the same string value as the value of the **source** property of *pattern*. Otherwise, the **source** property of this **RegExp** object is set to *P*.

The **lastIndex** property of this **RegExp** object is set to **0**.

The **options** property of this **RegExp** object is set as described in section 2.4.9.1.

This **RegExp** object is optimized using the assumption that it will be executed multiple times.

2.4.9 Properties of the RegExp Instances

JScript 5.x defines an additional property of the **RegExp** instances of [ECMA-262-1999]. This property is described in the following section.

2.4.9.1 options

The value of the **options** property is a string that specifies the values of the **global, ignoreCase**, and **multiline** properties of this **RegExp** instance. If the value of the **ignoreCase** property is **true**, the string contains the character "**i**". If the value of the **global** property is **true**, the string contains the character "**g**". If the value of the **multiline** property is **true**, the string contains the character "**i**". When present, the characters appear in the order "**igm**". If all of the **global, ignoreCase**, and **multiline** properties have the value **false**, the value of this property is the empty string. This property shall have the attributes { DontDelete, ReadOnly, DontEnum }.

2.4.10 The Error Constructor

JScript 5.x defines additional behaviors of the **Error** constructor of [ECMA-262-1999]. These behaviors are described in the following sections.

2.4.10.1 new Error ()

When the **Error** constructor is called with no arguments, the call is equivalent to calling the **Error** constructor and passing the number 0 as the only argument.

2.4.10.2 new Error(number, message)

When the **Error** constructor is called with two or more arguments, the following steps are taken:

0. The **[[Prototype]]** property of the newly constructed object is set to the original Error prototype object, the one that is the initial value of **Error.prototype** ([ECMA-262-1999] Section 15.11.3.1).
1. The **[[Class]]** property of the newly constructed **Error** object is set to **"Error"**.
2. Let *num* be **ToNumber**(*number*).
3. Let *msg* be **ToString**(*message*).
4. The **description** property of the newly constructed object is set to *msg*.
5. The **message** property of the newly constructed object is set to *msg*.
6. The **name** property of the newly constructed object is set to "**Error**".
7. The **number** property of the newly constructed object is set to *num*.
8. Return the newly constructed object.

2.4.11 Properties of Error Instances

JScript 5.x defines additional error instances inherited from the **[[Prototype]]** object of [ECMA-262-1999]. These error instances are described in the following sections.

2.4.11.1 description

The initial value of **description** is the same as the initial value of **message**.

2.4.11.2 number

An Error instance only initially has a **number** property if the first argument passed to the Error constructor was a number or could be converted to a number. The initial value of **number** is the number value passed to the constructor.

2.4.12 Native Error Types Used in This Standard

JScript 5.x defines additional native error types of [ECMA-262-1999]. These error instances are described in the following sections.

2.4.12.1 RegExpError

Indicates that a regular expression could not be parsed or that an error occurred while matching a regular expression. See [ECMA-262-1999] Sections 7.8.5, 15.10.2.2, 15.10.2.5, 15.10.2.15, 15.10.4.1, and 15.10.6.4.

2.4.12.2 ConversionError

This **NativeError** object is defined by JScript 5.x, but it is not raised by the JScript 5.x implementation or by any built-in objects.

2.4.13 Properties of NativeError Instances

Error instances inherit properties from their **[[Prototype]]** object and **Error** prototype as specified previously. In addition, those **NativeError** instances that are created to represent a runtime error that is detected by the JScript 5.x implementation have the following properties:

2.4.13.1 description

An **Error** instance only initially has a **description** property if it is created by the JScript 5.x implementation in response to the occurrence of a runtime error. The initial value of **description** is the same as the initial value of **message**.

2.4.13.2 number

An **Error** instance only initially has a **number** property if it is created by the JScript 5.x implementation in response to the occurrence of a runtime error. The initial value of **number** is the number value passed to the constructor.

2.4.14 The JSON Object

JScript 5.8 provides support for processing objects represented using the JSON Data Interchange Format. The JSON support in JScript 5.8 is an implementation of the JSON APIs defined in the ECMAScript 5th Edition Language Specification [ECMA-262/5]. The text in the sections that follow is a copy of the JSON specification text from clause 15.12 of [ECMA-262/5]. Additions or deletions to this text reflect variances between the JScript 5.8 JSON support and the original [ECMA-262/5] specification, and the differences between specification techniques used by the two base specifications.

The JSON object and its properties are not defined for JScript 5.7. They exist only in JScript 5.8.

The **JSON** object is a single object that contains two functions, **parse** and **stringify**, that are used to parse and construct JSON texts. The JSON Data Interchange Format is described in[RFC4627]. The JSON interchange format used in this specification is exactly that described by [RFC4627] with two exceptions:

1. The top level *JSONText* production of the ECMAScript JSON grammar may consist of any *JSONValue,* rather than being restricted to either a *JSONObject* or a *JSONArray* as specified by [RFC4627].

2. Conforming implementations of **JSON.parse** and **JSON.stringify** must support the exact interchange format described in this specification without any deletions or extensions to the format. This differs from [RFC4627], which permits a JSON parser to accept non-JSON forms and extensions.

The value of the **[[Prototype]]** internal property of the JSON object is the standard built-in Object prototype object ([ECMA-262-1999] Section 15.2.4). The value of the **[[Class]]** internal property of the JSON object is **"JSON"**. The value of the **[[Extensible]]** internal property of the JSON object is set to **true**.

The JSON object does not have a **[[Construct]]** internal property; it is not possible to use the JSON object as a constructor with the **new** operator.

The JSON object does not have a **[[Call]]** internal property; it is not possible to invoke the JSON object as a function.

2.4.14.1 The JSON Grammar

JSON.stringify produces a String that conforms to the following JSON grammar. **JSON.parse** accepts a String that conforms to the JSON grammar.

2.4.14.1.1 The JSON Lexical Grammar

JSON is similar to ECMAScript source text in that it consists of a sequence of characters conforming to the rules of *SourceCharacter*. The JSON Lexical Grammar defines the tokens that make up a JSON text similar to the manner that the ECMAScript lexical grammar defines the tokens of an ECMAScript source test. The JSON Lexical grammar recognizes only the white space character specified by the production *JSONWhiteSpace*. The JSON lexical grammar shares some productions with the ECMAScript lexical grammar. All nonterminal symbols of the grammar that do not begin with the characters "JSON" are defined by productions of the ECMAScript lexical grammar.

Syntax

JSONWhiteSpace **::**

<TAB><CR><LF><SP>

JSONString **::**

"*JSONStringCharacters*$_{opt}$ **"**

JSONStringCharacters **::**

JSONStringCharacter JSONStringCharacters$_{opt}$

JSONStringCharacter **::**

SourceCharacter **but not** *double-quote* **"** **or** *backslash* **** **or** *U+0000 thru U+001F*

**** *JSONEscapeSequence*

JSONEscapeSequence **::**

JSONEscapeCharacter

UnicodeEscapeSequence

JSONEscapeCharacter **:: one of**

" / \ b f n r t

JSONNumber **::**

-$_{opt}$ *DecimalIntegerLiteral JSONFraction*$_{opt}$ *ExponentPart*$_{opt}$

JSONFraction **::**

. [lookahead ∉ *DecimalDigit*]

. *DecimalDigits*

JSONNullLiteral **::**

NullLiteral

JSONBooleanLiteral **::**

BooleanLiteral

2.4.14.1.2 The JSON Syntactic Grammar

The JSON Syntactic Grammar defines a valid JSON text in terms of tokens defined by the JSON lexical grammar. The goal symbol of the grammar is *JSONText*.

Syntax

JSONText **:**

JSONValue

JSONValue **:**

JSONNullLiteralJSONBooleanLiteralJSONObjectJSONArrayJSONStringJSONNumber

JSONObject **:**

{ }
{ *JSONMemberList* **}**

JSONMember **:**

JSONString **:** *JSONValue*

JSONMemberList **:**

JSONMember JSONMemberList **,** *JSONMember*

JSONArray **:**

[ ]
[*JSONElementList* **]**

JSONElementList **:**

JSONValueJSONElementList **,** JSONValue

2.4.14.2 parse (text [, reviver])

The **parse** function parses a JSON text (a JSON-formatted String) and produces an ECMAScript value. The JSON format is a restricted form of ECMAScript literal. JSON objects are realized as ECMAScript objects. JSON arrays are realized as ECMAScript arrays. JSON strings, numbers, booleans, and null are realized as ECMAScript Strings, Numbers, Booleans, and **null**. JSON uses a more limited set of

white space characters than *WhiteSpace,* and allows Unicode code points U+2028 and U+2029 to directly appear in *JSONString* literals without using an escape sequence. The process of parsing is similar to [ECMA-262/5] sections 11.1.4 and 11.1.5 as constrained by the JSON grammar.

The optional *reviver* parameter is a function that takes two parameters, (*key* and *value*). It can filter and transform the results. It is called with each of the *key*/*value* pairs produced by the parse, and its return value is used instead of the original value. If it returns what it received, the structure is not modified. If it returns **undefined**, the property is deleted from the result.

1. Let *JText* be ToString(*text*).
2. Parse *JText* using the grammars in [ECMA-262/5] section 15.12.1. Raise a **SyntaxError** exception if *JText* did not conform to the JSON grammar for the goal symbol *JSONText*.
3. Let *unfiltered* be the result of parsing and evaluating *JText* as if it was the source text of an ECMAScript *program* (see [ECMA-262-1999] section 14) but using *JSONString* in place of *StringLiteral*. Note that since *JText* conforms to the JSON grammar, this result will be either a primitive value or an object that is defined by either an *ArrayLiteral* or an *ObjectLiteral*.
4. If (*reviver*) has a **[[Call]]** internal property, then
    1. Let *root* be a new object created as if by the expression **new Object()**, where **Object** is the standard built-in constructor with that name.
    2. Call the **[[Put]]** internal method of *root* with the empty String and *unfiltered* as arguments.
    3. Return the result of calling the abstract operation Walk, passing *root* and the empty String. The abstract operation Walk is described later in this section.
5. Else
    1. Return *unfiltered*.

The abstract operation Walk is a recursive abstract operation that takes two parameters: a *holder* object and the String *name* of a property in that object. Walk uses the value of *reviver* that was originally passed to the previous parse function.

1. Let *val* be the result of calling the **[[Get]]** internal method of *holder* with argument *name*.
2. If *val* is an object, then
    1. If the **[[Class]]** internal property of *val* is **"Array"**
        1. Set *I* to 0.
        2. Let *len* be the result of calling the **[[Get]]** internal method of *val* with argument **"length"**.
        3. Repeat while *I* < *len*,
            1. Let *newElement* be the result of calling the abstract operation Walk, passing *val* and ToString(*I*).
            2. If *newElement* is **undefined**, then
                1. Call the **[[Delete]]** internal method of *val* with ToString(*I*).
            3. Else
                1. Call the **[[Put]]** internal method of *val* with arguments ToString(*I*) and *newElement*.

4. Add 1 to *I*.

2. Else

    1. Let *keys* be an internal list of String values consisting of the names of all the own properties of *val* that do not have the DontEnum attribute. The ordering of the Strings should be the same as that used by the for-in statement.

        Note that JScript 5.x defines properties (see [ECMA-262-1999] 8.6.2.2) such that their DontEnum attribute is inherited from prototype properties with the same name. As a result of this, any own properties of *value* that have the same name as built-in properties that have the DontEnum attribute are not included in *keys*.

    2. For *each* String *P* in *keys* do,

        1. Let *newElement* be the result of calling the abstract operation Walk, *pa*ssing *val* and *P*.

        2. If *newElement* is **undefined**, then

            1. Call the **[[Delete]]** internal method of *val* with argument *P*.

        3. Else

            1. Call the **[[Put]]** internal method of *val* with arguments *P* and *newElement*.

    3. Return the result of calling the **[[Call]]** internal method of *reviver* passing *holder* as the **this** value and with an argument list consisting of *name* and *val*.

It is not permitted for a conforming implementation of **JSON.parse** to extend the JSON grammars. If an implementation wants to support a modified or extended JSON interchange format, it must do so by defining a different parse function.

NOTE: In the case where there are duplicate name Strings within an object, lexically preceding values for the same key shall be overwritten.

2.4.14.3 stringify (value [, replacer [, space]])

The **stringify** function returns a String in JSON format representing an ECMAScript value. It can take three parameters. The first parameter is required. The *value* parameter is an ECMAScript value, which is usually an object or array, although it can also be a String, Boolean, Number, or null. The optional *replacer* parameter is either a function that alters the way objects and arrays are stringified, or an array of Strings and Numbers that acts as a white list for selecting the object properties that will be stringified. The optional *space* parameter is a String or Number that allows the result to have white space injected into it to improve human readability.

These are the steps in stringifying an object:

1. Let *stack* be an empty List.

2. Let *indent* be the empty String.

3. Let *PropertyList* and *ReplacerFunction* be **undefined**.

4. If Type(*replacer*) is Object, then

    1. If *replacer* has a **[[Call]]** internal property, then

        1. Let *ReplacerFunction* be *replacer*.

    2. Else if the **[[Class]]** internal property of *replacer* is **"Array"**, then

1. Let *PropertyList* be an empty internal List.
2. For each value *v* of a property of *replacer* that has an array index property name. The properties are enumerated in the ascending array index order of their names.
    1. Let *item* be **undefined**.
    2. If Type(*v*) is String then let *item* be *v.*
    3. Else if Type(*v*) is Object then,
        1. If the **[[Class]]** internal property of *v* is **"String"** or **"Number"**, let *item* be ToString(*v*).
    4. If *item* is not undefined and *item* is not currently an element of *PropertyList* then,
        1. Append *item* to the end of *PropertyList*.

5. If Type(*space*) is Object then,
    1. If the **[[Class]]** internal property of *space* is **"Number"** then,
        1. Let *space* be ToNumber(*space*).
    2. Else if the **[[Class]]** internal property of *space* is **"String"** then,
        1. Let *space* be ToString(*space*).
6. If Type(*space*) is Number
    1. Let *space* be min(10, ToInteger(*space*)).
    2. Set *gap* to a String containing *space* space characters. This will be the empty String if *space* is less than 1.
7. Else if Type(*space)* is String
    1. If the number of characters in *space* is 10 or less, set *gap* to *space*; otherwise, set *gap* to a String consisting of the first 10 characters of *space*.
8. Else
    1. Set *gap* to the empty String.
9. Let *wrapper* be a new object created as if by the expression **new Object()**, where **Object** is the standard built-in constructor with that name.
10. Call the **[[Put]]** internal method of *wrapper* with arguments the empty String and *value*.
11. Return the result of calling the abstract operation *Str* with the empty String and *wrapper*.

The abstract operation *Str*(*key*, *holder*) has access to *ReplacerFunction* from the invocation of the **stringify** method. Its algorithm is as follows:

1. Let *value* be the result of calling the **[[Get]]** internal method of *holder* with argument *key*.
2. If Type(*value*) is Object, then
    1. If value is a host object, return **undefined**.
    2. Let *toJSON* be the result of calling the **[[Get]]** internal method of *value* with argument **"toJSON"**.

3. If *toJSON* has a **[[Call]]** internal property

    1. Let *value* be the result of calling the **[[Call]]** internal method of *toJSON*, passing *value* as the **this** value and with an argument list consisting of *key*.

3. If *ReplacerFunction* is not **undefined**, then

    1. Let *value* be the result of calling the **[[Call]]** internal method of *ReplacerFunction*, passing *holder* as the **this** value and with an argument list consisting of *key* and *value*.

4. If Type(*value*) is Object, then

    1. If the **[[Class]]** internal property of *value* is **"Number"**, then

        1. Let *value* be ToNumber(*value*).

    2. Else if the **[[Class]]** internal property of *value* is **"String"**, then

        1. Let *value* be ToString(*value*).

    3. Else if the **Class]]** internal property of *value* is **"Boolean"**, then

        1. Let *value* be the value of the **[[Value]]** internal property of *value*.

5. If *value* is **null** then return **"null"**.

6. If *value* is **true** then return **"true"**.

7. If *value* is **false** then return **"false"**.

8. If Type(*value*) is String, then return the result of calling the abstract operation *Quote* with argument *value*.

9. If Type(*value*) is Number

    1. If *value* is finite, return ToString(*value*).

    2. Else return **"null"**.

10. If Type(*value*) is Object, and *value* does not have a **[[Call]]** internal property

    1. If the **[[Class]]** internal property of *value* is **"Array"**, then

        1. Return the result of calling the abstract operation *JA* with argument *value*.

    2. Else, return the result of calling the abstract operation *JO* with argument *value*.

11. Return **undefined**.

The abstract operation *Quote*(*value*) wraps a String value in double quotation marks and escapes characters within it.

1. Let *product* be the double quotation mark character.

2. For each character *C* in *value*

    1. If *C* is the double quotation mark character or the backslash character

        1. Let *product* be the concatenation of *product* and the backslash character.

        2. Let *product* be the concatenation of *product* and *C*.

    2. Else if *C* is backspace, formfeed, newline, carriage return, or tab

1. Let *product* be the concatenation of *product* and the backslash character.
2. Let *abbrev* be the character corresponding to the value of *C* as follows:
    1. backspace **"b"**
    2. formfeed **"f"**
    3. newline **"n"**
    4. carriage return **"r"**
    5. tab **"t"**
3. Let *product* be the concatenation of *product* and *abbrev*.

3. Else if *C* is a control character having a code unit value less than the space character
    1. Let *product* be the concatenation of *product* and the backslash character.
    2. Let *product* be the concatenation of *product* and **"u"**.
    3. Let *hex* be the result of converting the numeric code unit value of *C* to a String of four hexadecimal digits.
    4. Let *product* be the concatenation of *product* and *hex*.
4. Else
    1. Let *product* be the concatenation of *product* and *C*.

3. Let *product* be the concatenation of *product* and the double quotation mark character.
4. Return *product*.

The abstract operation *JO*(*value*) serializes an object. It has access to the *stack*, *indent*, *gap*, *PropertyList*, *ReplacerFunction*, and *space* of the invocation of the **stringify** method.

1. If *stack* contains *value*, raise a **TypeError** exception because the structure is cyclical.
2. Append *value* to *stack*.
3. Let *stepback* be *indent*.
4. Let *indent* be the concatenation of *indent* and *gap*.
5. If *PropertyList* is not **undefined**, then
    1. Let *K* be *PropertyList*.
6. Else
    1. Let *K* be an internal List of Strings consisting of the names of all the own properties of *value* that do not have the DontEnum attribute. The ordering of the Strings should be the same as that used by the for-in statement.

       Note that JScript 5.x defines properties such that their DontEnum attribute is inherited from prototype properties with the same name. As a result of this, any own properties of value that have the same name as built-in properties that have the DontEnum attribute are not included in K.
7. Let *partial* be an empty List.

8. For each element *P* of *K*.
    1. Let *strP* be the result of calling the abstract operation *Str* with arguments *P* and *value*.
        1. If *PropertyList* is **undefined** and the call to *Str* caused new properties to be added to *value*, add the names of those properties to the end of *K*.
    2. If *strP* is not **undefined**
        1. Let *member* be the result of calling the abstract operation *Quote* with argument *P*.
        2. Let *member* be the concatenation of *member* and the colon character.
        3. If *gap* is not the empty String
            1. Let *member* be the concatenation of *member* and the *space* character.
        4. Let *member* be the concatenation of *member* and *strP*.
        5. Append *member* to *partial*.
9. If *partial* is empty, then
    1. Let *final* be "**{}**".
10. Else
    1. If *gap* is the empty String
        1. Let *properties* be a String formed by concatenating all the element Strings of *partial* with each adjacent pair of Strings separated with the comma character. A comma is not inserted either before the first String or after the last String.
        2. Let *final* be the result of concatenating "**{**", *properties*, and "**}**".
    2. Else if *gap* is not the empty String
        1. Let *separator* be the result of concatenating the comma character, the line feed character, and *indent*.
        2. Let *properties* be a String formed by concatenating all the element Strings of *partial* with each adjacent pair of Strings separated with *separator*. The *separator* String is not inserted either before the first String or after the last String.
        3. Let *final* be the result of concatenating "**{**", the line feed character, *indent*, *properties*, the line feed character, *stepback*, and "**}**".
11. Remove the last element of *stack*.
12. Let *indent* be *stepback*.
13. Return *final*.

The abstract operation *JA*(*value*) serializes an array. It has access to the *stack*, *indent*, *gap*, and *space* of the invocation of the stringify method. The representation of arrays includes only the elements between zero and **array.length** – 1 inclusive. Named properties are excluded from the stringification. An array is stringified as an open left bracket, elements separated by commas, and a closing right bracket.

1. If *stack* contains *value*, raise a **TypeError** exception because the structure is cyclical.
2. Append *value* to *stack*.

3. Let *stepback* be *indent*.
4. Let *indent* be the concatenation of *indent* and *gap*.
5. Let *partial* be an empty List.
6. Let *len* be the result of calling the **[[Get]]** internal method of value with argument **"length"**.
7. Let *index* be 0.
8. Repeat while *index* < *len*
    1. Let *strP* be the result of calling the abstract operation *Str* with arguments ToString(*index*) and *value*.
    2. If *strP* is **undefined**
        1. Append **"null"** to *partial*.
    3. Else
        1. Append *strP* to *partial*.
    4. Increment *index* by 1.
9. If *partial* is empty, then
    1. Let *final* be **"[]"**.
10. Else
    1. If *gap* is the empty String
        1. Let *properties* be a String formed by concatenating all the element Strings of *partial* with each adjacent pair of Strings separated with the comma character. A comma is not inserted either before the first String or after the last String.
        2. Let *final* be the result of concatenating **"[",** *properties*, and **"]".**
    2. Else
        1. Let *separator* be the result of concatenating the comma character, the line feed character, and *indent*.
        2. Let *properties* be a String formed by concatenating all the element Strings of *partial* with each adjacent pair of Strings separated with *separator*. The *separator* String is not inserted either before the first String or after the last String.
        3. Let *final* be the result of concatenating **"["**, the line feed character, *indent*, *properties*, the line feed character, *stepback*, and **"[**".
11. Remove the last element of *stack*.
12. Let *indent* be *stepback.*
13. Return *final.*

NOTE 1:

JSON structures are allowed to be nested to any depth, but they must be acyclic. If *value* is or contains a cyclic structure, the **stringify** function must raise a **TypeError** exception. This is an example of a value that cannot be stringified:

a = [];

a[0] = a;

my_text = JSON.stringify(a); // This must raise a TypeError.

NOTE 2:

Symbolic primitive values are rendered as follows:

- The **null** value is rendered in JSON text as the String `null`.
- The **undefined** value is not rendered.
- The **true** value is rendered in JSON text as the String `true`.
- The value is rendered in JSON text as the String `false`.

NOTE 3:

String values are wrapped in double quotes. The characters **"** and **** are escaped with **** prefixes. The characters **"** and **** are escaped with **** prefixes. Control characters are replaced with escape sequences **\uHHHH**, or with the shorter forms, **\b** (backspace), **\f** (formfeed), **\n** (newline), **\r** (carriage return), **\t** (tab).

NOTE 4:

Finite numbers are stringified as if by calling ToString**(***number***)**. **NaN** and Infinity regardless of sign are represented as the String **null**.

NOTE 5:

Values that do not have a JSON representation (such as **undefined** and functions) do not produce a String. Instead they produce the undefined value. In arrays, these values are represented as the String **null**. In objects, an unrepresentable value causes the property to be excluded from stringification.

NOTE 6:

An object is rendered as an opening left brace followed by zero or more properties, separated with commas, closed with a right brace. A property is a quoted String representing the key or property name, a colon, and the stringified property value. An array is rendered as an opening left bracket followed by zero or more values, separated with commas, closed with a right bracket.

This is the end of the JSON specification text from the [ECMA-262/5] standard.

2.4.15 The Debug Object

The **Debug** object is a single object that has some named properties, all of which are functions.

The value of the internal **[[Prototype]]** property of the **Debug** object is the **Object** prototype object (15.2.3.1). The value of the internal **[[Class]]** property of the **Debug** object is **"Object"**.

The **Debug** object does not have a **[[Construct]]** property; it is not possible to use the **Debug** object as a constructor with the **new** operator.

The **Debug** object does not have a **[[Call]]** property; it is not possible to invoke the **Debug** object as a function.

2.4.15.1 Function Properties of the Debug Object

The Debug object inherits properties from the Object prototype object as specified previously, and also has the following properties.

2.4.15.1.1 write ([item1 [, item2 [, ...]]])

If a host-dependent debugging facility is available, **ToString** is called once, in order, on each *item* argument. The result of the call is passed to the debugging facility with the intent that the result be output to the user without the addition of any line terminator characters. The function returns *undefined* regardless of whether or not a debugging facility is present.

2.4.15.1.2 writeln ([item1 [, item2 [, ...]]]))

If a host-dependent debugging facility is available, **ToString** is called once, in order, on each *item* argument. The result of the call is passed to the debugging facility with the intent that the result be output to the user without the insertion of any line terminator characters between item results. A line terminator should be output after the last *item* or if there are no *item* arguments. The function returns *undefined* regardless of whether a debugging facility is present.

The **length** property of the **write** function is *0*.

2.4.16 Enumerator Objects

Enumerator objects provide an alternative mechanism for iterating over the elements of **Array** instances and certain host objects.

For such objects, the order of enumeration is the same as occurs for the for-in statement ([ECMA-262-1999] Section 12.6.4)

2.4.16.1 The Enumerator Constructor Called as a Function

When **Enumerator** is called as a function rather than as a constructor, it returns **undefined**.

2.4.16.2 The Enumerator Constructor

When **Enumerator** is called as part of a **new** expression, it is a constructor: it initializes the newly created object.

2.4.16.2.1 new Enumerator ([collection])

When the **Enumerator** constructor is called with zero or one argument the following steps are taken:

1. If *collection* is not present, let *collection* be **undefined** and then go to step 6.
2. If *collection* is an Array instance, go to step 5.
3. If *collection* is a host object that supports an implementation-dependent enumeration protocol, go to step 5.
4. Raise a **TypeError** exception.
5. The **[[EnumerationState]]** property of the newly created object is set to a state indicating that the enumeration is at the first item of the enumeration of *collection*. If *collection* has no enumerable items, the state will indicate that the end of the enumeration has been reached.
6. The **[[Collection]]** property of the newly created object is set to *collection*.

7. The **[[Prototype]]** property of the newly constructed object is set to the original Error prototype object, the one that is the initial value of **Enumerator.prototype** (15.12+2.3.1).

8. The **[[Class]]** property of the newly constructed Error object is set to **"Object"**.

9. Return the newly constructed object.

2.4.16.3 Properties of the Enumerator Constructor

The value of the internal **[[Prototype]]** property of the **Enumerator** constructor is the **Function** prototype object ([ECMA-262-1999] Section 15.3.4).

The value of the **length** property is **7** (seven). In addition, the **Enumerator** constructor has the following property:

2.4.16.3.1 Enumerator.prototype

The initial value of **Enumerator.prototype** is the **Enumerator** prototype object (section 2.4.16.4).

This property has the attributes { DontEnum, DontDelete, ReadOnly }.

2.4.16.4 Properties of the Enumerator Prototype Object

The **Enumerator** prototype object is itself an **Enumerator** object with a **[[Collection]]** property of **undefined**, and which does not have an **[[EnumerationState]]** property.

The value of the internal **[[Prototype]]** internal property of the **Enumerator** prototype object is the **Object** prototype object ([ECMA-262/5] Section 15.2.3.1).

2.4.16.4.1 Enumerator.prototype.constructor

The initial value of **Enumerator.prototype.constructor** is the built-in **Enumerator** constructor.

2.4.16.4.2 Enumerator.prototype.atEnd ( )

If the **this** object is not an **Enumerator** object, raise a **TypeError** exception.

1. Let *collection* be the value of the **this** object's **[[Collection]]** property.

2. If *collection* is **undefined**, return **true**.

3. Let *state* be the value of the **this** object's **[[EnumerationState]]** property.

4. If *state* indicates that the end of the enumeration has been reached, return **true**.

5. Return **false**.

2.4.16.4.3 Enumerator.prototype.item ( )

If the **this** object is not an Enumerator object, raise a **TypeError** exception.

1. Let *collection* be the value of the **this** object's **[[Collection]]** property.

2. If *collection* is **undefined**, return **undefined**.

3. Let *state* be the value of the **this** object's **[[EnumerationState]]** property.

4. If *state* indicates that the end of the enumeration has been reached, return **undefined**.

5. Return the current enumeration item as indicated by *state*.

2.4.16.4.4 Enumerator.prototype.moveFirst ( )

If the **this** object is not an Enumerator object raise a **TypeError** exception.

1. Let *collection* be the value of the **this** object's **[[Collection]]** property.
2. If *collection* is **undefined**, return **undefined**.
3. Modify the **[[EnumerationState]]** property of the **this** object to a state indicating that the current enumeration of *collection* is now positioned at the original first item of the enumeration. If the current **[[EnumerationState]]** property indicates that the collection has no enumerable items, the new state will indicate that the end of the enumeration has been reached.
4. Return **undefined**.

2.4.16.4.5 Enumerator.prototype.moveNext ( )

If the **this** object is not an Enumerator object raise a **TypeError** exception.

1. Let *collection* be the value of the **this** object's **[[Collection]]** property.
2. If *collection* is **undefined**, return **undefined**.
3. Let *state* be the value of the **this** object's **[[EnumerationState]]** property.
4. If *state* indicates that the end of the enumeration has been reached, return **undefined**.
5. Modify *state* to a state indicating that the current enumeration of *collection* is now positioned at the next item beyond the current item of the enumeration. The new state may indicate that the end of the enumeration has been reached.
6. Update the **[[EnumerationState]]** property of the **this** object to *state*.
7. Return **undefined**.

2.4.16.5 Properties of Enumerator Instances

Enumerator instances inherit properties from their **[[Prototype]]** object as specified previously. In addition, **Enumerator** instances have an internal **[[Collection]]** property, and may have an internal **[[EnumeratorState]]** property.

2.4.17 VBArray Objects

Enumerator objects provide an alternative mechanism for iterating over the elements of **Array** instances and certain host objects.

For such objects, the order of enumeration is the same as the **for-in** statement ([ECMA-262-1999] section 12.6.4).

2.4.17.1 The VBArray Constructor Called as a Function

When **VBArray** is called as a function, it raises an exception if the argument is not a **SafeArray** value.

2.4.17.1.1 VBArray (value)

When the **VBArray** function is called, the following steps are taken:

1. If *Type(value)* is **SafeArray**, return *value*.

2. Raise a **TypeError** exception.

2.4.17.2 The VBArray Constructor

When **VBArray** is called as part of a new expression, it is a constructor: it initializes the newly created object.

2.4.17.2.1 new VBArray (value)

When the **VBArray** constructor is called with an argument value of zero or one, the following steps are taken:

1. If *Type(value)* is not **SafeArray**, raise a **TypeError** exception.

2. The **[[SArray]]** property of the newly created object is set to *value*.

3. The **[[Prototype]]** property of the newly constructed object is set to the initial value of the **VBArray prototype** object (section 2.4.17.3.1).

4. The **[[Class]]** property of the newly constructed **Error** object is set to Object.

5. Return the newly constructed object.

2.4.17.3 Properties of the VBArray Constructor

The value of the internal **[[Prototype]]** property of the **VBArray constructo**r is the **Function prototype** object (section 2.4.17.4).

The value of the **length** property is 1. In addition, the **VBArray constructor** has the **VBArray.prototype** property (section 2.4.17.3.1).

2.4.17.3.1 VBArray.prototype

The initial value of **VBArray.prototype** is the **VBArray prototype object** section 2.4.17.4.

This property has the attributes `DontEnum, DontDelete, ReadOnly`.

2.4.17.4 Properties of the VBArray Prototype Object

The VBArray prototype object is **VBArray** object with a **[[SArray]]** property that is a **SafeArray** that references a **COM SAFEARRAY** with *0* dimensions.

The value of the internal **[[Prototype]]** property of the **VBArray** prototype object is the **Object** prototype object ([ECMA-262-1999] section 15.2.3.1).

2.4.17.4.1 VBArray.prototype.constructor

The initial value of **VBArray.prototype.constructor** is the built-in **VBArray** constructor.

2.4.17.4.2 VBArray.prototype.dimensions ( )

1. Call **ToObject** passing the **this** value as the argument.

2. If *Result(1)* is not a **VBArray** instance, raise a **TypeError** exception.

3. Get the value of the **[[SArray]]** property of *Result(1)*.

4. Return the **Number** that is the number of dimensions of the **COM SAFEARRAY** referenced by Result(3).

2.4.17.4.3 VBArray.prototype.getItem (dim1 [, dim2, [dim3, ...]])

1. Call **ToObject** passing the **this** value as the argument.

2. If *Result(1)* is not a **VBArray** instance, raise a **TypeError** exception.

3. Get the value of the **[[SArray]]** property of *Result(1)*.

4. If no arguments were passed to this call, or if the number of arguments passed is greater than *Result(3)*, raise a **RangeError** exception.

5. For each argument *dim1* through *dimN*, let *IdimX* be **ToInteger**(*dimX*) where *X* is the numeric suffix of the argument name.

6. For each of *Idim1* through *IdimN*, if *IdimX* is less than the **lower** bound of dimension *X* of the **COM SAFEARRAY** referenced by *Result(3)* or if *IdimX* is greater than the **upper** bound of dimension *X*, raise a **RangeError** exception.

7. Return the value of the element identified by array indices *Idim1* through *IdimN* in the **COM SAFEARRAY** referenced by *Result(3)*.

The **length** property of the **getItem** function is 1.

2.4.17.4.4 VBArray.prototype.lbound ([dimension])

1. Call **ToObject** passing the **this** value as the argument.

2. If *Result(1)* is not a **VBArray** instance, raise a **TypeError** exception.

3. Get the value of the **[[SArray]]** property of *Result(1)*.

4. If *dimension* is not defined, use 1; otherwise, use **ToInteger**(*dimension*).

5. Get the Number that is the number of dimensions of the **COM SAFEARRAY** referenced by *Result(3)*.

6. If *Result(4)* is less than 1 or greater than *Result(5)*, raise a **RangeError** exception.

7. Return the Number that is the lower bound of dimension number *Result(4)* of the **COM SAFEARRAY** referenced by *Result(3*).

The **length** property of the **lbound** function is 0.

2.4.17.4.5 VBArray.prototype.toArray ( )

The method copies all the elements of a multi-dimensional **COM SAFEARRAY** into a one-dimensional **ECMAScript** Array instance. When called with no arguments, **toArray** performs the following steps:

1. Call **ToObject** passing the **this** value as the argument.

2. If Result(1) is not a **VBArray** instance, raise a **TypeError** exception.

3. Get the value of the **[[SArray]]** property of Result(1).

4. Let *SA* be the **COM SAFEARRAY** referenced by Result(3).

5. Let *dim* be the number of dimensions of the *SA*.

6. If *dim* is zero, return a new **Array** object that is created as if by evaluating the expression new Array(0) using the original **Array** constructor object.

7. Let *size* be the total number of array elements of *SA*.

8. Let *A* be a new **Array** object that is created as if by evaluating the expression new **Array**(*size*) using the original **Array** constructor object.

9. Access the elements of *SA* in row-major order and store the elements into the array indexed properties for *A* starting with property 0.

10. Return *A*.

2.4.17.4.6 VBArray.prototype.ubound ([dimension])

1. Call **ToObject** passing the **this** value as the argument.

2. If *Result(1)* is not a **VBArray** instance, raise a **TypeError** exception.

3. Get the value of the **[[SArray]]** property of *Result(1)*.

4. If *dimension* is not defined, use 1; otherwise, use **ToInteger**(*dimension*).

5. Get the Number that is the number of dimensions of the **COM SAFEARRAY** referenced by *Result(3)*.

6. If Result(4) is less than 1 or greater than Result(5), raise a **RangeError** exception.

7. Return the Number that is the upper bound of dimension number *Result(4)* of the **COM SAFEARRAY** referenced by *Result(3)*.

The **length** property of the **ubound** function is 0.

2.4.17.4.7 VBArray.prototype.valueOf ( )

1. Call **ToObject**, passing the **this** value as the argument.

2. If *Result(1)* is not a **VBArray** instance, raise a **TypeError** exception.

3. Get the value of the **[[SArray]]** property of *Result(1)*.

4. Return *Result(3)*.

2.4.17.5 Properties of VBArray Instances

VBArray instance inherits properties from the **[[Prototype]]** object as specified in **VBArray.prototype.valueOf ( )** section 2.4.17.4.7. In addition, **VBArray** instances have an internal **[[SArray]]** property with a value that is the **SafeArray** from which the instance was constructed.

2.4.18 ActiveXObject Objects

ActiveXObject objects provide a mechanism for creating and interacting with host objects provided by Microsoft Windows ActiveX automation servers.

2.4.18.1 The ActiveXObject Constructor Called as a Function

When **ActiveXObject** is called as a function, it performs the same argument validation that it performs when it is called as part of a new expression. After successfully completing validation, it always raises an **Error** exception.

2.4.18.1.1 ActiveXObject (name [, location]))

When the **ActiveXObject** function is called with one or more arguments, the following steps are taken:

1. Call **toPrimitive**(*name*, *hint Number*).
2. If the type *of Result(1)* is not **String**, raise a **TypeError** exception.
3. If *Result(1)* is an empty string, raise a **TypeError** exception.
4. If *location* is not present go to step 7.
5. Call **toPrimitive**(*location*, *hint Number*).
6. If the type of *Result(5)* is not **String**, raise a **TypeError** exception.
7. Raise an **Error** exception.

2.4.18.2 The ActiveXObject Constructor

When **ActiveXObject** is called as part of a new expression, it attempts to create a host object that corresponds to a Microsoft Windows ActiveX automation object.

2.4.18.2.1 new ActiveXObject ((name [, location]))

When the **ActiveXObject** constructor is called with one or more arguments, the following steps are taken:

1. Call **toPrimitive**(*name*, *hint Number*).
2. If the type of *Result(1)* is not **String**, raise a **TypeError** exception.
3. If *Result(1)* is an empty string, raise a **TypeError** exception.
4. If *location* is not present, go to step 7.
5. Call **toPrimitive**(*location*, *hint Number*).
6. If the type of *Result(5)* is not **String**, raise a **TypeError** exception.
7. Attempt to create a host object than can be used to communicate with the application and application-specific object identified by the **String** Result(1). If *location* was present, *Result(5)* identifies the server where the application resides; otherwise, the default server (the current machine) is used as the *location* of the application.
8. If any error occurs during Step 7, such that the host object cannot be created, raise an **Error** exception.
9. Return *Result(7)*.

The format of the string values passed as arguments to this constructor are defined by the host operating system.

The object returned by this constructor is a host object. It is not an instance of **ActiveXObject** and does not inherit properties from the **ActiveXObject** prototype object or from **Object.prototype**. The specific properties of such objects will vary and are dependent upon the specific argument values passed to this constructor.

2.4.18.3 Properties of the ActiveXObject Constructor

The value of the internal **[[Prototype]]** property of the **ActiveXObject** constructor is the Function prototype object ([ECMA-262-1999] section 15.3.4).

The value of the **length** property is **1**. In addition, the **ActiveXObject** constructor has the **ActiveXObject.prototype** property (section 2.4.18.3.1).

2.4.18.3.1 ActiveXObject.prototype

The initial value of **ActiveXObject.prototype** is the ActiveXObject prototype object ([ECMA-262-1999] section 15.12+3.4).

This property has the attributes *DontEnum, DontDelete, ReadOnly*.

The value of this property is not used by the **ActiveXObject** constructor. The value is not used as the **[[Prototype]]** value of host objects returned by the **ActiveXConstructor**.

2.4.18.4 Properties of the ActiveXObject Prototype Object

The **ActiveXObject prototype** object is an **Object** instance, not an **ActiveXObject** instance.

The value of the internal **[[Prototype]]** property of the **ActiveXObject prototype** object is the **Object prototype** object ([ECMA-262-1999] section 15.2.3.1).

2.4.18.4.1 ActiveXObject.prototype.constructor

The initial value of **ActiveXObject.prototype.constructor** is the built-in **ActiveXObject** constructor.

2.4.18.5 Properties of ActiveXObject Instances

ActiveXObject has no instances. Objects created by the **ActiveXObject** constructor are host objects that have properties which are determined by the external application associated with the specific host object.

3 Security Considerations

There are no additional security considerations.

4 Appendix A: Product Behavior

The information in this specification is applicable to the following Microsoft products or supplemental software. References to product versions include updates to those products.

- Windows Internet Explorer 7
- Windows Internet Explorer 8
- Windows Internet Explorer 9
- Windows Internet Explorer 10
- Internet Explorer 11
- Internet Explorer 11 for Windows 10

Exceptions, if any, are noted in this section. If an update version, service pack or Knowledge Base (KB) number appears with a product name, the behavior changed in that update. The new behavior also applies to subsequent updates unless otherwise specified. If a product edition appears with the product version, behavior is different in that product edition.

Unless otherwise specified, any statement of optional behavior in this specification that is prescribed using the terms "SHOULD" or "SHOULD NOT" implies product behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term "MAY" implies that the product does not follow the prescription.

5 Change Tracking

No table of changes is available. The document is either new or has had no changes since its last release.

6 Index

A

ActiveXObject Constructor
  newActiveXObject 52
ActiveXObject Constructor Properties 53
  prototype 53
ActiveXObject Constructor, The 52
ActiveXObject function 52
ActiveXObject Instances Properties 53
ActiveXObject Objects 51
ActiveXObject Prototype Object Properties 53
  constructor 53
Applicability 8

C

Change tracking 56
Conditional processing algorithm 10
Conditional source text processing 9

D

Debug Object 45
Debug Object Function Properties 46
  write 46
  writeIn 46

E

Enumerator Constructor Properties 47
  prototype 47
Enumerator Constructor, The 46
Enumerator Instances Properties 48
Enumerator Objects 46
Enumerator Prototype Object Properties 47
  atEnd 47
  constructor 47
  item 47
  moveFirst 48
  moveNext 48
Error Constructor 34
  newError () 34
  newError(number, message) 34
Error Instances Properties 34
  number 34

F

Function Instance Properties
  arguments 27
  caller 27
Function Instances 27
Function Object Methods
  [[Get]] 27

G

Global Object Function Properties
  CollectGarbage 20
  GetObject 22
  RuntimeObject 21
  ScriptEngine 20
  ScriptEngineBuildVersion 20
  ScriptEngineMajorVersion 20
  ScriptEngineMinorVersion 20
Global state 9
Glossary 6

I

Implementer - security considerations 54
Informative references 6
Introduction 6

J

JSON Grammar, The 36
JSON Lexical Grammar, The 36
JSON methods
  date time string format (section 2.4.6 29, section 2.4.6.1 30)
  getVarDate 31
  toSJON 31
JSON Object Functions
  parse 37
  stringify 39
JSON Object, The 35
JSON Syntactic Grammar, The 37

N

Native Error Instances Properties 35
  description 35
  number 35
Native Error Types 35
  ConversionError 35
  RegExpError 35
newEnumerator 46
Normative references 6

O

Object Functions 23
  defineProperty 24
  getOwnPropertyDescriptor 23
Objects
  Global 20
Overview (synopsis) 7

P

Product behavior 55

R

References 6
  informative 6
  normative 6
RegExp constructor 31
RegExp constructor properties
  index 31
  input 31
  lastIndex 31
  lastMatch 31

lastParen 32
leftContext 32
RegExp.$_ 32
RegExp.$1 – RegExp.$9 32
RegExp["$'"] 33
RegExp["$`"] 32
RegExp['$&'] 32
RegExp['$+'] 32
rightContext 32
RegExp Instances Properties 33
options 34
RegExp Prototype Object Properties (section 2.4.8 33, section 2.4.8.1 33)

S

Security - implementer considerations 54
Statements
debugger 19
String.prototype functions 28
anchor 28
big 28
blink 28
bold 28
fixed 28
fontcolor 29
fontsize 29
italics 29
link 29
small 29
strike 29
sub 29
sup 29

T

Tracking changes 56
Types
SafeArray 19
VarDate 19

V

VBArray (value) 48
VBArray Constructor
newArray (value) 49
VBArray Constructor Properties 49
VBArray.prototype 49
VBArray Constructor, The 49
VBArray Instances Properties 51
VBArray Objects 48
VBArray Prototype Object Properties 49
constructor (section 2.4.17.4.1 49, section 2.4.17.4.2 49)
getItem 50
lbound 50
toArray 50
ubound 51
valueOf 51