[MS-ES3]:

Microsoft JScript ECMA-262-1999 ECMAScript Language Specification Standards Support Document

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.
- **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.

Revision Summary

Date	Revision History	Revision Class	Comments
3/26/2010	1.0	New	Released new document.
5/26/2010	1.2	None	Introduced no new technical or language changes.
9/8/2010	1.3	Major	Significantly changed the technical content.
2/10/2011	2.0	Minor	Clarified the meaning of the technical content.
2/28/2011	2.1	Major	Significantly changed the technical content.
2/15/2012	2.2	Minor	Clarified the meaning of the technical content.
7/25/2012	3.0	Minor	Clarified the meaning of the technical content.
6/26/2013	4.0	Major	Significantly changed the technical content.
8/8/2013	4.1	Minor	Clarified the meaning of the technical content.
7/7/2015	4.3	Minor	Clarified the meaning of the technical content.
11/2/2015	4.3	None	No changes to the meaning, language, or formatting of the technical content.
3/22/2016	4.4	Minor	Clarified the meaning of the technical content.
4/19/2016	4.5	Minor	Clarified the meaning of the technical content.
11/2/2016	4.5	None	No changes to the meaning, language, or formatting of the technical content.
3/14/2017	4.5	None	No changes to the meaning, language, or formatting of the technical content.

Table of Contents

[MS-ES3] - v20170314
Microsoft JScript ECMA-262-1999 ECMAScript Language Specification Standards Support Document
Copyright © 2017 Microsoft Corporation
Release: March 14, 2017

1 Introduction

The JScript 5.x language is a dialect of the ECMAScript programming language. The JScript 5.x dialect is based upon the *ECMAScript Language Specification (Standard ECMA-262) Third Edition* [ECMA-262-1999] **,** published December 1999. This document describes the level of support provided by JScript 5.x for that specification.

There are several variants of the JScript 5.x language:

- JScript 5.7 first shipped with Windows® Internet Explorer® 7
- JScript 5.8 first shipped with Windows® Internet Explorer® 8

Within this document, JScript 5.x refers to any version of the JScript 5 language, beginning with JScript 5.7. JScript 5.7 and JScript 5.8 are used to refer to characteristics that are unique to those specific versions.

The [ECMA-262-1999] specifications contain guidance for authors of webpages, browser users, and user agents (browser applications). This conformance document considers only normative language from the related specifications that applies directly to user agents.

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

[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

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 Microsoft Implementations

The following Microsoft products implement some portion of the ECMAScript Third Edition specification [ECMA-262-1999]:

- Windows Internet Explorer 7 – implements the JScript 5.7 Language for all documents.
- Windows Internet Explorer 8 – implements the JScript 5.8 Language when loading documents in IE8 mode and the JScript 5.7 Language when loading documents in IE7 mode or quirks mode.
- Windows Internet Explorer 9 – implements the JScript 5.8 Language when loading documents in IE8 mode and the JScript 5.7 Language when loading documents in IE7 mode or quirks mode.
- Windows Internet Explorer 10 – implements the JScript 5.8 Language when loading documents in IE8 mode and the JScript 5.7 Language when loading documents in IE7 mode or quirks mode.
- Internet Explorer 11 – implements the JScript 5.8 Language when loading documents in IE8 mode and the JScript 5.7 Language when loading documents in IE7 mode or quirks mode.
- Internet Explorer 11 for Windows 10 – implements the JScript 5.8 Language when loading documents in IE8 mode and the JScript 5.7 Language when loading documents in IE7 mode or quirks mode.

Each version of Windows Internet Explorer implements multiple document modes, which can vary individually in their support of the standard. The following table lists the document modes available:

Browser Version	Document Modes Supported
Internet Explorer 7	Quirks mode (JScript 5.7) Standards mode (JScript 5.7)
Internet Explorer 8	Quirks mode (JScript 5.7) IE7 mode (JScript 5.7) IE8 mode (JScript 5.8)
Internet Explorer 9	Quirks mode (JScript 5.7) IE7 mode (JScript 5.7) IE8 mode (JScript 5.8)
Internet Explorer 10	Quirks mode (JScript 5.7) IE7 mode (JScript 5.7) IE8 mode (JScript 5.8)
Internet Explorer 11	Quirks mode (JScript 5.7) IE7 mode (JScript 5.7) IE8 mode (JScript 5.8)
Internet Explorer 11 for Windows 10	Quirks mode (JScript 5.7) IE7 mode (JScript 5.7) IE8 mode (JScript 5.8)

Throughout this document, JScript 5.x refers to any implementation of JScript 5.8 or JScript 5.7. "JScript 5.7" and "JScript 5.8" are used to refer to characteristics that are unique to implementations of those specific versions in the respective document modes in each version of Internet Explorer.

1.4 Conformance Requirements

To conform to [ECMA-262-1999] , a user agent must provide and support all the types, values, objects, properties, functions, and program syntax and semantics described in the specification (See [ECMA-262-1999] section 2, Conformance). Any optional portions that have been implemented must also be implemented as described by the specification. Normative language is usually used to define both required and optional portions. (For more information, see **[RFC2119]**.)

The following table lists the sections of [ECMA-262-1999] and whether they are considered normative or informative.

Sections	Normative/Informative
1	Informative
2-3	Normative
4	Informative
5-15	Normative
Annex A	Informative
Annex B	Informative

Relationship to Standards and Other Extensions

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 Notation

The following notations are used in this document to differentiate between notes of clarification, variation from the specification, and extension points:

Notation	Explanation
C####	This identifies a clarification of ambiguity in the target specification. This includes imprecise statements, omitted information, discrepancies, and errata. This does not include data formatting clarifications.
V####	This identifies an intended point of variability in the target specification such as the use of MAY, SHOULD, or RECOMMENDED. (See [RFC2119] .) This does not include extensibility points.
E####	Identifies extensibility points (such as optional implementation-specific data) in the target specification, which can impair interoperability.

Throughout this document, variations from the original [ECMA-262-1999] specification are indicated as follows:

- Double-underline – Text added to describe JScript 5.x behavior.
- ~~Strikethrough~~ – Portions that are not supported by JScript 5.x.

Underlined and strikethrough sections are used together to indicate where JScript 5.x differs from the behavior described in [ECMA-262-1999] .

For browser version and JScript version notation, see section **1.3**.

2 Conformance Statements

This section contains a full list of variations, clarification, and extension points in the Microsoft implementation of [ECMA-262-1999] .

- Section **2.1** includes only those variations that violate a MUST requirement in the target specification.
- Section **2.2** describes further variations from MAY and SHOULD requirements.
- Section **2.3** identifies variations in error handling.
- Section **2.4** identifies variations that impact security.

2.1 Normative Variations

The following sub-sections detail the normative variations from MUST requirements in [ECMA-262-1999] .

2.1.1 [ECMA-262-1999] Section 6, Source Text

V0001:

SourceCharacter **::**

any Unicode character

ECMAScript source text can contain any of the Basic Multilingual Plane Unicode characters. All Unicode white space characters are treated as white space~~, and all Unicode line/paragraph separators are treated as line separators~~. Non-Latin Unicode characters are allowed in identifiers, string literals, regular expression literals and comments.

V0002:

In string literals, regular expression literals and identifiers, any Basic Multilingual Plane character (code point) may also be expressed as a Unicode escape sequence consisting of six characters, namely **\u** plus four hexadecimal digits. Within a comment, such an escape sequence is effectively ignored as part of the comment. Within a string literal or regular expression literal, the Unicode escape sequence contributes one character to the value of the literal. Within an identifier, the escape sequence contributes one character to the identifier.

2.1.2 [ECMA-262-1999] Section 7, Lexical Conventions

V0003:

JScript 5.x also supports a "conditional compilation" feature which enables the inclusion of conditional text spans, within an ECMAScript source text, that are either not converted into input elements or which are replaced by alternative text spans prior to conversion into input elements. 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 input to the lexical conversion process described below.

2.1.3 [ECMA-262-1999] Section 7.1, Unicode Format-Control Characters

V0004:

The Unicode format-control characters (i.e., the characters in category "Cf" in the Unicode Character Database such as **LEFT-TO-RIGHT MARK** or **RIGHT-TO-LEFT MARK**) are control codes used to control the formatting of a range of text in the absence of higher-level protocols for this (such as mark-up languages). It is useful to allow these in source text to facilitate editing and display.

~~The format control characters can occur anywhere in the source text of an ECMAScript program. These characters are removed from the source text before applying the lexical grammar. Since these characters are removed before processing string and regular expression literals, one must use a. Unicode escape sequence (see 7.6) to include a Unicode format-control character inside a string or regular expression literal.~~

JScript 5.x does not remove category Cf characters from the source text before applying the lexical grammar.

2.1.4 [ECMA-262-1999] Section 7.3, Line Terminators

V0005:

The following characters are considered to be line terminators:

Code Point Value	Name	Formal Name
`\u000A`	Line Feed	<LF>
`\u000D`	Carriage Return	<CR>
~~`\u2028`~~	~~Line separator~~	~~<LS>~~
~~`\u2029`~~	~~Paragraph separator~~	~~<PS>~~

V0006:

LineTerminator **::**

<LF>
<CR>
~~<LS>~~
~~<PS>~~

JScript 5.x does not consider <LS> and <PS> to be line terminator characters.

2.1.5 [ECMA-262-1999] Section 7.4, Comments

V0007:

Syntax:

MultiLineNotAsteriskChar **::**

SourceCharacter **but not** *asterisk* ***** **or** <NUL>

MultiLineNotForwardSlashOrAsteriskChar **::**

SourceCharacter **but not** *forward-slash* **/** **or** *asterisk* ***** **or** <NUL>

2.1.6 [ECMA-262-1999] Section 7.5.3, Future Reserved Words

V0008:

The following words are used as keywords in proposed extensions and are therefore reserved to allow for the possibility of future adoption of those extensions.

Syntax:

FutureReservedWord **:: one of**

~~abstract~~	**enum**	**~~int~~**	**~~short~~**
~~boolean~~	**export**	**~~interface~~**	**~~static~~**
~~byte~~	**extends**	**~~long~~**	**super**
~~char~~	**~~final~~**	**~~native~~**	**~~synchronized~~**
class	**~~float~~**	**~~package~~**	**~~throws~~**
const	**~~goto~~**	**~~private~~**	**~~transient~~**
debugger	**~~implements~~**	**~~protected~~**	**~~volatile~~**
~~double~~	**import**	**~~Public~~**	

JScript 5.x only considers the following to be *FutureReservedWords*: class, const, debugger, enum, export, extends, import, super.

2.1.7 [ECMA-262-1999] Section 7.8.4, String Literals

V0009:

Syntax:

StringLiteral **::**

" *DoubleStringCharacters*$_{opt}$ "
' *SingleStringCharacters*$_{opt}$ '

DoubleStringCharacters **::**

DoubleStringCharacter DoubleStringCharacters$_{opt}$

SingleStringCharacters **::**

SingleStringCharacter SingleStringCharacters$_{opt}$

DoubleStringCharacter **::**

SourceCharacter **but not** *double-quote* **" or** *backslash* **\ or** *LineTerminator* **or** <NUL>
\ *EscapeSequence*
LineContinuation

SingleStringCharacter **::**

SourceCharacter **but not** *single-quote* **' or** *backslash* **\ or** *LineTerminator* **or** <NUL>
\ *EscapeSequence*
LineContinuation

JScript 5.x does not allow *StringLiterals* to contain the <NUL> (\u0000) character.

V0010:

LineContinuation **::**

\ *LineTerminatorSequence*

LineTerminatorSequence **::**

<LF>
<CR> [lookahead ≠ *<LF>*]
<CR> <LF>

EscapeSequence **::**

CharacterEscapeSequence
OctalEscapeSequence ~~**0** [lookahead ∉ *DecimalDigit*~~]
HexEscapeSequence
UnicodeEscapeSequence
8
9

JScript 5.x also supports *OctalEscapeSequence* as specified in [ECMA-262-1999] Annex B.1.2. That extension replaces the rule *EscapeSequence* **::** ***0*** [lookahead ∉ DecimalDigit] with the rule *EscapeSequence* **::** *OctalEscapeSequence.* See section 2.1.160**.**

V0011:

CharacterEscapeSequence **::**

SingleEscapeCharacter
NonEscapeCharacter

SingleEscapeCharacter **:: one of**

' " \ b f n r t ~~v~~

JScript 5.x does not consider **v** to be a *SingleEscapeCharacter*

V0012:

A string literal stands for a value of the String type. The string value (SV) of the literal is described in terms of character values (CV) contributed by the various parts of the string literal. As part of this process, some characters within the string literal are interpreted as having a mathematical value (MV), as described below or in [ECMA-262-1999] section 7.8.3.

- The SV of *StringLiteral* **:: ""** is the empty character sequence.
- The SV of *StringLiteral* **:: ''** is the empty character sequence.
- The SV of *StringLiteral* **:: "** *DoubleStringCharacters* **"** is the SV of *DoubleStringCharacters*.
- The SV of *StringLiteral* **:: '** *SingleStringCharacters* **'** is the SV of *SingleStringCharacters*.
- The SV of *DoubleStringCharacters* **::** *DoubleStringCharacter* is a sequence of one character, the CV of *DoubleStringCharacter*.
- The SV of *LineContinuation* **:: \** *LineTerminatorSequence* is the empty character sequence.

V0013:

- The CV of *EscapeSequence* **::** *CharacterEscapeSequence* is the CV of the *CharacterEscapeSequence*.
- The CV of *EscapeSequence* **:: 0** [lookahead ∉ *DecimalDigit*] is a <NUL> character (Unicode value 0000).
- The CV of *EscapeSequence* **::** *HexEscapeSequence* is the CV of *the HexEscapeSequence*.

- The CV of *EscapeSequence* **::** *UnicodeEscapeSequence* is the CV of the *UnicodeEscapeSequence*.
- The CV of *EscapeSequence* **:: 8** is an 8 character (Unicode value 0038).
- The CV of *EscapeSequence* **:: 9** is a 9 character (Unicode value 0039).

V0014:

- The CV of *CharacterEscapeSequence* **::** *SingleEscapeCharacter* is the character whose code point value is determined by the *SingleEscapeCharacter* according to the following table:

Escape Sequence	Code Point Value	Name	Symbol
\b	**\u0008**	backspace	<BS>
\t	**\u0009**	horizontal tab	<HT>
\n	**\u000A**	Line feed (new line)	<LF>
~~**\v**~~	~~**\u000B**~~	~~vertical tab~~	~~<VT>~~
\f	**\u000C**	form feed	<FF>
\r	**\u000D**	carriage return	<CR>
\"	**\u0022**	double quote	**"**
\'	**\u0027**	single quote	**'**
\\	**\u005C**	backslash	\

JScript 5.x does not consider **v** to be a *SingleEscapeCharacter*.

V0015:

NOTE

A 'LineTerminator' character cannot appear in a string literal, even if preceded by a backslash \. *The correct way to cause a line terminator character to be part of the string value of a string literal is to use an escape sequence such as* ***\n*** *or* ***\u000A***.

JScript 5.x allows a string literal to continue across multiple lines by including a \ as the last character of each continued line. The \ and the *LineTerminatorSequence* that follow it are not included in the value of the string literal.

2.1.8 [ECMA-262-1999] Section 7.8.5, Regular Expression Literals

V0016:

The productions below describe the syntax for a regular expression literal and are used by the input element scanner to find the end of the regular expression literal. The strings of characters comprising the *RegularExpressionBody* and the *RegularExpressionFlags* are passed uninterpreted to the regular expression constructor, which interprets them according to its own, more stringent grammar. An implementation may extend the regular expression constructor's grammar~~, but it should not extend the *RegularExpressionBody* and *RegularExpressionFlags* productions or the productions used by these productions~~.

Contrary to the above restriction, JScript 5.x extends the *RegularExpressionBody* production by excluding the occurrence of <NUL> as a *RegularExpressionchars* or *RegularExpressionFirstChar*. It also changes the *RegularExpressonFlags* production to exclude all possible flag characters other than `'g', 'i', and 'm'`

V0017:

Syntax:

RegularExpressionLiteral **::**

> **/** *RegularExpressionBody* **/** *RegularExpressionFlags*

RegularExpressionBody **::**

> *RegularExpressionFirstChar RegularExpressionChars*

RegularExpressionChars **::**

> [empty]
> *RegularExpressionChars RegularExpressionChar*

RegularExpressionFirstChar **::**

> *NonTerminator* **but not * or \ or / or** <NUL>
> *BackslashSequence*
> *RegularExpressionClass*

RegularExpressionChar **::**

> *NonTerminator* **but not \ or / or** <NUL>
> *BackslashSequence*
> *RegularExpressionClass*

BackslashSequence **::**

> **** *NonTerminator*

> JScript 5.x throws a **RegExpError** exception rather than a **SyntaxError** exception if the *NonTerminator* position of a *BackslashSequence* is occupied by a *LineTerminator*.

V0018:

Syntax:

NonTerminator **::**

> *SourceCharacter* **but not** *LineTerminator*

> JScript 5.x allows <LS> and <PS> to occur in regular expression literals because it does not consider them to be line terminator characters.

V0019:

Syntax:

RegularExpressionClass **::**

> **[** *RegularExpressionClassChars* **]**

RegularExpressionClassChars **::**

> [empty]
> *RegularExpressionClassChars RegularExpressionClassChar*

RegularExpressionClassChar **::**

NonTerminator **but not] or \ or** <NUL>
BackslashSequence

RegularExpressionFlags **::**

[empty]
RegularExpressionFlags ~~*IdentifierPart*~~ *RegExpFlag*

RegExpFlag **:: one of**

g i m

2.1.9 [ECMA-262-1999] Section 8, Types

V0020:

A value is an entity that takes on one of ~~nine~~ eleven types. There are ~~nine~~ eleven types (Undefined, Null, Boolean, String, Number, Object, SafeArray, VarDate, Reference, List, and Completion). Values of type Reference, List, and Completion are used only as intermediate results of expression evaluation and cannot be stored as properties of objects.

2.1.10 [ECMA-262-1999] Section 8.5, The Number Type

V0021:

In some implementations, external code might be able to detect a difference between various Not-a-Number values, but such behaviour is implementation-dependent; to ECMAScript code, all NaN values are indistinguishable from each other.

JScript 5.x does not normalize all internal NaN values to a single canonical NaN; therefore, external code may be able to observe multiple distinct NaN values.

2.1.11 [ECMA-262-1999] Section 8.6.2, Internal Properties and Methods

V0022:

The value of the **[[Prototype]]** property must be either an object or **null**, and every **[[Prototype]]** chain must have finite length (that is, starting from any object, recursively accessing the **[[Prototype]]** property must eventually lead to a **null** value). Whether or not a native object can have a host object as its **[[Prototype]]** depends on the implementation.

JScript 5.x does not permit a native object to have a host object as its **[[Prototype]]**.

2.1.12 [ECMA-262-1999] Section 8.6.2.2, [[Put]] (P, V)

V0023:

When the **[[Put]]** method of *O* is called with property *P* and value *V*, the following steps are taken:

1. Call the **[[CanPut]]** method of *O* with name *P.*
2. If Result(1) is **false**, return.
3. If *O* doesn't have a property with name *P,* go to step 6.
4. Set the value of the property to *V.* The attributes of the property are not changed.

5. Return.
6. Create a property with name *P*, set its value to *V* and give it empty attributes.
    1. Let *q* be the same value as *O*.
    2. Let *q* be the value of the **[[Prototype]]** property of *q*.
    3. If *q* is null, return.
    4. If *q* doesn't have a property with name *P*, go to step 6.2.
    5. If the property of *q* with name *P* does not have the DontEnum attribute, return.
    6. Give the property with the name *P* of *O* the DontEnum attribute.
7. Return.

In JScript 5.x a property created using **[[Put]]** is given the DontEnum attribute if it shadows a prototype property with the same name that already has the DontEnum attribute.

2.1.13 [ECMA-262-1999] Section 8.7, The Reference Type

V0024:

A **Reference** is a reference to a property of an object. A Reference consists of two components, the *base object* and the *property name.*

The following abstract operations are used in this specification to access the components of references:

- GetBase(V). Returns the base object component of the reference V; however if the type of the base object component is String return the result of calling ToObject with the base object component as the argument.
- GetPropertyName(V). Returns the property name component of the reference V.

2.1.14 [ECMA-262-1999] Section 8.7.1, GetValue (V)

V0025:

1. If Type(*V*) is not Reference, return *V*.
    1. If the type of the base object component of *V* is String, then go to step 6.
2. Call GetBase(*V*).
3. If Result(2) is **null**, throw a ~~**ReferenceError**~~ **TypeError** exception.
4. Call the **[[Get]]** method of Result(2), passing GetPropertyName(*V*) for the property name.
5. Return Result(4).
6. Let *str* be the String that is the base object component of *V*.
7. Call GetPropertyName(*V*).
8. If Result(6) is not an array index, then go to step 2.

9. Let *index* be ToUint32(Result(6)).

10. If *index* is greater or equal to the number of characters in *str*, then go to step 2

11. Return a String of length 1 that has as its only character the character at position *index* of *str*.

JScript 5.x throws a **TypeError** rather than **ReferenceError** when an attempt is made to get the value of a Reference value with a null base. This typically occurs when accessing an undeclared variable or function name.

Steps 6 to 11 permit the individual characters of a String value to be retrieved as if they were properties of an object. Note that JScript 5.x only supports property access to individual characters for String values. It does not support such property access for String wrapper objects.

2.1.15 [ECMA-262-1999] Section 9.1, ToPrimitive

V0026:

The operator ToPrimitive takes a Value argument and an optional argument *PreferredType*. The operator ToPrimitive converts its value argument to a non-Object type. If an object is capable of converting to more than one primitive type, it may use the optional hint *PreferredType* to favour that type. Conversion occurs according to the following table:

Input Type	Result
Undefined	The result equals the input argument (no conversion).
Null	The result equals the input argument (no conversion).
Boolean	The result equals the input argument (no conversion).
Number	The result equals the input argument (no conversion).
String	The result equals the input argument (no conversion).
SafeArray	The result equals the input argument (no conversion).
VarDate	The result equals the input argument (no conversion).
Object	Return a default value for the Object. The default value of an object is retrieved by calling the internal **[[DefaultValue]]** method of the object, passing the optional hint *PreferredType*. The behaviour of the **[[DefaultValue]]** method is defined by this specification for all native ECMAScript objects ([ECMA-262-1999] section 8.6.2.6).

2.1.16 [ECMA-262-1999] Section 9.2, To Boolean

V0027:

The operator ToBoolean converts its argument to a value of type Boolean according to the following table:

Input Type	Result
Undefined	**false**

Input Type	Result
Null	**false**
Boolean	The result equals the input argument (no conversion).
Number	The result is **false** if the argument is **+0**, -**0**, or **NaN**; otherwise the result is **true**.
String	The result is **false** if the argument is the empty string (its length is zero); otherwise the result is **true**.
SafeArray	**false**
VarDate	**false**
Object	**true**

2.1.17 [ECMA-262-1999] Section 9.3, ToNumber

V0028:

The operator ToNumber converts its argument to a value of type Number according to the following table:

Input Type	Result
Undefined	**NaN**
Null	**+0**
Boolean	The result is **1** if the argument is **true**. The result is **+0** if the argument is **false**.
Number	The result equals the input argument (no conversion).
String	See grammar and note below.
SafeArray	Throw a **TypeError** exception.
VarDate	The result is the Number that represents the internal numerical value of the VT Date value.
Object	Apply the following steps: 1. Call ToPrimitive(input argument, hint Number). 2. Call ToNumber(Result(1)). 3. Return Result(2).

2.1.18 [ECMA-262-1999] Section 9.8, ToString

V0029:

The operator ToString converts its argument to a value of type String according to the following table:

Input Type	Result
Undefined	**"undefined"**
Null	**"null"**
Boolean	If the argument is **true**, then the result is **"true"**. If the argument is **false**, then the result is **"false"**.
Number	See note below.
String	Return the input argument (no conversion)
SafeArray	Apply the following steps: 1. Call ToObject(input argument). 2. Call ToString(Result(1)). 3. Return Result(2).
VarDate	Return a String with contents representing the VarDate value, using the same representation format as that which is used by **Date.prototype.toString** ([ECMA-262-1999] section 15.9.5.2).
Object	Apply the following steps: 1. Call ToPrimitive(input argument, hint String). 2. Call ToString(Result(1)). 3. Return Result(2).

2.1.19 [ECMA-262-1999] Section 9.9, ToObject

V0030:

The operator ToObject converts its argument to a value of type Object according to the following table:

Input Type	Result
Undefined	Throw a **TypeError** exception.
Null	Throw a **TypeError** exception.
Boolean	Create a new Boolean object whose **[[value]]** property is set to the value of the boolean. See [ECMA-262-1999] section 15.6 for a description of Boolean objects.
Number	Create a new Number object whose **[[value]]** property is set to the value of the number. See [ECMA-262-1999] section 15.7 for a description of Number objects.
String	Create a new String object whose **[[value]]** property is set to the value of the string. See [ECMA-262-1999] section 15.5 for a description of String objects.
SafeArray	Create a new VBArray object as if by executing the ECMAScript expression: new VBArray(argument), where argument is the SafeArray value. See [MS-ES3EX] section **VBArray Objects** for a description of VBArray objects.
VarDate	Throw a TypeError exception.
Object	The result is the input argument (no conversion).

2.1.20 [ECMA-262-1999] Section 10.1.3, Variable Instantiation

V0031:

- For function code: for each formal parameter, as defined in the *FormalParameterList*, create a property of the variable object whose name is the *Identifier* and whose attributes are determined by the type of code. The values of the parameters are supplied by the caller as arguments to **[[Call]]**. If the caller supplies fewer parameter values than there are formal parameters, the extra formal parameters have value **undefined**. If two or more formal parameters share the same name, hence the same property, the corresponding property is given the value that was supplied for the last parameter with this name. If the value of this last parameter was not supplied by the caller, the value of the corresponding property is **undefined**. If any formal parameter has the name *arguments*, mark the current execution context as having a partially accessible arguments object. This state is used in [MS-ES3EX] section **VBArray Objects**.

  V0032:

- For each *FunctionDeclaration* or *FunctionExpression* in the code, in source text order, do one of the following depending upon the form of the *FunctionDeclaration* or *FunctionExpression*: ~~create a property of the variable object whose name is the Identifier in the FunctionDeclaration, whose value is the result returned by creating a Function object as described in [ECMA-262-1999] section 13, and whose attributes are determined by the type of code. If the variable object already has a property with this name, replace its value and attributes. Semantically, this step must follow the creation of FormalParameterList properties.~~

  - If the production is of the form *FunctionDeclaration* **: function (** *FormalParameterList$_{opt}$* **) {** *FunctionBody* **}** or *FunctionExpression* **: function (** *FormalParameterList$_{opt}$* **) {** *FunctionBody* **}** do nothing.

  - If the production is of the form *FunctionDeclaration* **: function** *Identifier* **(***FormalParameterList$_{opt}$* **) {** *FunctionBody* **}** or *FunctionExpression* **: function** *Identifier* **(** *FormalParameterList$_{opt}$* **) {** *FunctionBody* **}** create a property of the variable object whose name is the *Identifier* in the *FunctionDeclaration* or *FunctionExpression*, whose value is the result returned by creating a Function object as described in [ECMA-262-1999] section 13, and whose attributes are determined by the type of code. If the variable object already has a property with this name, replace its value and attributes. Semantically, this step must follow the creation of *FormalParameterList* properties.

  - If the production is of the form *FunctionDeclaration* **:** *JScriptFunction* or *FunctionExpression* **:** *JScriptFunction* perform the following steps:

    1. Let *func* be the result returned by creating a Function object as described in [ECMA-262-1999] section 13.

    2. Process the *FunctionBindingList* element of the *JScriptFunction* as described in [ECMA-262-1999] section 13 and using *func* and the attributes for the current type of code as processing arguments.

  In JScript 5.x each *FunctionExpression* is also included in the above processing step. This means that the value of such a *FunctionExpression* may be referenced by name within the code that contains it.

V0033:

- For each Catch, *VariableDeclaration* or *VariableDeclarationNoIn* in the code, create a property of the variable object whose name is the *Identifier* in the Catch, *VariableDeclaration* or *VariableDeclarationNoIn*, whose value is **undefined** and whose attributes are determined by the type of code. If there is already a property of the variable object with the name of a declared variable, the value of the property and its attributes are not changed. Semantically, this step must follow the creation of the *FormalParameterList* and *FunctionDeclaration* properties. In particular, if a declared variable has the same name as a declared function or formal parameter, the variable declaration does not disturb the existing property.

2.1.21 [ECMA-262-1999] Section 10.1.8, Arguments Object

V0034:

When control enters an execution context for function code, an arguments object is created and initialised as follows:

- The value of the internal **[[Prototype]]** property of the arguments object is the original Object prototype object, the one that is the initial value of **Object.prototype** (see [ECMA-262-1999] section 15.2.3.1).

- A property is created with name **callee** and property attributes { DontEnum }. The initial value of this property is the Function object being executed. This allows anonymous functions to be recursive.

- A property is created with name **caller** and property attributes { DontEnum }. Let *C* be the execution context that performed the call that caused the current execution context to be entered. The initial value of the **caller** property is **null** if *C* is an execution context for global code, eval code, or a built-in or host function object. Otherwise *C* is an execution context for function code and the initial value of the **caller** property is arguments object that was created when *C* was entered.

2.1.22 [ECMA-262-1999] Section 10.2, Entering an Execution Context

V0035:

> In JScript 5.x the sharing of storage between the properties of the arguments object and the corresponding properties to the activation object ceases when execution of the execution context that created the arguments object completes.

2.1.23 [ECMA-262-1999] Section 10.2.1, Global Code

V0036:

- The scope chain is created and initialised to contain the global object and no others.

- Variable instantiation is performed using the global object as the variable object and using property attributes { DontEnum, DontDelete }.

- The **this** value is the global object.

> JScript 5.x variable instantiations creates properties of the global object that have the **DontEnum** attribute.

2.1.24 [ECMA-262-1999] Section 10.2.2, Eval Code

V0037:

When control enters an execution context for eval code, the previous active execution context, referred to as the *calling context*, is used to determine the scope chain, the variable object, and the **this** value. If there is no calling context, then initialising the scope chain, variable instantiation, and determination of the **this** value are performed just as for global code.

> If the value of the eval property is used in any way other than a direct call as specified in [ECMA-262-1999] section 15.1.2.1, Jscript 5.x under Windows Internet Explorer 9 initializes the execution context as if it were the global execution context.

- The scope chain is initialised to contain the same objects, in the same order, as the calling context's scope chain. This includes objects added to the calling context's scope chain by **with** statements and **catch** clauses.
- Variable instantiation is performed using the calling context's variable object and using empty property attributes.
- The **this** value is the same as the **this** value of the calling context.

> In JScript 5.x an additional object with no properties is added to the front of the scope chain for eval code. This object is called the *eval scope*. Eval code may get, but may not put to, the value of a property of the calling context's variable object that has the name **arguments** and which is the actual arguments object of the calling context. The first time the eval code attempts to put to such a property a new property named **arguments** is added to the eval scope.

2.1.25 [ECMA-262-1999] Section 10.2.3, Function Code

V0038:

- The scope chain is initialised to contain the activation object followed by the objects in the scope chain stored in the **[[Scope]]** property of the Function object.
- Variable instantiation is performed using the activation object as the variable object and using property attributes { DontDelete }.
- The caller provides the **this** value. If the **this** value provided by the caller is **null** or **undefined**, or if the Type of the **this** value is VarDate ~~not an object (including the case where it is null)~~, then the **this** value is the global object. Otherwise, the result of calling ToObject with the caller-provided **this** value as the argument is used as the **this** value for the execution context.

> JScript 5.x performs ToObject conversion as part of establishing an execution context for function code rather than performing the conversions as part of the **Function.prototype.apply** and **Function.prototype.call** methods. Because of this difference, built-in functions and host functions may receive non-object values as their **this** value.

2.1.26 [ECMA-262-1999] Section 11.1.4, Array Initialiser

V0039:

Elision **:**

,
Elision **,**

Semantics

The production *ArrayLiteral* **: [** *Elision*$_{opt}$ **]** is evaluated as follows:

1. Create a new array as if by the expression **new Array()**.
2. Evaluate *Elision*; if not present, use the numeric value zero.
3. Call the **[[Put]]** method of Result(1) with arguments "**length**" and Result(2).
4. Return Result(1).

JScript 5.x sets the **length** property in step 3 to Result(2)+1. For example, an *ArrayLiteral* of the form [,] will have a length of 2 instead of 1 as specified above.

V0040:

The production *ArrayLiteral* **: [** *ElementList* **,** *Elision*$_{opt}$ **]** is evaluated as follows:

1. Evaluate *ElementList*.
2. Evaluate *Elision*; if not present, use the numeric value zero.
3. Call the **[[Get]]** method of Result(1) with argument "**length**".
4. Call the **[[Put]]** method of Result(1) with arguments "**length**" and (Result(2)+Result(3)).
5. Return Result(1).

If *Elision* is present, JScript 5.x uses the result of evaluating *Elision*+1 as Result(2). For example, an *ArrayLiteral* of the form [1,2,] has a length of 3 instead of 2 as specified above.

V0041:

The production *ElementList* **:** *Elision*$_{opt}$ *AssignmentExpression* is evaluated as follows:

1. Create a new array as if by the expression **new Array()**.
2. Evaluate *Elision*; if not present, use the numeric value zero.
3. Evaluate *AssignmentExpression*.
4. Call GetValue(Result(3)).
    1. If Result(4) is not the value undefined, go to step 5.
    2. Call the **[[Put]]** method of Result(1) with arguments **"length"** and (Result(2)+1).
    3. Return Result(1).
5. Call the **[[Put]]** method of Result(1) with arguments Result(2) and Result(4).
6. Return Result(1)

V0042:

The production *ElementList* **:** *ElementList* **,** *Elision*$_{opt}$ *AssignmentExpression* is evaluated as follows:

1. Evaluate *ElementList*.
2. Evaluate *Elision*; if not present, use the numeric value zero.
3. Evaluate *AssignmentExpression*.
4. Call GetValue(Result(3)).
5. Call the **[[Get]]** method of Result(1) with argument "**length**".
    1. If Result(4) is not the value **undefined**, go to step 6.
    2. If the browser is Windows Internet Explorer 7 or Windows Internet Explorer 8:
        1. Call the **[[Put]]** method of Result(1) with arguments **"length"** and (Result(2)+Result(5)+1).
        2. Return Result(1).
6. Call the **[[Put]]** method of Result(1) with arguments (Result(2)+Result(5)) and Result(4).
7. Return Result(1).

If the value of an *AssignmentExpression* in *ElementList* is **undefined**, JScript 5.x under Internet Explorer 7 or Internet Explorer 8 treats it as an elision. It does not create its own property of the array object corresponding to that array element. However, the length of the array is adjusted to include that element position.

2.1.27 [ECMA-262-1999] Section 11.1.5, Object Initialiser

V0043:

Syntax

ObjectLiteral **:**

{ }
{ *PropertyNameAndValueList* **}**
{ *PropertyNameAndValueList* **, }**

JScript 5.8 supports the occurrence of a single trailing comma as the last item within an *ObjectLiteral*. JScript 5.7 does not support this extension.

V0044:

Semantics

The productions *ObjectLiteral* **: {** *PropertyNameAndValueList* **}** and **{** *PropertyNameAndValueList* **, }** are ~~is~~ evaluated as follows:

1. Evaluate PropertyNameAndValueList.
2. Return Result(1);

2.1.28 [ECMA-262-1999] Section 11.2.1, Property Accessors

V0045:

The production *MemberExpression* **:** *MemberExpression* **[** *Expression* **]** is evaluated as follows:

1. Evaluate *MemberExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *Expression*.
4. Call GetValue(Result(3)).
5. If the type of Result(2) is String use Result(2), otherwise use the result of calling ~~Call~~ ToObject(Result(2)).
6. Call ToString(Result(4)).
7. Return a value of type Reference whose base object is Result(5) and whose property name is Result(6).

The change to step 5 is necessary to allow the individual characters of String values to be accessed as properties.

2.1.29 [ECMA-262-1999] Section 11.4.1, The delete Operator

V0046:

The production *UnaryExpression* **: delete** *UnaryExpression* is evaluated as follows:

1. Evaluate *UnaryExpression*.
    1. If *UnaryExpression* consists entirely of the identifier **this**, throw a **TypeError** exception.
2. If Type(Result(1)) is not Reference, ~~return **true**~~ throw a **TypeError** exception.
3. Call GetBase(Result(1)).
    1. If Result(3) is the global object, throw a **TypeError** exception.
4. Call GetPropertyName(Result(1)).
5. Call the **[[Delete]]** method on Result(3), providing Result(4) as the property name to delete.
6. Return Result(5).

In JScript 5.x, if *UnaryExpression* is the identifier **this** or an explicit reference to a property of the global object, a **TypeError** exception is thrown. For example, **delete this.prop** or **delete window.prop** would produce such an exception regardless of whether or not **prop** actually exists or how it was created. If *UnaryExpression* is a simple *Identifier* that resolves to a property of the global object, the above algorithm applies.

JScript also throws a **TypeError** if the value of the *UnaryExpression* is any Type of ECMAScript value other than Reference.

2.1.30 [ECMA-262-1999] Section 11.4.3, The typeof Operator

V0047:

The production *UnaryExpression* **: typeof** *UnaryExpression* is evaluated as follows:

1. Evaluate *UnaryExpression*.
2. If Type(Result(1)) is not Reference, go to step 4.
3. If GetBase(Result(1)) is **null**, return **"undefined"**.
4. Call GetValue(Result(1)).
5. Return a string determined by Type(Result(4)) according to the following table:

Type	Result
Undefined	**"undefined"**
Null	**"object"**
Boolean	**"boolean"**
Number	**"number"**
String	**"string"**
SafeArray	**"unknown"**
VarDate	**"date"**
Object (native and doesn't implement **[[Call]]**)	**"object"**
Object (native and implements **[[Call]]**)	**"function"**
Object (host)	Implementation-dependent JScript 5.x returns **object** for all host objects including those that implement **[[Call]]**.

2.1.31 [ECMA-262-1999] Section 11.6.1, The Addition Operator (+)

V0048:

The addition operator either performs string concatenation or numeric addition.

The production *AdditiveExpression* **:** *AdditiveExpression* **+** *MultiplicativeExpression* is evaluated as follows:

1. Evaluate *AdditiveExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *MultiplicativeExpression*.

4. Call GetValue(Result(3)).
5. Call ToPrimitive(Result(2)).
    1. If an exception was thrown during step 5 but not caught, return **undefined** (execution now proceeds as if no exception were thrown).
6. Call ToPrimitive(Result(4)).
    1. If an exception was thrown during step 6 but not caught, return Result(5) (execution now proceeds as if no exception were thrown).
7. If Type(Result(5)) is String *or* Type(Result(6)) is String, go to step 12. (Note that this step differs from step 3 in the comparison algorithm for the relational operators, by using *or* instead of *and*.)
8. Call ToNumber(Result(5)).
9. Call ToNumber(Result(6)).
10. Apply the addition operation to Result(8) and Result(9). See the note below ([ECMA-262-1999] section 11.6.3).
11. Return Result(10).
12. Call ToString(Result(5)).
13. Call ToString(Result(6)).
14. Concatenate Result(12) followed by Result(13).
15. Return Result(14).

The behavior described by steps 5.1 and 6.1 is an unintentional implementation defect that is present in all versions of JScript 5.x up to and including JScript 5.8.

2.1.32 [ECMA-262-1999] Section 11.8.2, The Greater-than Operator (>)

V0049:

The production *RelationalExpression* **:** *RelationalExpression* **>** *ShiftExpression* is evaluated as follows:

1. Evaluate *RelationalExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *ShiftExpression*.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(4) < Result(2) with the *LeftFirst* flag set to **false**. (see [ECMA-262-1999] section 11.8.5).
6. If Result(5) is **undefined**, return **false**. Otherwise, return Result(5).

ECMAScript generally uses a left-to-right evaluation order; however the [ECMA-262-1999] specification of the **>** operator results in an observable partial right-to-left evaluation order when the application of ToPrimitive on both operands has visible side effects. JScript 5.x implements strict left-to-right evaluation order for the operands of

>. Any ToPrimitive side effects caused by evaluating the left operand are visible before ToPrimitive is applied to the right operand.

2.1.33 [ECMA-262-1999] Section 11.8.3, The Less-than-or-equal Operator (<=)

V0050:

The production *RelationalExpression* **:** *RelationalExpression* **<=** *ShiftExpression* is evaluated as follows:

1. Evaluate *RelationalExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *ShiftExpression*.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(4) < Result(2) with the *LeftFirst* flag set to **false**. (see [ECMA-262-1999] section 11.8.5).
6. If Result(5) is **true** or **undefined**, return **false**. Otherwise, return **true**.

ECMAScript generally uses a left-to-right evaluation order; however, the ES3 specification of the **<=** operator results in an observable partial right-to-left evaluation order when the application of ToPrimitive on both operands has visible side effects. JScript 5.x implements strict left-to-right evaluation order for the operands of **<=**. Any ToPrimitive side effects caused by evaluating the left operand are visible before ToPrimitive is applied to the right operand.

2.1.34 [ECMA-262-1999] Section 11.8.5, The Abstract Relational Comparison Algorithm

V0051:

The comparison *x* < *y*, where *x* and *y* are values, produces **true**, **false**, or **undefined** (which indicates that at least one operand is **NaN**). In addition to x and y the algorithm takes a Boolean flag named *LeftFirst* as a parameter. The flag is used to control the order in which operations with potentially visible side effects are performed upon *x* and *y*. It is necessary to ensure left-to-right evaluation of expressions. The default value of *LeftFirst* is **true** and indicates that the *x* parameter corresponds to an expression that occurs to the left of the *y* parameter's corresponding expression. If *LeftFirst* is **false**, the reverse is the case and operations must be performed upon *y* before *x*. Such a comparison is performed as follows:

(The bulleted steps are added before step 1)

- If the *LeftFirst* flag is **true**, then
  - Let *px* be the result of calling Call ToPrimitive(*x*, hint Number).
  - Let *py* be the result of calling Call ToPrimitive(*x*, hint Number).
- Else the order of evaluation needs to be reversed to preserve let-to-right evaluation
  - Let *py* be the result of calling Call ToPrimitive(*x*, hint Number).
  - Let *px* be the result of calling Call ToPrimitive(*x*, hint Number).

1. ~~Call ToPrimitive(*x*, hint Number).~~ Use the value of *px*.
2. ~~Call ToPrimitive(*y*, hint Number).~~ Use the value of *py*.
3. If Type(Result(1)) is String and Type(Result(2)) is String, go to step 16. (Note that this step differs from step 7 in the algorithm for the addition operator **+** in using *and* instead of *or*.)
4. Call ToNumber(Result(1)).
5. Call ToNumber(Result(2)).
6. If Result(4) is **NaN**, return **undefined**.
7. If Result(5) is **NaN**, return **undefined**.
8. If Result(4) and Result(5) are the same number value, return **false**.
9. If Result(4) is **+0** and Result(5) is -**0**, return **false**.
10. If Result(4) is -**0** and Result(5) is **+0**, return **false**.
11. If Result(4) is **+∞**, return **false**.
12. If Result(5) is **+∞**, return **true**.
13. If Result(5) is -∞, return **false**.
14. If Result(4) is -∞, return **true**.
15. If the mathematical value of Result(4) is less than the mathematical value of Result(5)—note that these mathematical values are both finite and not both zero—return **true**. Otherwise, return **false**.
16. If Result(2) is a prefix of Result(1), return **false**. (A string value *p* is a prefix of string value *q* if *q* can be the result of concatenating *p* and some other string *r*. Note that any string is a prefix of itself, because *r* may be the empty string.)
17. If Result(1) is a prefix of Result(2), return **true**.
18. Let *k* be the smallest nonnegative integer such that the character at position *k* within Result(1) is different from the character at position *k* within Result(2). (There must be such a *k*, for neither string is a prefix of the other.)
19. Let *m* be the integer that is the code point value for the character at position *k* within Result(1).
20. Let *n* be the integer that is the code point value for the character at position *k* within Result(2).
21. If *m* < *n*, return **true**. Otherwise, return **false**.

2.1.35 [ECMA-262-1999] Section 11.9.3, The Abstract Equality Comparison Algorithm

V0052:

The comparison *x* == *y*, where *x* and *y* are values, produces **true** or **false**. Such a comparison is performed as follows:

(The bulleted steps are added before step 1)

- If Type(*x*) is SafeArray or Type(*y*) is SafeArray, return **false**.
- If Type(*x*) is VarDate or Type(*y*) is VarDate, return **false**.

1. If Type(*x*) is different from Type(*y*), go to step 14.
2. If Type(*x*) is Undefined, return **true**.
3. If Type(*x*) is Null, return **true**.
4. If Type(*x*) is not Number, go to step 11.
5. If *x* is **NaN**, return **false**.
6. If *y* is **NaN**, return **false**.
7. If *x* is the same number value as *y*, return **true**.
8. If *x* is **+0** and *y* is -**0**, return **true**.
9. If *x* is -**0** and *y* is **+0**, return **true**.
10. Return **false**.
11. If Type(*x*) is String, then return **true** if *x* and *y* are exactly the same sequence of characters (same length and same characters in corresponding positions). Otherwise, return **false**.
12. If Type(*x*) is Boolean, return **true** if *x* and *y* are both **true** or are both **false**. Otherwise, return **false**.
13. Return **true** if *x* and *y* refer to the same object or if they refer to objects joined to each other (see [ECMA-262-1999] section 13.1.2). Otherwise, return **false**.
14. If *x* is **null** and *y* is **undefined**, return **true**.
15. If *x* is **undefined** and *y* is **null**, return **true**.
16. If Type(*x*) is Number and Type(*y*) is String, return the result of the comparison *x* == ToNumber(*y*).
17. If Type(*x*) is String and Type(*y*) is Number, return the result of the comparison ToNumber(*x*) == *y*.
18. If Type(*x*) is Boolean, return the result of the comparison ToNumber(*x*) == *y*.
19. If Type(*y*) is Boolean, return the result of the comparison *x* == ToNumber(*y*).
20. If Type(*x*) is either String or Number and Type(*y*) is Object, return the result of the comparison *x* == ToPrimitive(*y*).
21. If Type(*x*) is Object and Type(*y*) is either String or Number, return the result of the comparison ToPrimitive(*x*) == *y*.
22. Return **false**.

For JScript 5.x, if either *x* or *y* is a host object then the "same object" determination in step 13 is implementation defined and dependent upon characteristics of the specific host objects. The method of determination used may be different from the "same object" determination made in step 13 of the Strict Equality Comparison Algorithm ([ECMA-262-1999] section 11.9.6). If *x* or *y* are host objects then interchanging their values may produce a different result.

2.1.36 [ECMA-262-1999] Section 11.9.6, The Strict Equality Comparison Algorithm

V0053:

The comparison *x* === *y*, where *x* and *y* are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If Type(*x*) is different from Type(*y*), return **false**.
    1. <u>If Type(*x*) is SafeArray or Type(*y*) is VarDate, return **false**.</u>
2. If Type(*x*) is Undefined, return **true**.
3. If Type(*x*) is Null, return **true**.
4. If Type(*x*) is not Number, go to step 11.
5. If *x* is **NaN**, return **false**.
6. If *y* is **NaN**, return **false**.
7. If *x* is the same number value as *y*, return **true**.
8. If *x* is **+0** and *y* is **-0**, return **true**.
9. If *x* is **-0** and *y* is **+0**, return **true**.
10. Return **false**.
11. If Type(*x*) is String, then return **true** if *x* and *y* are exactly the same sequence of characters (same length and same characters in corresponding positions); otherwise, return **false**.
12. If Type(*x*) is Boolean, return **true** if *x* and *y* are both **true** or are both **false**; otherwise, return **false**.
13. Return **true** if *x* and *y* refer to the same object or if they refer to objects joined to each other (see [ECMA-262-1999] section 13.1.2). Otherwise, return **false**.

> For JScript 5.x, if either *x* or *y* is a host object then the "same object" determination in step 13 is implementation defined and dependent upon characteristics of the specific host objects. The method of determination used may be different from the "same object" determination made in step 13 of the Abstract Equality Comparison Algorithm ([ECMA-262-1999] section 11.9.3). If *x* or *y* are host objects, the fact that step 13 returns true does not imply that step 13 of the Abstract Equality Comparison Algorithm would also return true for the same values. If *x* or *y* are host objects, interchanging their values may produce a different result.

2.1.37 [ECMA-262-1999] Section 12, Statements

V0054:

Syntax

Statement **:**

> *Block*
> *VariableStatement*
> *EmptyStatement*
> *ExpressionStatement*

IfStatement
IterationStatement
ContinueStatement
BreakStatement
ReturnStatement
WithStatement
LabelledStatement
SwitchStatement
ThrowStatement
TryStatement
DebuggerStatement
FunctionDeclaration

JScript 5.x allows a *FunctionDeclaration* to occur as a *Statement*.

2.1.38 [ECMA-262-1999] Section 12.1, Block

V0055:

Syntax

Block **:**

{ *StatementList$_{opt}$* **}**

In JScript 5.x any ambiguity between *Block* and the sequence *Block EmptyStatement* are resolved as *Block*.

V0056:

Semantics

The productions *Block* **: { }** and Block **: { };** are ~~is~~ evaluated as follows:

1. Return (**normal**, **empty**, **empty**).

The productions *Block* **: {** *StatementList* **}** and Block **: {** *StatementList$_{opt}$* **};** are ~~is~~ evaluated as follows:

1. Evaluate *StatementList*.
2. Return Result(1).

2.1.39 [ECMA-262-1999] Section 12.6.3, The for Statement

V0057:

The production *IterationStatement* **: for (***ExpressionNoIn$_{opt}$* **;** *Expression$_{opt}$* **;** *Expression$_{opt}$***)** *Statement* is evaluated as follows:

Step 1 below contains a specification error that is documented in ES3 Errata. JScript 5.x implements the following algorithm as corrected in the errata document.

1. If ~~the first Expression~~ *ExpressionNoIn* is not present, go to step 4.
2. Evaluate *ExpressionNoIn*.

3. Call GetValue(Result(2)). (This value is not used.)
4. Let *V* = **empty**.
5. If the first *Expression* is not present, go to step 10.
6. Evaluate the first *Expressio*n.
7. Call GetValue(Result(6)).
8. Call ToBoolean(Result(7)).
9. If Result(8) is **false**, go to step 19.
10. Evaluate *Statement*.
11. If Result(10).value is not **empty**, let *V* = Result(10).value
12. If Result(10).type is **break** and Result(10).target is in the current label set, go to step 19.
13. If Result(10).type is **continue** and Result(10).target is in the current label set, go to step 15.
14. If Result(10) is an abrupt completion, return Result(10).
15. If the second *Expression* is not present, go to step 5.
16. Evaluate the second *Expressio*n.
17. Call GetValue(Result(16). (This value is not used.)
18. Go to step 5.
19. Return (**normal**, *V*, **empty**).

V0058:

The production *IterationStatement* **: for (var** *VariableDeclarationListNoIn* **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement* is evaluated as follows:

Step 7 below contains a specification error that is documented in ES3 Errata. JScript 5.x implements the following algorithm as corrected in the errata document.

1. Evaluate *VariableDeclarationListNoIn*.
2. Let *V* = **empty**.
3. If the first *Expression* is not present, go to step 8.
4. Evaluate the first *Expression*.
5. Call GetValue(Result(4)).
6. Call ToBoolean(Result(5)).
7. If Result(6) is **false**, go to step ~~14~~17.
8. Evaluate *Statement*.
9. If Result(8).value is not **empty**, let *V* = Result(8).value.
10. If Result(8).type is **break** and Result(8).target is in the current label set, go to step 17.

11. If Result(8).type is **continue** and Result(8).target is in the current label set, go to step 13.
12. If Result(8) is an abrupt completion, return Result(8).
13. If the second *Expression* is not present, go to step 3.
14. Evaluate the second *Expression*.
15. Call GetValue(Result(14)). (This value is not used.)
16. Go to step 3.
17. Return (**normal**, *V*, **empty**).

2.1.40 [ECMA-262-1999] Section 12.6.4, The for-in Statement

V0059:

The production *IterationStatement* **: for (** *LeftHandSideExpression* **in** *Expression* **)** *Statement* is evaluated as follows:

1. Evaluate the *Expression*.
2. Call GetValue(Result(1)).
    1. If Type(Result(2) is VarDate, return (**normal, empty, empty**).
    2. If Result(2) is either **null** or **undefined**, return (**normal**, **empty**, **empty**).
3. Call ToObject(Result(2)).
4. Let *V* = **empty**.
5. Get the name of the next property of Result(3) that doesn't have the DontEnum attribute. If there is no such property, go to step 14.
6. Evaluate the *LeftHandSideExpression* (it may be evaluated repeatedly).
7. Call PutValue(Result(6), Result(5)).
8. Evaluate *Statement*.
9. If Result(8).value is not **empty**, let *V* = Result(8).value.
10. If Result(8).type is **break** and Result(8).target is in the current label set, go to step 14.
11. If Result(8).type is **continue** and Result(8).target is in the current label set, go to step 5.
12. If Result(8) is an abrupt completion, return Result(8).
13. Go to step 5.
14. Return (**normal**, *V*, **empty**).

V0060:

The production *IterationStatement* **: for (var** *VariableDeclarationNoIn* **in** *Expression* **)** *Statement* is evaluated as follows:

1. Evaluate *VariableDeclarationNoIn*.
2. Evaluate *Expression*.

    1. If Type(Result(2)) is VarDate, return (**normal, empty, empty**).
    2. If Result(2) is either **null** or **undefined**, return (**normal, empty, empty**).

3. Call GetValue(Result(2)).
4. Call ToObject(Result(3)).
5. Let *V* = **empty**.
6. Get the name of the next property of Result(4) that doesn't have the DontEnum attribute. If there is no such property, go to step 15.
7. Evaluate Result(1) as if it were an Identifier; see [ECMA-262-1999] 11.1.2. (yes, it may be evaluated repeatedly).
8. Call PutValue(Result(7), Result(6)).
9. Evaluate *Statement*.
10. If Result(9).value is not **empty**, let *V* = Result(9).value.
11. If Result(9).type is **break** and Result(9).target is in the current label set, go to step 15.
12. If Result(9).type is **continue** and Result(9).target is in the current label set, go to step 6.
13. If Result(8) is an abrupt completion, return Result(8).
14. Go to step 6.
15. Return (**normal**, *V*, **empty**).

In JScript 5.x no interations of the *Statement* are performed and no exception is thrown if the value of *Expression* is either **null** or **undefined**.

V0061:

The mechanics of enumerating the properties (step 5 in the first algorithm, step 6 in the second) is implementation dependent. The order of enumeration is defined by the object. Properties of the object being enumerated may be deleted during enumeration. If a property that has not yet been visited during enumeration is deleted, then it will not be visited. If new properties are added to the object being enumerated during enumeration, the newly added properties are not guaranteed to be visited in the active enumeration.

Enumerating the properties of an object includes enumerating properties of its prototype, and the prototype of the prototype, and so on, recursively; but a property of a prototype is not enumerated if it is "shadowed" because some previous object in the prototype chain has a property with the same name.

Note that JScript 5.x under Internet Explorer 7 or 8 defines properties such that their DontEnum attribute is inherited from prototype properties with the same name. As a result of this, any properties that have the same name as built-in properties of a prototype object that have the DontEnum attribute are not included in an enumeration. However JScript 5.x under Internet Explorer 9 includes the properties that have the same name as built-in properties of a prototype object in an enumeration.

In JScript 5.x the order of property enumeration is highly dependent upon dynamic characteristics of a program including the order in which properties are created and the order in which individual properties are accessed. These dynamic effects are most pronounced when enumerable properties are inherited from prototypes. For this reason, is not possible to provide a generalized specification of properties enumeration order that applies to all objects. However, in JScript 5.x, if an object inherits no enumerable properties from its prototypes, the object's properties will be enumerated in the order in which they were created. The order of property enumeration in JScript 5.x under Internet Explorer 9 may be different from the order returned by JScript 5.x under Internet Explorer 7 or 8.

2.1.41 [ECMA-262-1999] Section 12.11, The switch Statement

V0062:

Semantics

The ES3 errata state that the following algorithm contains many errors. JScript 5.x implements the revised algorithms provided by the errata document.

~~The production *CaseBlock* **: {** *CaseClauses DefaultClause CaseClauses* **}** is given an input parameter, *input*, and is evaluated as follows:~~

1. ~~Let *A* be the list of *CaseClause* items in the first *CaseClauses*, in source text order.~~
2. ~~For the next *CaseClause* in *A*, evaluate *CaseClause*. If there is no such *CaseClause*, go to step 7.~~
3. ~~If *input* is not equal to Result(2), as defined by the **!==** operator, go to step 2.~~
4. ~~Evaluate the *StatementList* of this *CaseClause*.~~
5. ~~If Result(4) is an abrupt completion then return Result(4).~~
6. ~~Go to step 13.~~
7. ~~Let *B* be the list of *CaseClause* items in the second *CaseClauses*, in source text order.~~
8. ~~For the next *CaseClause* in *B*, evaluate *CaseClause*. If there is no such *CaseClause*, go to step 15.~~
9. ~~If *input* is not equal to Result(8), as defined by the **!==** operator, go to step 8.~~
10. ~~Evaluate the *StatementList* of this *CaseClause*.~~
11. ~~If Result(10) is an abrupt completion then return Result(10)~~
12. ~~Go to step 18.~~
13. ~~For the next *CaseClause* in *A*, evaluate the *StatementList* of this *CaseClause*. If there is no such *CaseClause*, go to step 15.~~
14. ~~If Result(13) is an abrupt completion then return Result(13).~~
15. ~~Execute the *StatementList* of *DefaultClause*.~~
16. ~~If Result(15) is an abrupt completion then return Result(15)~~
17. ~~Let *B* be the list of *CaseClause* items in the second *CaseClauses*, in source text order.~~
18. ~~For the next *CaseClause* in *B*, evaluate the *StatementList* of this *CaseClause*. If there is no such *CaseClause*, return (**normal**, **empty**, **empty**).~~

19. ~~If Result(18) is an abrupt completion then return Result(18).~~

20. ~~Go to step 18.~~

The production *CaseBlock* **: {** *CaseClauses*$_{opt}$ **}** is given an input parameter, *input*, and is evaluated as follows:

1. Let *V* = **empty**.
2. Let *A* be the list of *CaseClause* items in source text order.
3. Let *C* be the next *CaseClause* in *A*. If there is no such *CaseClause*, then go to step 16.
4. Evaluate *C*.
5. If *input* is not equal to Result(4) as defined by the **!==** operator, then go to step 3.
6. If *C* does not have a *StatementList*, then go to step 10.
7. Evaluate *C*'s *StatementList* and let *R* be the result.
8. If *R* is an abrupt completion, then return *R*.
9. Let *V* = *R.*value.
10. Let *C* be the next *CaseClause* in *A*. If there is no such *CaseClause*, then go to step 16.
11. If *C* does not have a *StatementList*, then go to step 10.
12. Evaluate *C*'s *StatementList* and let *R* be the result.
13. If *R.*value is not empty, then let *V* = *R.*value.
14. If *R* is an abrupt completion, then return (*R*.type, *V*, *R*.target).
15. Go to step 10.
16. Return (**normal**, *V*, **empty**).

V0063:

The production *CaseBlock* **: {** *CaseClauses*$_{opt}$ *DefaultClause* *CaseClauses*$_{opt}$ **}** is given an input parameter, *input*, and is evaluated as follows:

1. Let *V* = **empty**.
2. Let *A* be the list of *CaseClause* items in the first *CaseClauses*, in source text order.
3. Let *C* be the next *CaseClause* in *A*. If there is no such *CaseClause*, then go to step 11.
4. Evaluate *C*.
5. If *input* is not equal to Result(4) as defined by the **!==** operator, then go to step 3.
6. If *C* does not have a *StatementList*, then go to step 20.
7. Evaluate *C*'s *StatementList* and let *R* be the result.
8. If *R* is an abrupt completion, then return *R*.
9. Let *V* = *R*.value.

10. Go to step 20.
11. Let *B* be the list of *CaseClause* items in the second *CaseClauses*, in source text order.
12. Let *C* be the next *CaseClause* in *B*. If there is no such *CaseClause*, then go to step 26.
13. Evaluate *C*.
14. If *input* is not equal to Result(13) as defined by the **!==** operator, then go to step 12.
15. If *C* does not have a *StatementList*, then go to step 31.
16. Evaluate *C*'s *StatementList* and let *R* be the result.
17. If *R* is an abrupt completion, then return *R*.
18. Let *V* = *R.*value.
19. Go to step 31.
20. Let *C* be the next *CaseClause* in *A*. If there is no such *CaseClause*, then go to step 26.
21. If *C* does not have a *StatementList*, then go to step 20.
22. Evaluate *C*'s *StatementList* and let *R* be the result.
23. If *R.*value is not empty, then let *V* = *R.*value.
24. If *R* is an abrupt completion, then return (*R*.type, *V*, *R.*target).
25. Go to step 20.
26. If the *DefaultClause* does not have a *StatementList*, then go to step 30.
27. Evaluate the *DefaultClause*'s *StatementList* and let *R* be the result.
28. If *R.*value is not empty, then let *V* = *R.*value.
29. If *R* is an abrupt completion, then return (*R*.type, *V*, *R.*target).
30. Let *B* be the list of *CaseClause* items in the second *CaseClauses*, in source text order.
31. Let *C* be the next *CaseClause* in *B*. If there is no such *CaseClause*, then go to step 37.
32. If *C* does not have a *StatementList*, then go to step 31.
33. Evaluate *C*'s *StatementList* and let *R* be the result.
34. If *R.*value is not empty, then let *V* = *R.*value.
35. If *R* is an abrupt completion, then return (*R*.type, *V*, *R.*target).
36. Go to step 31.
37. Return (**normal**, *V*, **empty**).

2.1.42 [ECMA-262-1999] Section 12.14, The try Statement

V0064:

The production *TryStatement* **: try** *Block Catch Finally* is evaluated as follows:

Step 5 below contains a specification error that is documented in the ES3 errata. JScript 5.x implements the following algorithm as corrected in the errata document.

1. Evaluate *Block*.
2. Let *C* = Result(1).
3. If Result(1).type is not **throw**, go to step 6.
4. Evaluate *Catch* with parameter Result(1).
5. ~~If Result(4).type is not **normal**,~~ Let *C* = Result(4).
6. Evaluate *Finally*.
7. If Result(6).type is **normal**, return *C*.
8. Return Result(6).

V0065:

The production *Catch* **: catch (***Identifier* **)** *Block* is evaluated as follows:

1. Let *C* be the parameter that has been passed to this production.
2. ~~Create a new object as if by the expression **new Object()**.~~
3. ~~Create a property in the object Result(2). The property's name is *Identifier*, value is *C*.value, and attributes are { DontDelete }.~~
    1. Evaluate *Identifier* as described in [ECMA-262-1999] section 11.1.2.
    2. Call PutValue(Result(t2),*C*).
4. Add Result(2) to the front of the scope chain.
5. Evaluate *Block*.
6. ~~Remove Result(2) from the front of the scope chain.~~
7. Return Result(5).

JScript 5.x does not create a new scope chain element to contain the binding of a *Catch* parameter. Instead variable instantiation (see [ECMA-262-1999] section 10.1.3) creates variables in the current variable object for all *Catch* parameters. The parameter variable of each *Catch* is initialized, as if by a Simple Assignment, to the actual parameter value before evaluating the *Catch*'s *Block*.

2.1.43 [ECMA-262-1999] Section 13, Function Definition

V0066:

Syntax

FunctionDeclaration **:**

function *Identifier* **(** *FormalParameterList*$_{opt}$ **)** **{** *FunctionBody* **}** *JScriptFunction*

FunctionExpression **:**

function *Identifier$_{opt}$* **(** *FormalParameterList$_{opt}$* **)** **{** *FunctionBody* **}** *JScriptFunction*

In JScript 5.x the *Identifier* of a *FunctionDeclaration* is optional. However, a *FunctionDeclaration* without an *Identifier* is not instantiated during variable instantiation ([ECMA-262-1999] 10.1.3) and hence has no observable effect upon the evaluation of an ECMAScript program.

Any ambiguities between the alternatives of *FunctionDeclaration* and *FunctionExpression* are resolved in favour of the first alternative rather than the *JScriptFunction* alternative.

JScriptFunction **:**

function *FunctionBindingList* **(** *FormalParameterList$_{opt}$* **)** **{** *FunctionBody* **}**

FunctionBindingList **:**

FunctionBinding
FunctionBindingList
FunctionBinding

FunctionBinding **:**

SimpleFunctionBinding
MethodBinding
EventHandlerBinding

SimpleFunctionBinding **:**

Identifier [lookahead ∉ {*NameQualifier, EventDesignator*}]

MethodBinding **:**

ObjectPath NameQualifier Identifier [lookahead ∉ {*NameQualifier, EventDesignator*}]

EventHandlerBinding **:**

ObjectPath EventDesignator Identifier

ObjectPath **:**

Identifier
ObjectPath NameQualifier Identifier

NameQualifier **: .**

EventDesignator **: ::**

FormalParameterList **:**

Identifier
FormalParameterList **,** *Identifier*

FunctionBody **:**

SourceElements

V0067:

Semantics

The productions *FunctionDeclaration* **: function** *Identifier* **(***FormalParameterList*$_{opt}$**) {** *FunctionBody* **}** and *FunctionExpression* **: function** *Identifier* **(** *FormalParameterList*$_{opt}$ **) {** *FunctionBody* **}** are ~~is~~ processed for variable instantiation ~~function declarations~~ as follows:

Step 1 below contains a specification error that is documented in the ES3 Errata. JScript 5.x implements the following algorithm as corrected in the errata document.

1. Create a new Function object as specified in [ECMA-262-1999] section 13.2 with parameters specified by *FormalParameterList*$_{opt}$, and body specified by *FunctionBody*. Pass in the scope chain of the running execution context as the *Scope*.

2. Create a property of the current variable object (as specified in [ECMA-262-1999] section 10.1.3) with name *Identifier* and value Result(1).

V0068:

The productions *FunctionDeclaration* **:** *JScriptFunction* and *FunctionExpression* **:** *JScriptFunction* are processed for variable instantiation as follows:

1. Create a new Function object as specified in [ECMA-262-1999] section 13.2 with parameters specified by the *FormalParameterList*$_{opt}$ element of the *JScriptFunction* and body specified by the *FunctionBody* element. Pass in the scope chain of the running execution context as the *Scope*.

2. Return the value Result(1).

A *FunctionDeclaration* may be evaluated as a *Statement*. When either the production *FunctionDeclaration* **: function** *Identifier* **(** *FormalParameterList*$_{opt}$ **) {** *FunctionBody* **}** or the production *FunctionDeclaration* **:** *JScriptFunction* is evaluated the following step is performed:

1. Return (**normal**, **empty**, **empty**).

V0069:

The production *FunctionExpression* **: function (** *FormalParameterList*$_{opt}$ **) {** *FunctionBody* **}** is evaluated as follows:

Step 2 below contains a specification error that is documented in the ES3 Errata. JScript 5.x implements the following algorithm as corrected in the errata document.

1. Create a new Function object as specified in [ECMA-262-1999] section 13.2 with parameters specified by *FormalParameterList*$_{opt}$ and body specified by *FunctionBody*. Pass in the scope chain of the running execution context as the *Scope*.

2. Return Result(~~2~~1).

V0070:

The productions *FunctionExpression* **: function** *Identifier* **(** *FormalParameterList*$_{opt}$ **) {** *FunctionBody* **}** and the production *FunctionExpression* **:** *JScriptFunction* are ~~is~~ evaluated as follows:

1. ~~Create a new object as if by the expression **new Object()**.~~

2. ~~Add Result(1) to the front of the scope chain.~~

3. Create a new Function object as specified in [ECMA-262-1999] section 13.2 with parameters specified by *FormalParameterList*$_{opt}$ and body specified by *FunctionBody*. Pass in the scope chain of the running execution context as the *Scope*.

4. ~~Create a property in the object Result(1). The property's name is *Identifier*, value is Result(3), and attributes are { DontDelete, ReadOnly }.~~

5. ~~Remove Result(1) from the front of the scope chain.~~

6. Return Result(3).

V0071:

NOTE

The Identifier in a FunctionExpression can be referenced from inside the FunctionExpression's FunctionBody to allow the function to call itself recursively. ~~*However, unlike in a FunctionDeclaration, the Identifier in a FunctionExpression cannot be referenced from and does not affect the scope enclosing the FunctionExpression.*~~

In JScript 5.x *FunctionExpressions* are processed during variable instantiation ([ECMA-262-1999] section 10.1.3) and their names may affect the enclosing scope.

When evaluating a *FunctionExpression*, JScript 5.x does not create the local scope object in step 1 above and does not create a local binding for the *Identifier* of the *FunctionExpression* (step 4). This means that a reference to the *Identifier* from within the *FunctionBody* of such a *FunctionExpression* may evaluate to a different value from the evaluated function object.

V0072:

The production *FunctionBindingList* **:** *FunctionBindingList***,** *FunctionBinding* is processed during variable instantiation as follows when passed a function object *func* and the attributes *attrs* as arguments:

1. Process the *FunctionBindingList* passing *func* and *attrs* as the arguments.

2. Process the *FunctionBinding* passing *func* and *attrs* as the arguments.

V0073:

The production *FunctionBindingList* **:** *FunctionBindingList***,** *FunctionBinding* is processed during variable instantiation as follows when passed a function object *func* and the attributes *attrs* as arguments:

1. Process the *FunctionBinding* passing *func* and *attrs* as the arguments.

V0074:

The production *FunctionBinding* **:** *SimpleFunctionBinding* is processed during variable instantiation as follows when passed a function object *func* and the attributes *attrs* as arguments:

1. Let *id* be the *Identifier* element of the *SimpleFunctionBinding.*

2. If *id* is **arguments**, return.

3. Create a property of the variable object of the running execution context with a name if *id*, a value of *func*, and with the attributes that are contained in *attrs*. If the variable object already has a property with this name, replace its value and attributes.

V0075:

The production *FunctionBinding* **:** *MethodBinding* is processed during variable instantiation as follows when passed a function object *func* and the attributes *attrs* as arguments:

1. Let *objRef* be the result of evaluating the *ObjectPath* element of *MethodBinding.*
2. Call PutValue(*objRef*,*func*).
3. Return.

V0076:

The production *FunctionBinding* **:** *EventHandlerBinding* is processed during variable instantiation as follows when passed a function object *func* and the attributes *attrs* as arguments:

1. Let *objRef* be the result of evaluating the *ObjectPath* element of *MethodBinding.*
2. Call GetValue(Result(1)).
3. Call ToObject(Result(2)).
4. If Result(3) is not a host object that supports event attachment return.
5. Let *eventName* be a string containing the text of the *Identifier* element of *EventHandlerBinding*.
6. Perform the host specific action that will associate *func* as an event handler on Result(3) for the event named *eventName*. This action may throw exceptions.
7. Return.

If the host is Internet Explorer, step 6 above is equivalent to invoking the *attachEvent* method of Result(3) passing *eventName* and *func* as the arguments.

V0077:

The production *ObjectPath* **:** *Identifier* is evaluated identically to the manner that the production *PrimaryExpression* **:** *Identifier* would be evaluated for the same Identifier (see [ECMA-262-1999] section10.1.4).

The production *ObjectPath* **:** *ObjectPath NameQualifier Identifier* is evaluated as follows:

1. Evaluate *ObjectPath*.
2. Call GetValue(Result(1)).
3. Call ToObject(Result(2)).
4. Return a value of type Reference that has a base object of Result(3) and a property name of *Identifier*.

2.1.44 [ECMA-262-1999] Section 13.2, Creating Function Objects

V0078:

Given an optional parameter list specified by *FormalParameterList*, a body specified by *FunctionBody*, and a scope chain specified by *Scope*, a Function object is constructed as follows:

1. If there already exists an object *E* that was created by an earlier call to this section's algorithm, and if that call to this section's algorithm was given a *FunctionBody* that is equated to the *FunctionBody* given now, then go to step 13. (If there is more than one object *E* satisfying these criteria, choose one at the implementation's discretion.)
2. Create a new native ECMAScript object and let *F* be that object.
3. Set the **[[Class]]** property of *F* to **"Function"**.
4. Set the **[[Prototype]]** property of *F* to the original Function prototype object as specified in [ECMA-262-1999] section 15.3.3.1.
5. Set the **[[Call]]** property of *F* as described in [ECMA-262-1999] section 13.2.1.
6. Set the **[[Construct]]** property of *F* as described in [ECMA-262-1999] section 13.2.2.
7. Set the **[[Scope]]** property of *F* to a new scope chain ([ECMA-262-1999] section 10.1.4) that contains the same objects as *Scope*.
8. Set the **length** property of *F* to the number of formal properties specified in *FormalParameterList*. If no parameters are specified, set the **length** property of *F* to 0. This property is given attributes as specified in [ECMA-262-1999] section 15.3.5.1.
9. Create a new object as would be constructed by the expression **new Object()**.
10. Set the **constructor** property of Result(9) to *F*. This property is given attributes { DontEnum }.
11. Set the **prototype** property of *F* to Result(9). This property is given attributes as specified in [ECMA-262-1999] section 15.3.5.2.
12. Return *F*.
13. At the implementation's discretion, go to either step 2 or step 14.
14. Create a new native ECMAScript object joined to *E* and let *F* be that object. Copy all non-internal properties and their attributes from *E* to *F* so that all non-internal properties are identical in *E* and *F*.
15. Set the **[[Class]]** property of *F* to **"Function"**.
16. Set the **[[Prototype]]** property of *F* to the original Function prototype object as specified in [ECMA-262-1999] section 15.3.3.1.
17. Set the **[[Call]]** property of *F* as described in [ECMA-262-1999] section 13.2.1.
18. Set the **[[Construct]]** property of *F* as described in [ECMA-262-1999] section 13.2.2.
19. Set the **[[Scope]]** property of *F* to a new scope chain ([ECMA-262-1999] section 10.1.4) that contains the same objects as *Scope*.
20. Return *F*.

JScript 5.x never joins function objects. Step 13 of the above algorithm always goes to step 2.

2.1.45 [ECMA-262-1999] Section 13.2.2, [[Construct]]

V0079:

When the **[[Construct]]** property for a Function object *F* is called, the following steps are taken:

1. Create a new native ECMAScript object.
2. Set the **[[Class]]** property of Result(1) to **"Object"**.
3. Get the value of the **prototype** property of the *F*.
4. If Result(3) is ~~an~~ native object, set the **[[Prototype]]** property of Result(1) to Result(3).
5. If Result(3) is a host object or is not an object, set the **[[Prototype]]** property of Result(1) to the original Object prototype object as described in [ECMA-262-1999] 15.2.3.1.
6. Invoke the **[[Call]]** property of *F*, providing Result(1) as the **this** value and providing the argument list passed into **[[Construct]]** as the argument values.
7. If Type(Result(6)) is Object then return Result(6).
8. Return Result(1).

2.1.46 [ECMA-262-1999] Section 15, Native ECMAScript Objects

V0080:

Unless otherwise specified in the description of a particular function, if a function or constructor described in this section is given more arguments than the function is specified to allow, the behaviour of the function or constructor is undefined. In particular, an implementation is permitted (but not required) to throw a **TypeError** exception in this case.

JScript does not throw an exception when a built-in function is called with extra arguments. The extra arguments are ignored by the function, which otherwise behaves as specified in this section.

2.1.47 [ECMA-262-1999] Section 15.1, The Global Object

V0081:

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

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

The values of the **[[Prototype]]** and **[[Class]]** properties of the global object are implementation-dependent.

In JScript 5.x the global object is a host object rather than a native object. The **[[Class]]** property of the global object has the value **Object**. The JScript 5.x global object does not actually have a **[[Prototype]]** property but for all situations described in this specification it behaves as if it had a **[[Prototype]]** property whose value was

null. The global object does not inherit any properties from the built-in **Object.prototype** object.

2.1.48 [ECMA-262-1999] Section 15.1.2.1, eval(x)

V0082:

If value of the **eval** property is used in any way other than a direct call (that is, other than by the explicit use of its name as an *Identifier* which is the *MemberExpression* in a *CallExpression*), or if the **eval** property is assigned to, an **EvalError** exception may be thrown.

JScript 5.x does not restrict usage of the function that is the initial value of the **eval** property or restrict assignment to the **eval** property. It does not throw **EvalError** in the situations.

2.1.49 [ECMA-262-1999] Section 15.1.2.2, parseInt (string, radix)

V0083:

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

1. Call ToString(*string*).
2. Let *S* be a newly created substring of Result(1) consisting of the first character that is not a *StrWhiteSpaceChar* and all characters following that character. (In other words, remove leading white space.)
3. Let *sign* be 1.
4. If *S* is not empty and the first character of *S* is a minus sign **-**, let *sign* be -1.
5. If *S* is not empty and the first character of *S* is a plus sign **+** or a minus sign **-**, then remove the first character from *S*.
6. Let *R* = ToInt32(*radix*).
7. If *R* = 0, go to step 11.
8. If *R* < 2 or *R* > 36, then return **NaN**.
9. If *R* = 16, go to step 13.
10. Go to step 14.
11. Let *R* = 10.
12. If the length of *S* is at least 1 and the first character of *S* is "**0**", then at the implementation's discretion either let *R* = 8 or leave *R* unchanged. JScript 5.x always sets *R* = 8 in this situation.
13. If the length of *S* is at least 2 and the first two characters of *S* are either "**0x**" or "**0X**", then remove the first two characters from *S* and let *R* = 16.
14. If *S* contains any character that is not a radix-*R* digit, then let *Z* be the substring of *S* consisting of all characters before the first such character; otherwise, let *Z* be *S*.
15. If *Z* is empty, return **NaN**.
16. Compute the mathematical integer value that is represented by *Z* in radix-*R* notation, using the letters **A**-**Z** and **a**-**z** for digits with values 10 through 35. (However, if *R* is 10 and *Z* contains more

than 20 significant digits, every significant digit after the 20th may be replaced by a **0** digit, at the option of the implementation (JScript 5.x replaces digits after the 20th by a **0**); and if *R* is not 2, 4, 8, 10, 16, or 32, then Result(16) may be an implementation-dependent approximation to the mathematical integer value that is represented by *Z* in radix-*R* notation. No approximations are used for values of R.)

17. Compute the number value for Result(16).

18. Return *sign* × Result(17).

2.1.50 [ECMA-262-1999] Section 15.2.1.1, Object ([value])

V0084:

When the **Object** function is called with no arguments or with one argument *value*, the following steps are taken:

1. If *value* is **null**, **undefined** or not supplied, or if Type(*value*) is VarDate, create and return a new Object object exactly as if the object constructor had been called with the same arguments ([ECMA-262-1999] section 15.2.2.1).

2. Return ToObject(*value*).

2.1.51 [ECMA-262-1999] Section 15.2.2.1, new Object ([value])

V0085:

When the **Object** constructor is called with no arguments or with one argument *value*, the following steps are taken:

1. If *value* is not supplied, go to step 8.

2. If the type of *value* is not Object, go to step 5.

3. If the *value* is a native ECMAScript object, do not create a new object but simply return *value*.

4. If the *value* is a host object, then actions are taken and a result is returned in an implementation-dependent manner that may depend on the host object.

   JScript 5.x simply returns *value* if it is a host object.

5. If the type of *value* is String, return ToObject(*value*).

6. If the type of *value* is Boolean, return ToObject(*value*).

7. If the type of *value* is Number, return ToObject(*value*).

8. (The argument *value* was not supplied or its type was SafeArray, VarDate, Null, or Undefined.)

   Create a new native ECMAScript object.

   The **[[Prototype]]** property of the newly constructed object is set to the Object prototype object.

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

   The newly constructed object has no **[[Value]]** property.

   Return the newly created native object.

2.1.52 [ECMA-262-1999] Section 15.2.3, Properties of the Object Constructor

V0086:

The value of the internal **[[Prototype]]** property of the Object constructor is the Function prototype object.

Besides the internal properties and the **length** property (whose value is ~~**1**~~ **0**), the Object constructor has the following properties:

2.1.53 [ECMA-262-1999] Section 15.2.4.2, Object.prototype.toString ()

V0087:

When the **toString** method is called, the following steps are taken:

(The bulleted steps are added before step 1)

- (0.1) If the Type of the **this** value is VarDate, return "[object Object]".
- (0.2) If the **this** value is **null** or **undefined** use the global object, otherwise use the result of calling ToObject with the **this** value as the argument.
- (0.3) If the value in 0.2 is a host object, return "[object Object]".

1. Get the **[[Class]]** property of the value in 0.2 ~~this object~~.
2. Compute a string value by concatenating the three strings **"[object "**, Result(1), and **"]"**.
3. Return Result(2).

2.1.54 [ECMA-262-1999] Section 15.2.4.3, Object.prototoype.toLocaleString ()

V0088:

This function returns the result of ~~calling **toString()**.~~ the following steps:

1. If the **this** object is a host object, return **"[object]"**.
2. If the Type of the **this** value is a VarDate, throw a **TypeError** exception.
3. Call ToObject with the **this** value as the argument.
4. If *toStr* does not have a **[[Call]]** method, throw a **TypeError** exception.
5. Return the result of calling the **[[Call]]** method of *toStr* passing Result(3) as the **this** value and with no other arguments.

2.1.55 [ECMA-262-1999] Section 15.2.4.4, Object.prototype.valueOf ()

V0089:

The **valueOf** method returns its **this** value. If the object is the result of calling the Object constructor with a host object ([ECMA-262-1999] section 15.2.2.1), it is implementation-defined whether **valueOf** returns its **this** value or another value such as the host object originally passed to the constructor.

JScript 5.x returns its **this** value without applying any coercions to it. If this method is called using the JScript 5.x implementations of the **Function.prototype.call** or

Function.prototype.apply methods and passing either **null** or **undefined** as the **this** argument, the result is not the global object. If a primitive Number, String, or Boolean value is passed as the **this** argument the result is the passed primitive value rather than a corresponding wrapper object.

2.1.56 [ECMA-262-1999] Section 15.2.4.5, Object.prototype.hasOwnProperty (V)

V0090:

When the **hasOwnProperty** method is called with argument *V*, the following steps are taken:

1. Let *O* be ~~this object~~ the result of calling ToObject passing the **this** value as the argument.
2. Call ToString(*V*).
3. If *O* doesn't have a property with the name given by Result(2), return **false**.
4. Return **true**.

NOTE

*Unlike **[[HasProperty]]** ([ECMA-262-1999] section 8.6.2.4), this method does not consider objects in the prototype chain.*

2.1.57 [ECMA-262-1999] Section 15.2.4.6, Object.prototype.isPrototypeOf (V)

V0091:

When the **isPrototypeOf** method is called with argument *V*, the following steps are taken:

1. Let *O* be ~~this object~~ the result of calling ToObject passing the **this** value as the argument.
2. If *V* is not an object, return **false**.
    1. If *O* and *V* refer to the same object, return **true**.
3. Let *V* be the value of the **[[Prototype]]** property of *V*.
4. if *V* is **null**, return **false**
5. If *O* and *V* refer to the same object or if they refer to objects joined to each other ([ECMA-262-1999] section 13.1.2), return **true**.
6. Go to step 3.

In JScript 5.x, the **isPrototypeOf** method returns **true** rather than **false** if the **this** value and the argument are the same object.

2.1.58 [ECMA-262-1999] Section 15.2.4.7, Object.prototype.propertyIsEnumerable (V)

V0092:

When the **propertyIsEnumerable** method is called with argument *V*, the following steps are taken:

1. Let *O* be ~~this object~~ the result of calling ToObject passing the **this** value as the argument.

    1. If *O* is a host object, throw a **TypeError** exception.

2. Call ToString(*V*).
3. If *O* doesn't have a property with the name given by Result(2), return **false**.
4. If the property has the DontEnum attribute, return **false**.
5. Return **true**.

NOTE

This method does not consider objects in the prototype chain.

2.1.59 [ECMA-262-1999] Section 15.3.4. Properties of the Function Prototype Object

V0093:

The Function prototype object is itself a Function object (its **[[Class]]** is **"Function"**) that, when invoked, accepts any arguments and returns **undefined**.

The value of the internal **[[Prototype]]** property of the Function prototype object is the Object prototype object ([ECMA-262-1999] section 15.3.2.1).

It is a function with an "empty body"; if it is invoked, it merely returns **undefined**.

The Function prototype object does not have a **valueOf** property of its own; however, it inherits the **valueOf** property from the Object prototype Object.

From the above specification language it is unclear whether or not the Function prototype object has all the properties of a Function instance as described in [ECMA-262-1999] section 15.3.5. In addition, [ECMA-262-1999] section 15 states: "None of the built-in functions described in this section shall implement the internal **[[Construct]]** method unless otherwise specified in the description of a particular function. None of the built-in functions described in this section shall initially have a **prototype** property unless otherwise specified in the description of a particular function." The Function prototype object is itself such a built-in function.

In JScript 5.x the Function prototype object does not have a **prototype** property. It also does not have **[[Construct]]** or **[[HasInstance]]** properties. Because of the lack of these properties applying the **new** operator to the Function prototype object or using the Function prototype object as the right hand operand of the **instanceof** operator throws a **TypeError** exception.

2.1.60 [ECMA-262-1999] Section 15.3.4.2, Function.prototype.toString ()

V0094:

An implementation-dependent representation of the function is returned. This representation has the syntax of a ~~*FunctionDeclaration*~~ *FunctionExpression*. Note in particular that the use and placement of white space, line terminators, and semicolons within the representation string is implementation-dependent.

The representation of a function implemented using ECMAScript code is the exact sequence of characters used to define the function. The first character of the representation is the **'f'** of function, and the final character is the final **'}'** of the function definition. However, if the function was defined using a FunctionExpression which is immediately surrounded by one or more levels of grouping operators ([ECMA-262-1999] section 11.1.6) then the first character of the representation is the **'('** of the outermost such grouping operator and the final character is the **')'** of the outermost such grouping operator.

If the function was created by the Function constructor ([ECMA-262-1999] section 15.3.2.1) the representation of the function consists of the string **'function anonymous('**, immediately followed by the value of *P* used in step 16 of the [ECMA-262-1999] section 15.3.2.1 algorithm that created the function, immediately followed by the string **') {'**, immediately followed by a <LF> character, immediate followed by the value of *body* used in step 16 of the algorithm, immediately followed by a <LF> character and the character **'}'.**

If the function is not implemented using ECMAScript code (it is a built-in function or a host object function), the *FunctionBody* of the generated representation does not conform to ECMAScript syntax. Instead, the *FunctionBody* consists of the text **"[native code]"**.

The format of the representation generated by JScript 5.x is most appropriately described as having the syntax of a standard ECMAScript, Third Edition *FunctionExpression* rather than a *FunctionDeclaration*. This is because in the case of anonymous functions created via a *FunctionExpression* that does not include the optional *Identifier*) the generated syntax does not include the optional *Identifier* and hence does not conform to the base standard's definition of *FunctionExpression*.

2.1.61 [ECMA-262-1999] Section 15.3.4.3, Function.prototype.apply (thisArg, argArray)

V0095:

The **apply** method takes two arguments, *thisArg* and *argArray*, and performs a function call using the **[[Call]]** property of the object. If the object does not have a **[[Call]]** property, a **TypeError** exception is thrown.

~~If *thisArg* is **null** or **undefined**, the called function is passed the global object as the **this** value. Otherwise, the called function is passed ToObject(*thisArg*) as the **this** value.~~ If no arguments are present, the global object is used as the value of *thisArg*.

2.1.62 [ECMA-262-1999] Section 15.3.4.4, Function.prototype.call (thisArg [, arg1 [, arg2, …]])

V0096:

The **call** method takes one or more arguments, *thisArg* and (optionally) *arg1*, *arg2* etc, and performs a function call using the **[[Call]]** property of the object. If the object does not have a **[[Call]]** property, a **TypeError** exception is thrown. The called function is passed *arg1*, *arg2*, etc. as the arguments.

~~If *thisArg* is **null** or **undefined**, the called function is passed the global object as the **this** value. Otherwise, the called function is passed ToObject(*thisArg*) as the **this** value.~~ If no arguments are present, the global object is used as the value of *thisArg*.

The **length** property of the **call** method is **1**.

2.1.63 [ECMA-262-1999] Section 15.3.5.2, prototype

V0097:

The value of the **prototype** property is used to initialise the internal **[[Prototype]]** property of a newly created object before the Function object is invoked as a constructor for that newly created object. This property has the attribute { ~~DontDelete~~ DontEnum }.

2.1.64 [ECMA-262-1999] Section 15.4.2.1, new Array ([item0 [, item1 [, …]]])

V0098:

The **0** property of the newly constructed object is set to *item0* (if supplied); the **1** property of the newly constructed object is set to *item1* (if supplied); and, in general, for as many arguments as there are, the *k* property of the newly constructed object is set to argument *k*, where the first argument is considered to be argument number **0**. If the value of an argument item is **undefined**, an own property of the newly constructed object corresponding to that argument is not created.

2.1.65 [ECMA-262-1999] Section 15.4.4.3, Array.prototype.toLocaleString ()

V0099:

The elements of the array are converted to strings using their **toLocaleString** methods, and these strings are then concatenated, separated by occurrences of a separator string that has been derived in an implementation-defined locale-specific way. The result of calling this function is intended to be analogous to the result of **toString**, except that the result of this function is intended to be locale-specific.

The result is calculated as follows:

1. Call the **[[Get]]** method of this object with argument **"length"**.
2. Call ToUint32(Result(1)).
3. Let *separator* be the list-separator string appropriate for the host environment's current locale (this is derived in an implementation-defined way).
4. Call ToString(*separator*).
5. If Result(2) is zero, return the empty string.
6. Call the **[[Get]]** method of this object with argument **"0"**.
    1. If Type(Result(6)) is VarDate, let *R* be **"[object Object]"** and then go to step 9.
7. If Result(6) is **undefined** or **null**, use the empty string; otherwise, call ToObject(Result(6)).toLocaleString(). If the recursive call to **toLocaleString** would cause a non-terminating recursion use the empty string as the result of this step.
8. Let *R* be Result(7).
9. Let *k* be **1**.
10. If *k* equals Result(2), return *R*.
11. Let *S* be a string value produced by concatenating *R* and Result(4).
12. Call the **[[Get]]** method of this object with argument ToString(*k*).

    1. If Type(Result(12)) is VarDate, use **"[object Object]"** as Result(13) and then go to step 14.

13. If Result(12) is **undefined** or **null**, use the empty string; otherwise, call ToObject(Result(12)).toLocaleString(). If the recursive call to **toLocaleString** would cause a non-terminating recursion use the empty string as the result of this step.

14. Let *R* be a string value produced by concatenating *S* and Result(13).

15. Increase *k* by 1.

16. Go to step 10.

The **toLocaleString** function is not generic; it throws a **TypeError** exception if its **this** value is not an Array object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

> JScript 5.x determines the *separator* in step 3 by using the Windows **GetLocaleInfo** system function and requesting the LOCALE_SLIST value for the current user locale.

2.1.66 [ECMA-262-1999] Section 15.4.4.4, Array.prototype.concat ([item1 [, item2 [, …]]])

When the **concat** method is called with zero or more arguments *item1*, *item2*, etc., it returns an array containing the array elements of the object followed by the array elements of each argument in order.

The following steps are taken:

1. Let *A* be a new array created as if by the expression **new Array()**.
2. Let *n* be 0.
    1. Let *biasN* = 0.
3. Let *E* be this object.
4. If *E* is not an Array object, go to step 16.
5. Let *k* be 0.
    1. Let *biasK* = 0.
6. Call the **[[Get]]** method of *E* with argument **"length"**.
7. If *k* equals Result(6) go to step 19.
8. Call ToString(*k*-*biasK*).
9. If *E* has a property named by Result(8), go to step 10, but if *E* has no property named by Result(8), go to step 13.
10. Call ToString(n-*biasN*).
11. Call the **[[Get]]** method of *E* with argument Result(8).
12. Call the **[[Put]]** method of *A* with arguments Result(10) and Result(11).
13. Increase *n* by 1.
    1. If *n* > 2147483647, then let *biasN* = 4294967296; else let *biasN* = 0.
14. Increase *k* by 1.

    1. If *k* > 2147483647, then let *biasK* = 4294967296; else let *biasK* = 0.

15. Go to step 7.

16. Call ToString(*n*-*biasN*).

17. Call the **[[Put]]** method of *A* with arguments Result(16) and *E*.

18. Increase *n* by 1.

    1. If *n* > 2147483647, then let *biasN* = 4294967296; else let *biasN* = 0.

19. Get the next argument in the argument list; if there are no more arguments, go to step 22.

20. Let *E* be Result(19).

21. Go to step 4.

22. Call the **[[Put]]** method of *A* with arguments **"length"** and *n*.

23. Return *A*.

The **length** property of the **concat** method is **1**.

As specified above in step 3, JScript 5.x uses the passed **this** value without applying a ToObject coercion to it. If this method is called using the JScript 5.x implementations of the **Function.prototype.call** or **Function.prototype.apply** methods and passing either **null** or **undefined** as the **this** argument, the global object is not used as the **this** value. If a Number, String, or Boolean value is passed as the **this** argument the passed value rather than a corresponding wrapper object is used as the **this** value.

V0100:

NOTE

*The **concat** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **concat** function can be applied successfully to a host object is implementation-dependent.*

In JScript 5.x the **concat** function handles array index property names with numeric values greater than 2^{31}-1 differently from numerically smaller array index property names. As this behavior differs from the base specification and from the probable user intent the use of this function on objects containing such properties should be avoided.

2.1.67 [ECMA-262-1999] Section 15.4.4.5, Array.prototype.join (separator)

V0101:

The elements of the array are converted to strings, and these strings are then concatenated, separated by occurrences of the *separator*. If no separator is provided, a single comma is used as the separator.

In JScript 5.7, if the value **undefined** is explicitly provided as the separator argument, the string **"undefined"** is used as the separator.

The **join** method takes one argument, *separator*, and performs the following steps:

(The bulleted steps are added before step 1)

- Let *O* be the result of calling ToObject with the **this** value as the argument.
- If *O* is a host object, throw a **TypeError** exception.

1. Call the **[[Get]]** method of *O* ~~this object~~ with argument **"length"**.
2. Call ToUint32(Result(1)).
    1. If JScript 5.7 and the *separator* argument is not present, let *separator* be the single character **","**.
    2. If JScript 5.7, go to step 4.
3. If *separator* is **undefined**, let *separator* be the single-character string **","**.
4. Call ToString(*separator*).
5. If Result(2) is zero, return the empty string.
6. Call the **[[Get]]** method of *O* ~~this object~~ with argument **"0"**.
7. If Result(6) is **undefined** or **null**, use the empty string; otherwise, call ToString(Result(6)). If the call to ToString would cause a non-terminating recursion use the empty string as the result of this step.
8. Let *R* be Result(7).
9. Let *k* be **1**.
10. If *k* equals Result(2), return *R*.
11. Let *S* be a string value produced by concatenating *R* and Result(4).
12. Call the **[[Get]]** method of *O* ~~this object~~ with argument ToString(*k*).
13. If Result(12) is **undefined** or **null**, use the empty string; otherwise, call ToString(Result(12)). If the call to ToString would cause a nonterminating recursion use the empty string as the result of this step.
14. Let *R* be a string value produced by concatenating *S* and Result(13).
15. Increase *k* by 1.
16. Go to step 10.

The **length** property of the **join** method is **1**.

V0102:

NOTE

*The **join** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method. Whether the **join** function can be applied successfully to a host object is implementation-dependent. JScript 5.x does not allow the **join** function to be applied to a host object.*

2.1.68 [ECMA-262-1999] Section 15.4.4.6, Array.prototype.pop ()

V0103:

The last element of the array is removed from the array and returned.

(The bulleted steps are added before step 1)

- Let *O* be the result of calling ToObject with the **this** value as the argument.
- If *O* is a host object, throw a **TypeError** exception.

1. Call the **[[Get]]** method of *O* ~~this object~~ with argument **"length"**.
2. Call ToUint32(Result(1)).
3. If Result(2) is not zero, go to step 6.
4. Call the **[[Put]]** method of *O* ~~this object~~ with arguments **"length"** and Result(2).
5. Return **undefined**.
6. Call ToString(Result(2)-1).
7. Call the **[[Get]]** method of *O* ~~this object~~ with argument Result(6).
8. Call the **[[Delete]]** method of *O* ~~this object~~ with argument Result(6).
9. Call the **[[Put]]** method of *O* ~~this object~~ with arguments **"length"** and (Result(2)-1).
    1. If JScript 5.x under Internet Explorer 7 or 8 and Result(2) > 2147483648, return **undefined**.
10. Return Result(7).

V0104:

NOTE

*The **pop** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. ~~Whether the **pop** function can be applied successfully to a host object is implementation-dependent.~~ JScript 5.x does not allow the **pop** function to be applied to a host object.*

2.1.69 [ECMA-262-1999] Section 15.4.4.7, Array.prototype.push ([item1 [, item2 [, …]]])

V0105:

The arguments are appended to the end of the array, in the order in which they appear. The new length of the array is returned as the result of the call.

When the **push** method is called with zero or more arguments *item1, item2*, etc., the following steps are taken:

(The bulleted steps are added before step 1)

- Let *O* be the result of calling ToObject with the **this** value as the argument.
- If *O* is a host object, throw a **TypeError** exception.
- If JScript 5.7 and if **false** is the result of calling the **[[HasProperty]]** method of *O* with name "**length**", return **undefined**.

1. Call the **[[Get]]** method of *O* ~~this object~~ with argument **"length"**.
2. Let *n* be the result of calling ToUint32(Result(1)).

3. Get the next argument in the argument list; if there are no more arguments, go to step 7.
    1. If JScript.X under Internet Explorer 9 and *n* < 2147483648 then let *indx* be *n*; else throw a **RangeError** exception.
    2. If JScript.X under Internet Explorer 7 or 8 and *n* < 2147483648 then let *indx* be *n;* else let *indx* be *n*-4294967296.
4. Call the **[[Put]]** method of *O* ~~this object~~ with arguments ToString(*indx* ~~*n*~~) and Result(3).
5. Increase *n* by 1.
6. Go to step 3.
7. Call the **[[Put]]** method of *O* ~~this object~~ with arguments **"length"** and *n*.
8. If JScript.X under Internet Explorer 9, return ~~Return~~ n.
9. If JScript.X under Internet Explorer 7 or 8 and *n* < 2147483648 return *n*; else return *n*-4294967296.

JScript 5.x under Internet Explorer 7 or 8 does not conform to the base specification in situations where the initial value of the array's **length** property after conversion using ToUint32 is greater than 2147483647 (which is 2^{31}-1) or where the push method's base specification operation would cause the array's length to exceed that value. In such situations, any array elements that would have been created with indices greater than 2147483647 are instead created with properties names that are the string representation of the negative integer that is the 32-bit 2's complement interpretation of 32-bit encoding of the index value. The **length** property is adjusted normally in conformance to the base specification; however, if the final length value is greater than 2147483647, the return value is the negative integer that is the 32-bit 2's complement interpretation of 32-bit encoding of the final length value.

2.1.70 [ECMA-262-1999] Section 15.4.4.8, Array.prototype.reverse ()

V0106:

The elements of the array are arranged so as to reverse their order. The object is returned as the result of the call.

(The bulleted steps are added before step 1)

- Let *O* be the result of calling ToObject with the **this** value as the argument.
- If *O* is a host object, throw a **TypeError** exception.

1. Call the **[[Get]]** method of *O* ~~this object~~ with argument **"length"**.
2. Call ToUint32(Result(1)).
3. Compute floor(Result(2)/2).
4. Let *k* be 0.
5. If *k* equals Result(3), return *O* ~~this object~~.
6. Compute Result(2)-*k*-1.
    1. If k > 2147483647, then let *biasLower* = 4294967296; else let *biasLower* = 0.

2. If Result(6) > 2147483647, then let *biasUpper* = 4294967296; else let *biasUpper* = 0.

7. Call ToString(*k*-biasLower).

8. Call ToString(Result(6)-*biasUpper*).

9. Call the **[[Get]]** method of *O* ~~this object~~ with argument Result(7).

10. Call the **[[Get]]** method of *O* ~~this object~~ with argument Result(8).

11. If this object does not have a property named by Result(8), go to step 19.

12. If this object does not have a property named by Result(7), go to step 16.

13. Call the **[[Put]]** method of *O* ~~this object~~ with arguments Result(7) and Result(10).

14. Call the **[[Put]]** method of *O* ~~this object~~ with arguments Result(8) and Result(9).

15. Go to step 25.

16. Call the **[[Put]]** method of *O* ~~this object~~ with arguments Result(7) and Result(10).

17. Call the **[[Delete]]** method on *O* ~~this object~~, providing Result(8) as the name of the property to delete.

18. Go to step 25.

19. If this object does not have a property named by Result(7), go to step 23.

20. Call the **[[Delete]]** method on *O* ~~this object~~, providing Result(7) as the name of the property to delete.

21. Call the **[[Put]]** method of *O* ~~this object~~ with arguments Result(8) and Result(9).

22. Go to step 25.

23. Call the **[[Delete]]** method on *O* ~~this object~~, providing Result(7) as the name of the property to delete.

24. Call the **[[Delete]]** method on *O* ~~this object~~, providing Result(8) as the name of the property to delete.

25. Increase *k* by 1.

26. Go to step 5.

V0107:

NOTE

*The **reverse** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method. Whether the **reverse** function can be applied successfully to a host object is implementation-dependent. JScript 5.x does not allow the **reverse** function to be applied to a host object.*

In JScript 5.x the **reverse** function handles array index property names with numeric values greater than 2^{31}-1 differently from numerically smaller array index property names. As this behavior differs from the base specification and from the probable user intent the use of this function on objects containing such properties should be avoided.

2.1.71 [ECMA-262-1999] Section 15.4.4.9, Array.prototype.shift ()

V0108:

The first element of the array is removed from the array and returned.

(The bulleted steps are added before step 1)

- Let *O* be the result of calling ToObject with the **this** value as the argument.
- If *O* is a host object, throw a **TypeError** exception.
- If JScript 5.7 and if **false** is the result of calling the **[[HasProperty]]** method of *O* with name **"length"**, return **undefined**.

1. Call the **[[Get]]** method of *O* ~~this object~~ with argument "**length**".
2. Call ToUint32(Result(1)).
3. If Result(2) is not zero, go to step 6.
4. Call the **[[Put]]** method of *O* ~~this object~~ with arguments "**length**" and Result(2).
5. Return **undefined**.
6. Call the **[[Get]]** method of *O* ~~this object~~ with argument **0**.
7. Let *k* be 1.
8. If *k* equals Result(2), go to step 18.
    1. If *k* > 2147483647, then let *biasSrc* = 4294967296; else let *biasSrc* = 0.
    2. If *k*-1 > 2147483647, then let *biasDst* = 4294967296; else let *biasDst* = 0.
9. Call ToString(*k*-*biasSrc*).
10. Call ToString(*k*-1-*biasDst*).
11. If *O* ~~this object~~ has a property named by Result(9), go to step 12; but if *O* ~~this object~~ has no property named by Result(9), then go to step 15.
12. Call the **[[Get]]** method of *O* ~~this object~~ with argument Result(9).
13. Call the **[[Put]]** method of *O* ~~this object~~ with arguments Result(10) and Result(12).
14. Go to step 16.
15. Call the **[[Delete]]** method of *O* ~~this object~~ with argument Result(10).
16. Increase *k* by 1.
17. Go to step 8.
18. If JScript 5.8 call the **[[Delete]]** method of *O* ~~this object~~ with argument ToString(Result(2)-1).
19. Call the **[[Put]]** method of *O* ~~this object~~ with arguments "**length**" and (Result(2)-1).
20. Return Result(6).

V0109:

NOTE

*The **shift** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **shift** function can be applied successfully to a host object is implementation-dependent. Jscript 5.x does not allow the **shift** function to be applied to a host object.*

In Jscript 5.x the **shift** function handles array index property names with numeric values greater than 2^{31}-1 differently from numerically smaller array index property names. As this behavior differs from the base specification and from probable user intent, the use of this function on objects containing such properties should be avoided.

2.1.72 [ECMA-262-1999] Section 15.4.4.10, Array.prototype.slice (start, end)

V0110:

The **slice** method takes two arguments, *start* and *end*, and returns an array containing the elements of the array from element *start* up to, but not including, element *end* (or through the end of the array if *end* is not present ~~**undefined**~~). If *start* is negative, it is treated as (*length*+*start*) where *length* is the length of the array. If *end* is negative, it is treated as (*length*+*end*) where *length* is the length of the array. The following steps are taken:

(The bulleted steps are added before step 1)

- Let *O* be the result of calling ToObject with the **this** value as the argument.
- If *O* is a host object, throw a **TypeError** exception.

1. Let *A* be a new array created as if by the expression **new Array()**.
2. Call the **[[Get]]** method of *O* ~~this object~~ with argument **"length"**.
3. Call ToUint32(Result(2)).
    1. If *end* is not present, set *end* to Result(3).
4. Call ToInteger(*start*).
5. If Result(4) is negative, use max((Result(3)+Result(4)),0); else use min(Result(4),Result(3)).
6. Let *k* be Result(5).
7. If *end* is **undefined**, use *0* ~~Result(3)~~; else use ToInteger(*end*).
8. If Result(7) is negative, use max((Result(3)+Result(7)),0); else use min(Result(7),Result(3)).
9. Let *n* be 0.
10. If *k* is greater than or equal to Result(8), go to step 19.
    1. If *k* > 2147483647, then let *biasSrc* = 4294967296; else let *biasSrc* = 0.
    2. If *n* > 2147483647, then let *biasDst* = 4294967296; else let *biasDst* = 0.
11. Call ToString(*k*-*biasSrc*).
12. If *O* ~~this object~~ has a property named by Result(11), go to step 13; but if *O* ~~this object~~ has no property named by Result(11), then go to step 16.
13. Call ToString(*n*-*biasSrc*).

14. Call the **[[Get]]** method of *O* ~~this object~~ with argument Result(11).

15. Call the **[[Put]]** method of *A* with arguments Result(13) and Result(14).

16. Increase *k* by 1.

17. Increase *n* by 1.

18. Go to step 10.

19. Call the **[[Put]]** method of *A* with arguments **"length"** and *n*.

20. Return *A*.

The **length** property of the **slice** method is **2**.

V0111:

NOTE

*The **slice** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **slice** function can be applied successfully to a host object is implementation-dependent. JScript 5.x does not allow the **slice** function to be applied to a host object.*

In JScript 5.x the **slice** function handles array index property names with numeric values greater than 2^{31}-1 differently from numerically smaller array index property names. As this behavior differs from the base specification and from probable user intent, the use of this function on objects containing such properties should be avoided.

2.1.73 [ECMA-262-1999] Section 15.4.4.11, Array.prototype.sort (comparefn)

V0112:

The elements of this array are sorted. The sort is not necessarily stable (that is, elements that compare equal do not necessarily remain in their original order). If *comparefn* is present ~~not **undefined**~~, it should be a function that accepts two arguments *x* and *y* and returns a negative value if $x < y$, zero if $x = y$, or a positive value if $x > y$.

If *comparefn* is present ~~not **undefined**~~ and is not a consistent comparison function for the elements of this array (see below), the behaviour of **sort** is implementation-defined. Let *len* be ToUint32(**this.length**). If there exist integers *i* and *j* and an object *P* such that all of the conditions below are satisfied then the behaviour of **sort** is implementation-defined:

- $0 \le i < len$
- $0 \le j < len$
- **this** does not have a property with name ToString(*i*)
- *P* is obtained by following one or more **[[Prototype]]** properties starting at **this**
- *P* has a property with name ToString(*j*)

V0113:

Otherwise the following steps are taken.

(The bulleted steps are added before step 1)

- Let *obj* be the result of calling ToObject with the **this** value as the argument.
- If *obj* is a host object, throw a **TypeError** exception.
- If **false** is the result of calling the **[[HasProperty]]** method of *obj* with name **"length"**, return *obj*.

1. Call the **[[Get]]** method of *obj* ~~this object~~ with argument **"length"**.
2. Call ToUint32(Result(1)).
3. Perform an implementation-dependent sequence of calls to the **[[Get]]**, **[[Put]]**, and **[[Delete]]** methods of *obj* ~~this object~~ and to SortCompare (described below), where the first argument for each call to **[[Get]]**, **[[Put]]**, or **[[Delete]]** is a nonnegative integer less than Result(2) and where the arguments for calls to SortCompare are results of previous calls to the **[[Get]]** method.
4. Return *obj* ~~this object~~.

V0114:

The returned object must have the following two properties.

- There must be some mathematical permutation π of the nonnegative integers less than Result(2), such that for every nonnegative integer *j* less than Result(2), if property **old[***j***]** existed, then **new[π(***j***)]** is exactly the same value as **old[***j***],** but if property **old[***j***]** did not exist, then **new[π(***j***)]** does not exist.
- Then for all nonnegative integers *j* and *k*, each less than Result(2), if SortCompare(*j*,*k*) < 0 (see SortCompare below), then **π**(*j*) < **π**(*k*).

Here the notation **old[***j***]** is used to refer to the hypothetical result of calling the **[[Get]]** method of *obj* ~~this object~~ with argument *j* before this function is executed, and the notation **new[***j***]** to refer to the hypothetical result of calling the **[[Get]]** method of *obj* ~~this object~~ with argument *j* after this function has been executed.

V0115:

When the SortCompare operator is called with two arguments *j* and *k*, the following steps are taken:

1. Call ToString(*j*).
2. Call ToString(*k*).
3. If *obj* ~~this object~~ does not have a property named by Result(1), and *obj* ~~this object~~ does not have a property named by Result(2), return **+0**.
4. If *obj* ~~this object~~ does not have a property named by Result(1), return 1.
5. If *obj* ~~this object~~ does not have a property named by Result(2), return -1.
6. Call the **[[Get]]** method of *obj* ~~this object~~ with argument Result(1).
7. Call the **[[Get]]** method of *obj* ~~this object~~ with argument Result(2).
8. Let *x* be Result(6).
9. Let *y* be Result(7).
10. If *x* and *y* are both **undefined**, return **+0**.
11. If *x* is **undefined**, return 1.

12. If *y* is **undefined**, return -1.

    1. If the argument *comparefn* is not present or the value **null**, go to step 16.

13. If the argument *comparefn* is not a function ~~**undefined**~~, throw a **TypeError** exception ~~go to step 16~~.

14. Call *comparefn* with arguments *x* and *y*.

    1. If Result(14) is a Number, return Result(14).
    2. Call ToPrimitive with argument Result(14) and hint Number.
    3. If browser is Internet Explorer 7 or 8 and Result(14.1) is **undefined**, throw a **TypeError** exception.
    4. Call ToNumber with argument Result(14.1).

15. Return Result(14.4).

16. Call ToString(*x*).

17. Call ToString(*y*).

18. If Result(16) < Result(17), return -1.

19. If Result(16) > Result(17), return 1.

20. Return **+0**.

V0116:

NOTE 2

*The **sort** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method. Whether the **sort** function can be applied successfully to a host object is implementation-dependent. JScript 5.x does not allow the **sort** function to be applied to a host object.*

2.1.74 [ECMA-262-1999] Section 15.4.4.12, Array.prototype.splice (start, deleteCount [, item1 [, item2 [, ...]]])

V0117:

When the **splice** method is called with two or more arguments *start*, *deleteCount* and (optionally) *item1*, *item2*, etc., the *deleteCount* elements of the array starting at array index *start* are replaced by the arguments *item1*, *item2*, etc. The following steps are taken:

(The bulleted steps are added before step 1)

- Let *O* be the result of calling ToObject with the **this** value as the argument.
- If *O* is a host object, throw a **TypeError** exception.
- If JScript 5.7 and if **false** is the result of calling the **[[HasProperty]]** method of *O* with name **"length"**, return **undefined**.

1. Let *A* be a new array created as if by the expression **new Array()**.
2. Call the **[[Get]]** method of *O* ~~this object~~ with argument **"length"**.

3. Call ToUint32(Result(2)).
4. Call ToInteger(*start*).
5. If Result(4) is negative, use max((Result(3)+Result(4)),0); else use min(Result(4),Result(3)).
6. Compute min(max(ToInteger(*deleteCount*),0),Result(3)-Result(5)).
7. Let *k* be 0.
8. If *k* equals Result(6), go to step 16.
    1. If Result(5)+*k* > 2147483647, then let *biasSrc* = 4294967296; else let *biasSrc* = 0.
    2. If *k* > 2147483647, then let *biasK* = 4294967296; else let *biasK* = 0.
9. Call ToString(Result(5)+*k*-*biasSrc*).
10. If *O* ~~this object~~ has a property named by Result(9), go to step 11; but if *O* ~~this object~~ has no property named by Result(9), then go to step 14.
11. Call ToString(*k*-*biasK*).
12. Call the **[[Get]]** method of *O* ~~this object~~ with argument Result(9).
13. Call the **[[Put]]** method of *A* with arguments Result(11) and Result(12).
14. Increment *k* by 1.
15. Go to step 8.
16. Call the **[[Put]]** method of *A* with arguments **"length"** and Result(6).
17. Compute the number of additional arguments *item1*, *item2*, etc.
18. If Result(17) is equal to Result(6), go to step 48.
19. If Result(17) is greater than Result(6), go to step 37.
20. Let *k* be Result(5).
21. If *k* is equal to (Result(3)-Result(6)), go to step 31.
    1. If *k*+Result(6) > 2147483647, then let *biasSrc* = 4294967296; else let *biasSrc* = 0.
    2. If *k*+Result(17) > 2147483647, then let *biasDst* = 4294967296; else let *biasDst* = 0.
22. Call ToString(*k*+Result(6)-*biasSrc*).
23. Call ToString(*k*+Result(17)-*biasDst*).
24. If *O* ~~this object~~ has a property named by Result(22), go to step 25; but if *O* ~~this object~~ has no property named by Result(22), then go to step 28.
25. Call the **[[Get]]** method of *O* ~~this object~~ with argument Result(22).
26. Call the **[[Put]]** method of *O* ~~this object~~ with arguments Result(23) and Result(25).
27. Go to step 29.
28. Call the **[[Delete]]** method of *O* ~~this object~~ with argument Result(23).

29. Increase *k* by 1.
30. Go to step 21.
31. Let *k* be Result(3).
32. If *k* is equal to (Result(3)-Result(6)+Result(17)), go to step 48.
33. Call ToString(*k*-1).
34. ~~Call the **[[Delete]]** method of this object with argument Result(33).~~
35. Decrease *k* by 1.
36. Go to step 32.
37. Let *k* be (Result(3)-Result(6)).
38. If *k* is equal to Result(5), go to step 48.
    1. If *k*+Result(6)-1 > 2147483647, then let *biasSrc* = 4294967296; else let *biasSrc* = 0.
    2. If *k*+Result(17)-1 > 2147483647, then let *biasDst* = 4294967296; else let *biasDst* = 0.
39. Call ToString(*k*+Result(6)-1-*biasSrc*).
40. Call ToString(*k*+Result(17)-1-*biasDst*)
41. If *O* ~~this object~~ has a property named by Result(39), go to step 42; but if *O* ~~this object~~ has no property named by Result(39), then go to step 45.
42. Call the **[[Get]]** method of *O* ~~this object~~ with argument Result(39).
43. Call the **[[Put]]** method of *O* ~~this object~~ with arguments Result(40) and Result(42).
44. Go to step 46.
45. Call the **[[Delete]]** method of *O* ~~this object~~ with argument Result(40).
46. Decrease *k* by 1.
47. Go to step 38.
48. Let *k* be Result(5).
49. Get the next argument in the part of the argument list that starts with *item1*; if there are no more arguments, go to step 52.1 ~~53~~.
    1. If *k* > 2147483647, then let *biasK* = 4294967296; else let *biasK* = 0.
50. Call the **[[Put]]** method of *O* ~~this object~~ with arguments ToString(*k*-*biasK*) and Result(49).
51. Increase *k* by 1.
52. Go to step 49.
    1. If Result(6) ≠ Result(17), then go to step 54.
53. Call the **[[Put]]** method of O ~~this object~~ with arguments **"length"** and (Result(3)-Result(6)+Result(17)).
54. Return *A*.

The **length** property of the **splice** method is **2**.

V0118:

NOTE

*The **splice** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **splice** function can be applied successfully to a host object is implementation-dependent. JScript 5.x does not allow the **splice** function to be applied to a host object.*

In JScript 5.x the **splice** function handles array index property names with numeric values greater than 2^{31}-1 differently from numerically smaller array index property names. As this behavior differs from the base specification and from probable user intent, the use of this function on objects containing such properties should be avoided.

2.1.75 [ECMA-262-1999] Section 15.4.4.13, Array.prototype.unshift ([item1 [, item2 [, …]]])

V0119:

The arguments are prepended to the start of the array, such that their order within the array is the same as the order in which they appear in the argument list.

When the **unshift** method is called with zero or more arguments *item1, item2*, etc., the following steps are taken:

(The bulleted steps are added before step 1)

- Let *O* be the result of calling ToObject with the **this** value as the argument.
- If *O* is a host object, throw a **TypeError** exception.
- If JScript 5.7 and if **false** is the result of calling the **[[HasProperty]]** method of *O* with name **"length"**, return **undefined**.

1. Call the **[[Get]]** method of *O* ~~this object~~ with argument **"length"**.
2. Call ToUint32(Result(1)).
3. Compute the number of arguments.
4. Let *k* be Result(2).
5. If *k* is zero, go to step 15.
    1. If *k*-1 > 2147483647, then let *biasSrc* = 4294967296; else let *biasSrc* = 0.
    2. If *k*+Result(3)-1 > 2147483647, then let *biasDst* = 4294967296; else let *biasDst* = 0.
6. Call ToString(*k*-1-*biasSrc*).
7. Call ToString(*k*+Result(3)-1-*biasDest*).
8. If *O* ~~this object~~ has a property named by Result(6), go to step 9; but if *O* ~~this object~~ has no property named by Result(6), then go to step 12.
9. Call the **[[Get]]** method of *O* ~~this object~~ with argument Result(6).

10. Call the **[[Put]]** method of *O* ~~this object~~ with arguments Result(7) and Result(9).

11. Go to step 13.

12. Call the **[[Delete]]** method of *O* ~~this object~~ with argument Result(7).

13. Decrease *k* by 1.

14. Go to step 5.

15. Let *k* be 0.

16. Get the next argument in the part of the argument list that starts with *item1*; if there are no more arguments, go to step 20.1 ~~21~~.

17. Call ToString(*k*).

18. Call the **[[Put]]** method of *O* ~~this object~~ with arguments Result(17) and Result(16).

19. Increase *k* by 1.

20. Go to step 16.

    1. If Result(3) is zero, then go to step 21.1.

21. Call the **[[Put]]** method of *O* ~~this object~~ with arguments **"length"** and (Result(2)+Result(3)).

    1. If Jscript 5.7, return **undefined**.

22. Return (Result(2)+Result(3)).

The **length** property of the **unshift** method is **1**.

V0120:

NOTE

*The **unshift** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **unshift** function can be applied successfully to a host object is implementation-dependent. Jscript 5.x does not allow the **unshift** function to be applied to a host object.*

In Jscript 5.x the **unshift** function handles array index property names with numeric values greater than 2^{31}-1 differently from numerically smaller array index property names. As this behavior differs from the base specification and from probable user intent, the use of this function on objects containing such properties should be avoided.

2.1.76 [ECMA-262-1999] Section 15.4.5.1, [[Put]] (P, V)

V0121:

Array objects use a variation of the **[[Put]]** method used for other native ECMAScript objects ([ECMA-262-1999] section 8.6.2.2).

Assume *A* is an Array object and *P* is a string.

When the **[[Put]]** method of *A* is called with property *P* and value *V*, the following steps are taken:

(The bulleted step is added before step 1)

- If JScript 5.7 and *P* is **"length"**, go to step 12.

1. Call the **[[CanPut]]** method of *A* with name *P*.
2. If Result(1) is **false**, return.
3. If *A* doesn't have a property with name *P*, go to step 7.
4. If P is **"length"**, go to step 12.
5. Set the value of property *P* of *A* to *V*.
6. Go to step 8.
7. Create a property with name *P*, set its value to *V* and give it empty attributes.
8. If *P* is not an array index and not **'4294967295'**, return.
    1. If *P* is **'4294967295'**, go to step 17.
9. If ToUint32(*P*) is less than the value of the **length** property of *A*, then return.
10. Change (or set) the value of the **length** property of *A* to ToUint32(*P*)+1.
11. Return.
12. Compute ToUint32(*V*).
    1. If $0 \le$ ToNumber(*V*) $<$ 4294967296, go to step 14.
13. If Result(12) is not equal to ToNumber(*V*), throw a **RangeError** exception.
14. For every integer *k* that is less than the value of the **length** property of *A* but not less than Result(12), if *A* itself has a property (not an inherited property) named ToString(*k*), then delete that property.
15. Set the value of property *P* of *A* to Result(12).
16. Return.
17. For every integer *k* that is less than the value of the **length** property of *A* but not less than 0, if *A* itself has a property (not an inherited property) named ToString(*k*), then delete that property.
18. Change (or set) the value of the **length** property of *A* to 0.
19. Return.

V0122:

JScript 5.x does not throw a **RangeError** if an attempt is made to set the length property of an array object to a positive, non-integer value less than 2^{32}. If the property named **'4294967295'** of an array object is set, the **length** property of the array is set to 0 and any existing array index named properties are deleted if their names are array indices smaller than the former value of the **length** property.

2.1.77 [ECMA-262-1999] Section 15.4.5.2, length

V0123:

The **length** property of this Array object is always numerically greater than the name of every property whose name is an array index.

The **length** property has the attributes { DontEnum, DontDelete }. However, for JScript 5.7 the **length** property in addition has the ReadOnly attribute.

> The existence of the ReadOnly attribute for JScript 5.7 does not prevent modification of the value of the **length** property of array instances because the **[[Put]]** method for array objects as defined in [ECMA-262-1999] section 15.4.5.1 does not call **[[CanPut]]** for the **length** property. However, the existence of the ReadOnly attribute does affect the result of the **[[CanPut]]** method in any situations where it is actually called. In particular, any object that inherits its **length** property from an array instance has a ReadOnly **length** property.

2.1.78 [ECMA-262-1999] Section 15.5.3.2, String.fromCharCode ([char0 [, char1 [, …]]])

V0124:

Returns a string value containing as many characters as the number of arguments. Each argument specifies one character of the resulting string, with the first argument specifying the first character, and so on, from left to right. An argument is converted to a character by applying the operation ToUint16 ([ECMA-262-1999] section 9.7) and regarding the resulting 16-bit integer as the code point value of a character. If no arguments are supplied, the result is the empty string.

The **length** property of the **fromCharCode** function is ~~**1**~~ **0**.

2.1.79 [ECMA-262-1999] Section 15.5.4, Properties of the String Prototype Object

V0125:

The String prototype object is itself a String object ~~(its **[[Class]]** is **"String"**)~~ whose value is an empty string. For JScript 5.x, the **[[Class]]** of the String prototype object is **"Object".**

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

2.1.80 [ECMA-262-1999] Section 15.5.4.3, String.prototype.valueOf ()

V0126:

Returns this string value.

The **valueOf** function is not generic; it throws a **TypeError** exception if its **this** value is not a String or a String object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

2.1.81 [ECMA-262-1999] Section 15.5.4.7, String.prototype.indexOf (searchString, position)

V0127:

The **length** property of the **indexOf** method is ~~**1**~~ **2**.

2.1.82 [ECMA-262-1999] Section 15.5.4.8, String.prototype.lastIndexOf (searchString, position)

V0128:

The **length** property of the **lastIndexOf** method is ~~1~~ **2**.

2.1.83 [ECMA-262-1999] Section 15.5.4.9, String.prototype.localeCompare (that)

V0129:

The actual return values are left implementation-defined to permit implementers to encode additional information in the result value, but the function is required to define a total ordering on all strings and to return **0** when comparing two strings that are considered canonically equivalent by the Unicode standard.

For JScript 5.x running on Windows, the returned value is determined as follows:

1. Call ToString passing the **this** object as the argument.
2. Call ToString passing *that* as the argument.
3. Call the Windows CompareString system function passing Result(1), Result(2) and the current locale information as arguments. The value **0** is passed as the *dwCmpFlags* argument.
4. Return Result(3).

2.1.84 [ECMA-262-1999] Section 15.5.4.10, String.prototype.match (regexp)

V0130:

Let *string* denote the result of converting the **this** value to a string using ToString.

If *regexp* is not present, return **null**.

If *regexp* is not an object whose **[[Class]]** property is **"RegExp"**, it is replaced with the result of the expression **new RegExp(***regexp***)**. ~~Let *string* denote the result of converting the **this** value to a string.~~ Then do one of the following:

- If *regexp*.**global** is **false**: Return the result obtained by invoking **RegExp.prototype.exec** (see [ECMA-262-1999] section 15.10.6.2) on *regexp* with *string* as parameter.
- If *regexp*.**global** is **true**: Set the *regexp*.**lastIndex** property to 0 and invoke **RegExp.prototype.exec** repeatedly until there is no match. If there is a match with an empty string (in other words, if the value of *regexp*.**lastIndex** is left unchanged), increment *regexp*.**lastIndex** by 1. Let *n* be the number of matches. If *n* = 0, then the value returned is **null**; otherwise the ~~The~~ value returned is an array with the **length** property set to *n* and properties 0 through *n*-1 corresponding to the first elements of the results of all matching invocations of **RegExp.prototype.exec**.

The above change corrects a specification error that is documented in the ES3 errata. JScript 5.x implements the correction.

Because the above function is defined to use the RegExp object and its methods, the output of this function is subject to all of the variances from the base specification that are specified in [ECMA-262-1999] section 15.10 and its subsections.

2.1.85 [ECMA-262-1999] Section 15.5.4.11, String.prototype.replace (searchValue, replaceValue)

V0131:

Let *string* denote the result of converting the **this** value to a string using ToString.

If *replaceValue* is a not function, let *newstring* denote the result of converting *replaceValue* to a string using ToString.

> JScript 5.x converts *replaceValue* to a string prior to converting *searchValue* to a string.

V0132:

~~Otherwise, let *newstring* denote the result of converting *replaceValue* to a string.~~ If *searchValue* is not a regular expression, the result is a string value derived from the original input string by replacing each matched substring with *searchString*. Otherwise, the ~~The~~ result is a string value derived from the original input string by replacing each matched substring with a string derived from *newstring* by replacing characters in *newstring* by replacement text as specified in the following table. These **$** replacements are done left-to-right, and, once such a replacement is performed, the new replacement text is not subject to further replacements. For example, **"$1,$2".replace(/(\$(\d))/g, "$$1-$1$2")** returns **"$1-$11,$1$22"**. A **$** in *newstring* that does not match any of the forms below is left as is.

The **length** property of the **replace** method is 1 rather than 2.

2.1.86 [ECMA-262-1999] Section 15.5.4.12, String.prototype.search (regexp)

V0133:

Let *string* denote the result of converting the **this** value to a string using ToString.

If *regexp* is not present, return **null**.

If *regexp* is not an object whose **[[Class]]** property is **"RegExp"**, it is replaced with the result of the expression **new RegExp(***regexp***)**. ~~Let *string* denote the result of converting the **this** value to a string.~~

The value *string* is searched from its beginning for an occurrence of the regular expression pattern *regexp*. The result is a number indicating the offset within the string where the pattern matched, or -1 if there was no match.

The **length** property of the **search** method is 0 rather than 1.

2.1.87 [ECMA-262-1999] Section 15.5.4.13, String.prototype.slice (start, end)

V0134:

The **length** property of the **slice** method is ~~2~~ **0**.

2.1.88 [ECMA-262-1999] Section 15.5.4.14, String.prototype.split (separator, limit)

V0135:

If *separator* is a regular expression that contains capturing parentheses, then each time *separator* is matched the results (~~including~~ excluding any ~~**undefined**~~ empty string results) of the capturing parentheses are spliced into the output array. (For example,

"A<B>bold</B>and<CODE>coded</CODE>".split(/<(\/)?([^<>]+)>/) evaluates to the array **["A", ~~undefined,~~ "B", "bold", "/", "B", "and", ~~undefined,~~ "CODE", "coded", "/", "CODE", ""]**.)

JScript 5.x does not include empty-string capturing parentheses result values in the output array. Note that such results would be **undefined** result values according to the base specification however, as specified in [ECMA-262-1999] section 15.10 and its subsections. JScript produces empty string values for unmatched capturing parentheses.

V0136:

When the **split** method is called, the following steps are taken:

1. Let *S* = ToString(**this**).
2. Let *A* be a new array created as if by the expression **new Array()**.
3. If *limit* is **undefined** or **null**, let *lim* = 2^{32}-1 and go to step 4~~; else let *lim* = ToUint32(*limit*)~~.
    1. Let *lim* = ToInteger(*limit*) however if an exception is thrown while performing ToInteger ignore the exception and let *lim* = 0.
    2. If *lim* is **NaN**, let *lim* = 0 and go to step 4 (not step 3.4).
    3. If *lim* is negative, let *lim* = 2^{32}-1 and go to step 4 (not step 3.4).
    4. Let *lim* be the smaller of *lim* and 2^{32}-1.
4. Let *s* be the number of characters in *S*.
5. Let *p* = 0.
6. If *separator* is a RegExp object (its **[[Class]]** is **"RegExp"**), let *R* = *separator*; otherwise let *R* = ToString(*separator*).
7. If *lim* = 0, return *A*.
8. If *separator* is **undefined**, go to step 33.
9. If *s* = 0, go to step 31.
10. Let *q* = *p*.
11. If *q* = *s*, go to step 28.
12. Call *SplitMatch*(*R*, *S*, *q*) and let *z* be its MatchResult result.
13. If *z* is **failure**, go to step 26.
14. *z* must be a State. Let *e* be *z*'s *endIndex* and let *cap* be *z*'s *captures* array.
15. If *e* = *p*, go to step 26.
16. Let *T* be a string value equal to the substring of *S* consisting of the characters at positions *p* (inclusive) through *q* (exclusive).

    1. If *T* is the empty string, then go to step 19.

17. Call the **[[Put]]** method of *A* with arguments *A*.**length** and *T*.

18. If *A*.**length** = lim, return *A*.

19. Let *p* = *e*.

20. Let *i* = 0.

21. If *i* is equal to the number of elements in *cap*, go to step 10.

22. Let *i* = *i*+1.

    1. If *cap*[*i*] is the empty string or **undefined**, go to step 21.

23. Call the **[[Put]]** method of *A* with arguments *A*.**length** and *cap*[*i*].

24. If *A*.**length** = *lim*, return *A*.

25. Go to step 21.

26. Let *q* = *q*+1.

27. Go to step 11.

28. Let *T* be a string value equal to the substring of *S* consisting of the characters at positions *p* (inclusive) through *s* (exclusive).

29. Call the **[[Put]]** method of *A* with arguments *A*.**length** and *T*.

30. Return *A*.

31. Call *SplitMatch*(*R*, *S*, 0) and let *z* be its MatchResult result.

32. If *z* is not **failure**, return *A*.

33. Call the **[[Put]]** method of *A* with arguments "**0**" and *S*.

34. Return *A*.

2.1.89 [ECMA-262-1999] Section 15.5.4.17, String.prototype.toLocaleLowerCase ()

V0137:

This function works exactly the same as **toLowerCase** except that its result is intended to yield the correct result for the host environment's current locale, rather than a locale-independent result. There will only be a difference in the few cases (such as Turkish) where the rules for that language conflict with the regular Unicode case mappings.

For JScript 5.x running on Windows, the returned string is determined as follows:

1. Call ToString passing the **this** object as the argument.
2. Call the Windows LCMapString system function passing Result(1) and the current locale information. The value LC_MAP_LOWERCASE is passed as the map flags argument.
3. Return Result(2).

2.1.90 [ECMA-262-1999] Section 15.5.4.19, String.prototype.toLocaleUpperCase ()

V0138:

This function works exactly the same as **toUpperCase** except that its result is intended to yield the correct result for the host environment's current locale, rather than a locale-independent result. There will only be a difference in the few cases (such as Turkish) where the rules for that language conflict with the regular Unicode case mappings.

For JScript 5.x running on Windows, the returned string is determined as follows:

1. Call ToString passing the **this** object as the argument.
2. Call the Windows LCMapString system function passing Result(1) and the current locale information. The value LC_MAP_UPPERCASE is passed as the map flags argument.
3. Return Result(2).

2.1.91 [ECMA-262-1999] Section 15.7.4, Properties of the Number Prototype Object

V0139:

In the following descriptions of functions that are properties of the Number prototype object, the phrase "this Number object" refers to the object that is the **this** value for the invocation of the function, if the **this** value is an object; a **TypeError** exception is thrown if the **this** value is neither a number nor ~~not~~ an object for which the value of the internal **[[Class]]** property is **"Number"**. Also, the phrase "this number value" refers to the number that is the **this** value or the number value represented by this Number object, that is, the value of the internal **[[Value]]** property of this Number object.

2.1.92 [ECMA-262-1999] Section 15.7.4.2, Number.prototype.toString (radix)

V0140:

If *radix* is **null** or **undefined**, throw a **TypeError** exception.

Let *radNumber* be the result of calling ToNumber with *radix* as the argument.

If *radNumber* is **NaN**, throw a **TypeError** exception.

Let *radInteger* be the result of calling ToInteger with *radNumber* as the argument.

If *radix* is not present or *radInteger* is the number 10 ~~or undefined~~, then this number value is given as an argument to the ToString operator; the resulting string value is returned.

If *radInteger* ~~radix~~ is an integer from 2 to 36, but not 10, the result is a string, the choice of which is implementation-dependent.

For JScript 5.x the result string consists of a representation of this number value expressed using the radix that is the value of *radInteger*. Letters a-z are used for digits with values 10 through 35. The algorithm used to generate the string representation is the algorithm specified in [ECMA-262-1999] section 9.8.1 generalized for radixes other than 10.

Otherwise throw a **TypeError** exception.

The **toString** function is not generic; it throws a **TypeError** exception if its **this** value is not a Number or a Number object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

2.1.93 [ECMA-262-1999] Section 15.7.4.3, Number.prototype.toLocaleString ()

V0141:

Produces a string value that represents the value of the Number formatted according to the conventions of the host environment's current locale. This function is implementation-dependent, and it is permissible, but not encouraged, for it to return the same thing as **toString**.

For JScript 5.x running on Windows, the string is determined as follows:

1. If this number value is an integer, return the result of calling ToString with this number value as the argument.
2. If this number value is **NaN**, then return the string value **"NaN".**
3. If this number value is **+Infinity** or **-Infinity**, then return the statically localized string that describes such a value.
4. Create a string value using the algorithm of **Number.prototype.toFixed** with this number value as the **this** value and the actual number of significant decimal digits of this number value as the argument.
5. Call the Windows GetNumberFormat system function passing Result(4) and the current locale information. The values 0 and NULL are passed as the format flags and the lpFormat arguments.
6. If the call in step 5 succeeded, then return Result(5).
7. If the calls in either step 3 or step 5 failed, then return the result of calling the standard built-in **Date.prototype.toString** with Result(1) as its this object.
8. Call the Windows OLE Automation function VariantChangeType passing Result(4) and the current locale information.
9. Return the string value corresponding to Result(11).

2.1.94 [ECMA-262-1999] Section 15.7.4.4, Number.prototype.valueOf ()

V0142:

Returns this number value.

The **valueOf** function is not generic; it throws a **TypeError** exception if its **this** value is not a Number or a Number object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

2.1.95 [ECMA-262-1999] Section 15.7.4.5, Number.prototype.toFixed (fractionDigits)

V0143:

Return a string containing the number represented in fixed-point notation with *fractionDigits* digits after the decimal point. If *fractionDigits* is **undefined**, 0 is assumed. Specifically, perform the following steps:

```
1. Let $f$ be ToInteger(*fractionDigits*). (If *fractionDigits* is **undefined**, this step produces the value **0**).
    1. If $f$ is +∞, or -∞, then let $f$ be 0.
2. If $f < 0$ or $f > 20$, throw a **RangeError** exception.
3. Let $x$ be this number value.
4. If $x$ is **NaN**, return the string **"NaN"**.
5. Let $s$ be the empty string.
6. If $x \geq 0$, go to step 9.
7. Let $s$ be **"-"**.
8. Let $x = -x$.
9. ~~If $x \geq 10^{21}$, let $m$ = ToString($x$) and go to step 20.~~
    1. Let *scaledX* be $x*10^f$.
    2. If *scaledX* ≥ 0.50 and *scaledX* < 0.95, let $x$ be 0.
10. Let $n$ be an integer for which the exact mathematical value of $n \times 10^f - x$ is as close to zero as possible. If there are two such $n$, pick the larger $n$.
11. If $n = 0$, let $m$ be the string **"0"**. Otherwise, let $m$ be the string consisting of the digits of the decimal representation of $n$ (in order, with no leading zeroes).
12. If $f = 0$, go to step 20.
13. Let $k$ be the number of characters in $m$.
14. If $k > f$, go to step 18.
15. Let $z$ be the string consisting of $f+1-k$ occurrences of the character '0'.
16. Let $m$ be the concatenation of strings $z$ and $m$.
17. Let $k = f + 1$.
18. Let a be the first $k-f$ characters of $m$, and let $b$ be the remaining $f$ characters of $m$.
19. Let $m$ be the concatenation of the three strings $a$, **"."**, and $b$.
20. Return the concatenation of the strings $s$ and $m$.
```

V0144:

An implementation is permitted to extend the behavior of **toFixed** for values of *fractionDigits* less than 0 or greater than 20. In this case **toFixed** would not necessarily throw **RangeError** for such values.

Jscript 5.x under Internet Explorer 7 or 8 treats as if it were the value 0 any value of *fractionDigits* that when converted to an integer is equal to either +∞, or -∞.

In situations where the absolute value of the number value times 10^f is in the interval [0.50, 0.95), Jscript 5.x under Internet Explorer 7 or 8 produces its result as if the number value were 0.

2.1.96 [ECMA-262-1999] Section 15.7.4.6, Number.prototype.toExponential (fractionDigits)

V0145:

1. Let *x* be the **this** number value.
2. Let *f* be ToInteger(*fractionDigits*).
    1. If browser is Internet Explorer 7 or 8 and *f* is +∞ or -∞, then let *f* be 0.
3. If *x* is **NaN**, return the string **"NaN"**.
4. Let *s* be the empty string.
5. If $x \geq 0$, go to step 8.
6. Let *s* be **"-"**.
7. Let $x = -x$.
8. If $x = +\infty$, let m = **"Infinity"** and go to step 30.
9. If *fractionDigits* was not passed as an argument ~~is **undefined**~~, go to step 14.
10. If $f < 0$ or $f > 20$, throw a **RangeError** exception.
11. If $x = 0$, go to step 16.
12. Let *e* and *n* be integers such that $10^f \leq n < 10^{f+1}$ and for which the exact mathematical value of $n \times 10^{e-f} - x$ is as close to zero as possible. If there are two such sets of *e* and *n*, pick the *e* and *n* for which $n \times 10^{e-f}$ is larger.
13. Go to step 20.
14. If $x \neq 0$, go to step 19.
15. Let $f = 0$.
16. Let *m* be the string consisting of *f*+1 occurrences of the character '0'.
17. Let $e = 0$.
18. Go to step 21.
19. Let *e*, *n*, and *f* be integers such that $f \geq 0$, $10^f \leq n < 10^{f+1}$, the number value for $n \times 10^{e-f}$ is *x*, and *f* is as small as possible. Note that the decimal representation of *n* has *f*+1 digits, *n* is not divisible by 10, and the least significant digit of *n* is not necessarily uniquely determined by these criteria.
20. Let *m* be the string consisting of the digits of the decimal representation of *n* (in order, with no leading zeroes).
21. If $f = 0$, go to step 24.
22. Let *a* be the first character of *m*, and let *b* be the remaining *f* characters of *m*.
23. Let *m* be the concatenation of the three strings *a*, **"."**, and *b*.
24. If $e = 0$, let c = **"+"** and d = **"0"** and go to step 29.
25. If $e > 0$, let c = **"+"** and go to step 28.

26. Let *c* = **"-"**.

27. Let *e* = -*e*.

28. Let *d* be the string consisting of the digits of the decimal representation of *e* (in order, with no leading zeroes).

29. Let *m* be the concatenation of the four strings *m*, **"e"**, *c*, and *d*.

30. Return the concatenation of the strings *s* and *m*.

The **length** property of the **toExponential** method is **1**.

V0146:

An implementation is permitted to extend the behaviour of **toExponential** for values of *fractionDigits* less than 0 or greater than 20. In this case **toExponential** would not necessarily throw **RangeError** for such values. JScript 5.x under Internet Explorer 7 or 8 treats as if it were the value 0 any value of *fractionDigits* that when converted to an integer is equal to either +∞ or -∞.

2.1.97 [ECMA-262-1999] Section 15.7.4.7, Number.prototype.toPrecision (precision)

V0147:

Return a string containing the number represented either in exponential notation with one digit before the significand's decimal point and *precision*-1 digits after the significand's decimal point or in fixed notation with *precision* significant digits. If *precision* is **undefined**, call ToString ([ECMA-262-1999] section 9.8.1) instead.

JScript 5.7 under Internet Explorer 7 or 8 throws a **RangeError** exception if **undefined** is explicitly passed to this function as the *precision* argument. If does not throw the exception if *precision* is **undefined** because no arguments were provided by the caller.

V0148:

Specifically, perform the following steps:

1. Let *x* be the **this** number value.
    1. If running JScript 5.7, and the value **undefined** was explicitly passed as the *precision* argument, throw a **RangeError** exception.
2. If *precision* is **undefined**, return ToString(x).
3. Let *p* be ToInteger(*precision*).
4. If *x* is **NaN**, return the string **"NaN"**.
5. Let *s* be the empty string.
6. If *x* ≥ 0, go to step 9.
7. Let *s* be **"-"**.
8. Let *x* = -*x*.
9. If *x* = +∞, let *m* = **"Infinity"** and go to step 30.

10. If $p < 1$ or $p > 21$, throw a **RangeError** exception.

11. If $x \neq 0$, go to step 15.

12. Let *m* be the string consisting of *p* occurrences of the character '0'.

13. Let $e = 0$.

14. Go to step 18.

15. Let *e* and *n* be integers such that $10^{p-1} \leq n < 10^p$ and for which the exact mathematical value of $n \times 10^{e-p+1} - x$ is as close to zero as possible. If there are two such sets of *e* and *n*, pick the e and *n* for which $n \times 10^{e-p+1}$ is larger.

16. Let *m* be the string consisting of the digits of the decimal representation of *n* (in order, with no leading zeroes).

17. If $e < -6$ or $e \geq p$, go to step 22.

18. If $e = p-1$, go to step 30.

19. If $e \geq 0$, let *m* be the concatenation of the first $e+1$ characters of *m*, the character '.', and the remaining $p-(e+1)$ characters of *m* and go to step 30.

20. Let *m* be the concatenation of the string **"0."**, $-(e+1)$ occurrences of the character '0', and the string *m*.

21. Go to step 30.

22. Let *a* be the first character of *m*, and let *b* be the remaining $p-1$ characters of *m*.

23. Let *m* be the concatenation of the three strings *a*, **"."**, and *b*.

24. If $e = 0$, let $c =$ **"+"** and $d =$ **"0"** and go to step 29.

25. If $e > 0$, let $c =$ **"+"** and go to step 28.

26. Let $c =$ **"-"**.

27. Let $e = -e$.

28. Let *d* be the string consisting of the digits of the decimal representation of *e* (in order, with no leading zeroes).

29. Let *m* be the concatenation of the four strings *m*, **"e"**, *c*, and *d*.

30. Return the concatenation of the strings *s* and *m*.

The **length** property of the **toPrecision** method is 1.

V0149:

An implementation is permitted to extend the behaviour of **toPrecision** for values of *precision* less than 1 or greater than 21. In this case **toPrecision** would not necessarily throw **RangeError** for such values.

JScript 5.x does not extend the behavior of **toPrecision** for values of *precision* less than 1 or greater than 21.

2.1.98 [ECMA-262-1999] Section 15.8.2, Function Properties of the Math Object

V0150:

*Although the choice of algorithms is left to the implementation, it is recommended (but not specified by this standard) that implementations use the approximation algorithms for IEEE 754 arithmetic contained in **fdlibm**, the freely distributable mathematical library from Sun Microsystems (fdlibm-comment@sunpro.eng.sun.com). This specification also requires specific results for certain argument values that represent boundary cases of interest*

JScript 5.x uses the implementation of these functions provided by the Windows C/C++ Run-time Libraries.

2.1.99 [ECMA-262-1999] Section 15.9.1.8, Local Time Zone Adjustment

V0151:

An implementation of ECMAScript is expected to determine the local time zone adjustment. The local time zone adjustment is a value LocalTZA measured in milliseconds which when added to UTC represents the local *standard* time. Daylight saving time is *not* reflected by LocalTZA. The value LocalTZA does not vary with time but depends only on the geographic location.

JScript 5.x uses the available facilities of the host operating system to determine the local time zone adjustment.

2.1.100 [ECMA-262-1999] Section 15.9.1.9, Daylight Saving Time Adjustment

V0152:

If the host environment provides functionality for determining daylight saving time, the implementation of ECMAScript is free to map the year in question to an equivalent year (same leap-year-ness and same starting week day for the year) for which the host environment provides daylight saving time information. The only restriction is that all equivalent years should produce the same result.

JScript 5.x does equivalent year mapping to determine daylight savings time adjustments. The equivalent year that is used is determined according to the following table:

Week day of Jan. 1:	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Non-leap years < 2007	1995	1979	1991	1975	1987	1971	1983
Leap years < 2007	1884	1996	1980	1992	1976	1988	1972
Non-leap years ≥ 2007	2023	2035	2019	2031	2015	2027	2011
Leap years ≥ 2007	2012	2024	2036	2020	2032	2016	2028

2.1.101 [ECMA-262-1999] Section 15.9.1.14, TimeClip (time)

V0153:

The operator TimeClip calculates a number of milliseconds from its argument, which must be an ECMAScript number value. This operator functions as follows:

The change in step 3 below corrects an error in the base specification.

1. If *time* is not finite, return **NaN**.

2. If abs(Result(1)) > **8.64 x 10^{15}**, return **NaN**.

3. Return an implementation-dependent choice of either ToInteger(*time* ~~Result(2)~~) or ToInteger(*time* ~~Result(2)~~) + (**+0**).

   (Adding a positive zero converts -**0** to **+0**.)

JScript 5.x returns ToInteger(time)

2.1.102 [ECMA-262-1999] Section 15.9.4.2, Date.parse (string)

V0154:

If *string* is not present or is the value **null** or **undefined**, the **parse** function returns **NaN**. Otherwise, the ~~The~~ **parse** function applies the ~~ToString~~ ToPrimitive operator to its argument and then applies ToString to that result and interprets the resulting string as a date; it returns a number, the UTC time value corresponding to the date. The string may be interpreted as a local time, a UTC time, or a time in some other time zone, depending on the contents of the string.

V0155:

JScript 5.x parses the string value and produces a value in accordance with the following grammar and rules. If the string can not be recognized starting with the production *DateString* according to these rules, the number value **NaN** is returned.

Date String Syntax

The following lexical grammar defines the tokens that make up date strings.

DateToken **::**

Separator
NumericDateToken
AlphaDateToken
DateComment
OffsetFlag

Separator **:: one of**

, : / <SP>

DateComment **::**

(*DateCommentBody$_{opt}$* **)**

DateCommentBody **::**

DateCommentChars
DateComment$_{opt}$
DateComment
DateCommentBody$_{opt}$

DateCommentChars **::**

DateCommentChar DateCommentChars$_{opt}$

DateCommentChar **::**

DateChar **but not (** or **)**

OffsetFlag **:: one of**

+ -

AlphaDatetoken **::**

AlphaDateComponent period$_{opt}$

AlphaDateComponent **::**

WeekDay
Month
TimeZone
MilitaryTimeZone
AmPmFlag
AdBcFlag

period **::**

.

V0156:

WeekDay **::**

Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday

Month **::**

January
February
March
April
May
June
July
August
September
October
November
December

TimeZone **::**

est
edt
cst

cdt
mst
mdt
pst
pdt
gmt
utc

MilitaryTimeZone **::**

a [lookahead ∉ { .m m d .d p u}]
p [lookahead ∉ { .m m d s}]
b [lookahead ∉ { .c c}]
f [lookahead ∉ { e i}]
m [lookahead ∉ { a d o s}]
s [lookahead ∉ { a e u}]
o [lookahead ≠ c]
n [lookahead ≠ o]
d [lookahead ≠ e]
t [lookahead ∉ { h u}]
w [lookahead ≠ e]
e [lookahead ∉ { d s}]
c [lookahead ∉ { d s}]
g [lookahead ≠ m]
u [lookahead ≠ t]
UniqueMilitaryTimeZone

UniqueMilitaryTimeZone **:: one of**

z y x v r q h i k l

AmPmFlag **::**

am
a.m
pm
p.m

AdBcFlag **::**

ad
a.d
bc
b.c

V0157:

NumericDateToken **::**

NumericDateComponent **-**
NumericDateComponent [lookahead ≠ -]

NumericDateComponent **::**

DateDigit [lookahead ∉ DateDigit]
DateDigit DateDigit [lookahead ∉ DateDigit]
DateDigit DateDigit DateDigit [lookahead ∉ DateDigit]
DateDigit DateDigit DateDigit DateDigit [lookahead ∉ DateDigit]
DateDigit DateDigit DateDigit DateDigit DateDigit [lookahead ∉ DateDigit]
DateDigit DateDigit DateDigit DateDigit DateDigit DateDigit [lookahead ∉ DateDigit]

DateDigit **:: one of**

0 1 2 3 4 5 6 7 8 9

V0158:

Sunday **::**

su
sun
sund
sunda
sunday

Monday **::**

mo
mon
mond
monda
monday

Tuesday **::**

tu
tue
tues
tuesd
tuesda
tuesday

Wednesday **::**

we
wed
wedn
wedne
wednes
wednesd
wednesda
wednesday

Thursday **::**

th
thu
thur
thurs
thursd
thursda
thursday

Friday **::**

fr
fri
frid
frida
friday

Saturday **::**

sa
sat
satu
satur
saturd
saturda
saturday

V0159:

January **::**

ja
jan
janu
janua
januar
january

February **::**

fe
feb
febr
febru
februa
februar
february

March **::**

ma
mar
marc
march

April **::**

ap
apr
apri
april

May **::**

ma
may

June **::**

jun
june

July **::**

ju
jul
july

August **::**

au
aug
augu
augus
august

September **::**

se
sep
sept
septe
septem
septemb
septembe
september

October **::**

oc
oct
octo
octob
octobe
october

November **::**

no
nov
nove
novem
novemb
novembe
november

December **::**

de
dec
dece
decem
decemb
decembe
december

V0160:

Parsing rules for **Date.parse** Date strings:

1. The string to be parsed is converted to lower case before applying these rules.
2. The above grammar defines by means of *NumericDateTokens* or *AlphaDateTokens* the following components of a date object: weekday, year, month, date, hours, minutes, seconds, time zone, AD/BC flag, and AM/PM flag.
3. Any date string has to define at least year, month, and date components. No component can be multiply defined.
4. Except for cases that are explicitly specified otherwise, the components can be in any order.

5. OffsetFlags:

    '+' and '-' (when not following a number) act as offset classifier. The next numeric component following an offset classifier is classified as an offset value. The numeric component doesn't have to immediately follow '**+**/**-**'.

    +offset and -offset cannot be specified before the year field. +/- offsets refers to UTC time zone and set the time zone to UTC. It is an error to have a time zone component following a +/- offset.

6. Time classification of numeric components. The separator char '**:**' acts as a time classifier:

    '**:** ' following a number classifies the previous numeric component as 'hour'.

    '**:**' following a number classified as 'hour' will classify the next numeric component as 'minute'. The next numeric component doesn't have to follow immediately.

    '**:**' following a numeric component classified as 'minute' will classify the next numeric component as 'seconds'. The next number doesn't have to follow immediately.

7. Date classification of numeric components.

    A not classified number with value >= 70 is always classified as year. Even when it is followed by a '**:**' and could be classified as hour. In this case '**:**' is a simple separator.

    A number not classified by a classifier is always classified as a date.

    '/' and '-' separator chars can act as classifiers:

        '/' or '-' following a numeric component classifies that numeric component as month.

        '/' or '-' following a numeric component classified as month will classify the next numeric component as a date. The next numeric component does not have to follow immediately.

        '/' or '-' following a numeric component classified as date will classify the next numeric component as a year. The next numeric component does not have to follow immediately.

8. The week day is ignored regardless of whether it is correct or incorrect.

9. The default value for AD/BC flag is AD.

10. When AM/PM flag is not defined the default interpretation for hours is 24hr notation. The AM flag is ignored when the time is > 13:00:00. When the PM flag is used the time has to be < 12:00.

V0161:

Algorithm for computing the time value:

Via classification, numeric components, and alpha components, numeric values are calculated for: year, month, date, time. The following adjustments are done because of the flags, offsets, and timezones:

1. If the BC/AD flag is BC, year = -year + 1. Note that 1 BC is year 0 and 2 BC is year - 1.

2. If the BC/AD flag is AD and the year value is < 100, then year = year + 1900. This rule allows the short form for year. For example, 99 stands for 1999.

3. The time value (time in the day) is calculated in seconds from the hour, minute, and seconds components. AM/PM flag can change the time value:

    1. If no AM/PM flag is present the time is considered in 24hrs notation and no adjustment is done.

    2. If time >= 12 * 3600 and time < 13 * 3600 and AM then time = time – 12 * 3600 (for example, 12:45 AM means 0:45).

    3. If the PM and time < 12 * 3600, then time = time + 12 * 3600 (for example, 2PM means 14:00).

4. Zone adjustment. The result of 3 is adjusted by the zone disp values specified below. Check the values for TimeZone and MilitaryTimeZone. If 'zone' is the value for a given zone, the time is adjusted by: time = time - zone * 60.

5. Offset adjustment. The offset value applies to the time in UTC zone. Let *nn* be the value of the numeric component following an offset. The formula for the value in seconds that adds up to the UTC time is:

    If *nn* <24

        vOffset = 60 * *nn* * 60

    If *nn* >= 24

        vOffset = 60 * (*nn* modulo 100) + (floor (*nn* / 100)) * 60))

    time = Result(4) - *vOffset* * 60

6. Date adjustment. Set *date* = *date* – 1.

7. Month adjustment. Set *month* = (*month*-1).

8. Final calculation:

    year = *year* + floor(*month* / 12)

    month = Remainder(*month*, 12)

    day = *day* + DayFromYear(*year*)

    day = *day* + DayNumbersForTheMonthOfALeapYear(*month*);

    if *month* >= 2 && year is not a leap then *day* = *day* - 1

    result = *day* * 86400000 + *time*

9. If no time zone was specified, consider this time in the current local time zone and get the UTC displacement of the time.

Time Zone	UTC displacement
est	-5

Time Zone	**UTC displacement**
edt	-4
cst	-6
cdt	-5
mst	-7
mdt	-6
pst	-8
pdt	-7
gmt	0
utc	0

Military Time Zone	**UTC displacement**
z	0
y	12
x	11
w	10
v	9
u	8
t	7
s	6
r	5
q	4
p	3
o	2
n	1
a	-1
b	-2
c	-3
d	-4
e	-5
f	-6
g	-7
h	-8

Military Time Zone	UTC displacement
i	-9
k	-10
l	-10
m	12

2.1.103 [ECMA-262-1999] Section 15.9.4.3, Date.UTC (year, month [, date [, hours [, minutes [, seconds [, ms]]]]])

V0162:

The changes in the following algorithm specify JScript 5.x's behavior when this function is called with fewer than two arguments.

1. If *year* is supplied use ~~Call~~ ToNumber(*year*) ; else use **0**.
2. If *month* is supplied use ~~Call~~ ToNumber(*month*) ; else use **0**.
3. If *date* is supplied use ToNumber(*date*); else use **1**.
4. If *hours* is supplied use ToNumber(*hours*); else use **0**.
5. If *minutes* is supplied use ToNumber(*minutes*); else use **0**.
6. If *seconds* is supplied use ToNumber(*seconds*); else use **0**.
7. If *ms* is supplied use ToNumber(*ms*); else use **0**.
8. If Result(1) is not **NaN** and 0 ≤ ToInteger(Result(1)) ≤ 99, Result(8) is 1900+ToInteger(Result(1)); otherwise, Result(8) is Result(1).
9. Compute MakeDay(Result(8), Result(2), Result(3)).
10. Compute MakeTime(Result(4), Result(5), Result(6), Result(7)).
11. Return TimeClip(MakeDate(Result(9), Result(10))).

The **length** property of the **UTC** function is **7**.

2.1.104 [ECMA-262-1999] Section 15.9.5, Properties of the Date Prototype Object

V0163:

The Date prototype object is itself a Date object (its **[[Class]]** is **"Date"**) whose value is **0** ~~**NaN**~~.

For JScript 5.x, the time value of the Date prototype object is 0 rather than **NaN**.

2.1.105 [ECMA-262-1999] Section 15.9.5.2, Date.prototype.toString ()

V0164:

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the Date in the current time zone in a convenient, human-readable form.

The string is determined as follows:

1. Let *tv* be this time value.
2. If *tv* is **NaN**, return the string **"NaN"**.
3. Let *t* be LocalTime(*tv*).
4. Using *t*, create a string value with the following format, based upon the format description below. The format is: DDDbMMMbddbhh:mm:ssbzzzzzzbyyyyy
5. Return Result(4).

The format is defined as follows:

Date part	Meaning
DDD	The day of the week abbreviation from the set: **Sun Mon Tue Wed Thu Fri Sat**
b	A single space character
MMM	The month name abbreviation from the set: **Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec**
dd	The day of the month as one or two decimal digits, from **1** to **31**.
hh	The number of complete hours since midnight as two decimal digits.
:	The colon character.
mm	The number of complete minutes since the start of the hour as two decimal digits.
ss	The number of complete seconds since the start of the minute as two decimal digits.
zzzzzz	If the local time offset from UTC is an integral number of hours between -8 and -5 inclusive, this is the standard abbreviation for the corresponding North American time zone which is one of: **EST EDT CST CDT MST MDT PST PDT**. Otherwise this is the characters **UTC** followed by a **+** or **-** character corresponding to the sign of the local offset from UTC followed by the two decimal digit hours part of the UTC offset and the two decimal digit minutes part of the UTC offset.
yyyyy	If YearFromTime(*t*) is > then this is 3 or more digits that is the value of YearFromTime(*t*). Otherwise, this is the one or more decimal digits corresponding to the number that is 1 - YearFromTime(t) followed by a single space character followed by the characters **B.C.**

V0165:

NOTE

*For any Date value d with a milliseconds amount of zero, the result of **Date.parse(**d.**toString())** is equal to d.**valueOf()**. See [ECMA-262-1999] section 15.9.4.2.* ~~*It is intended that for any Date value d, the result of Date.prototype.parse(d.toString())(15.9.4.2) is equal to d.*~~

The above change corrects a specification error that is documented in the ES3 errata. JScript 5.x implements the correction.

2.1.106 [ECMA-262-1999] Section 15.9.5.3, Date.prototype.toDateString ()

V0166:

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the "date" portion of the Date in the current time zone in a convenient, human-readable form.

For JScript 5.x running on Windows, the string is determined as follows:

1. Let *tv* be this time value.
2. If *tv* is **NaN**, return the string **"NaN"**.
3. Let *t* be LocalTime(*tv*).
4. Using *t*, create a string value with the following format, based upon the format description given in [ECMA-262-1999] Section 15.9.5.3. The format is **DDDbMMMbddbyyyyy**.
5. Return Result(4).

2.1.107 [ECMA-262-1999] Section 15.9.5.4, Date.prototype.toTimeString ()

V0167:

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the "time" portion of the Date in the current time zone in a convenient, human-readable form.

For JScript 5.x running on Windows, the string is determined as follows:

1. Let *tv* be this time value.
2. If *tv* is **NaN**, return the string **"NaN"**.
3. Let *t* be LocalTime(*tv*).
4. Using *t*, create a string value with the following format, based upon the format description given in [ECMA-262-1999] Section 15.9.5.3. The format is: **hh:mm:ssbzzzzzz**.
5. Return Result(4).

2.1.108 [ECMA-262-1999] Section 15.9.5.5, Date.prototype.toLocaleString ()

V0168:

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the Date in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment's current locale.

For JScript 5.x running on Windows, the string is determined as follows:

1. Using the system locale settings, get the local time value corresponding to the date value. This may include applying any appropriate civil time adjustments.

2. If the year of Result(1) is <= 1600 or >=10000, then return the result of calling the standard built-in **Date.prototype.toString** with Result(1) as its this object.
3. Use the Microsoft Windows GetDateFormat system function to format the date and time corresponding to Result(1). The format flags passed to the function is **DATE_LONGDATE** for most locales. However, if the current locale's language is Arabic or Hebrew the flags passed are **DATE_LONGDATE | DATE_RTLREADING**.
4. If the call in step 3 failed and the current locale language is Hebrew, then throw a **RangeError** exception.
5. Use the Microsoft Windows GetTimeFormat system function to format the date and time corresponding to Result(1). The format flags passed to the default value, **0**.
6. If the calls in either step 3 or step 5 failed, then return the result of calling the standard built-in **Date.prototype.toString** with Result(1) as its this object.
7. Return the string value that is the result of concatenating Result(3), a space character, and Result(5).

2.1.109 [ECMA-262-1999] Section 15.9.5.6, Date.prototype.toLocaleDateString ()

V0169:

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the "date" portion of the Date in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment's current locale.

For JScript 5.x running on Windows, the string is determined as follows:

1. Using the system locale settings, get the local time value corresponding to the date value. This may include applying any appropriate civil time adjustments.
2. If the year of Result(1) is <= 1600 or >= 10000, then return the result of calling the standard built-in **Date.prototype.toString** with Result(1) as its this object.
3. Use the Microsoft Windows GetDateFormat system function to format the date and time corresponding to Result(1). The format flags passed to the function is **DATE_LONGDATE** for most locales. However, if the current locale's language is Arabic or Hebrew the flags passed are **DATE_LONGDATE | DATE_RTLREADING**.
4. If the call in step 3 failed and the current locale language is Hebrew, then throw a **RangeError** exception.
5. If the call in step 3 failed, then return the result of calling the standard built-in **Date.prototype.toString** with Result(1) as its this object.
6. Return the string value that is Result(3).

2.1.110 [ECMA-262-1999] Section 15.9.5.7, Date.prototype.toLocaleTimeString ()

V0170:

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the "time" portion of the Date in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment's current locale.

For JScript 5.x running on Windows, the string is determined as follows:

1. Using the system locale settings, get the local time value corresponding to the date value. This may include applying any appropriate civil time adjustments.
2. If the year of Result(1) is <= 1600 or >= 10000, then return the result of calling the standard built-in **Date.prototype.toString** with Result(1) as its this object.
3. Use the Windows GetTimeFormat system function to format the date and time corresponding to Result(1). The format flags passed to the default value, **0**.
4. If the call in step 3 failed, then return the result of calling the standard built-in **Date.prototype.toString** with Result(1) as its **this** object.
5. Return the string value that is Result(3).

2.1.111 [ECMA-262-1999] Section 15.9.5.28, Date.prototype.setMilliseconds (ms)

V0171:

(The bulleted step is added before step 1)

- If the argument *ms* is not present, throw a **TypeError** exception.

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*ms*).
3. Compute MakeTime(HourFromTime(*t*), MinFromTime(*t*), SecFromTime(*t*), Result(2)).
4. Compute UTC(MakeDate(Day(*t*), Result(3))).
5. Set the **[[Value]]** property of the **this** value to TimeClip(Result(4)).
6. Return the value of the **[[Value]]** property of the **this** value.

2.1.112 [ECMA-262-1999] Section 15.9.5.29, Date.prototype.setUTCMilliseconds (ms)

V0172:

(The bulleted step is added before step 1)

- If the argument *ms* is not present, throw a **TypeError** exception.

1. Let *t* be this time value.
2. Call ToNumber(*ms*).
3. Compute MakeTime(HourFromTime(*t*), MinFromTime(*t*), SecFromTime(*t*), Result(2)).
4. Compute MakeDate(Day(*t*), Result(3)).

5. Set the **[[Value]]** property of the **this** value to TimeClip(Result(4)).
6. Return the value of the **[[Value]]** property of the **this** value.

2.1.113 [ECMA-262-1999] Section 15.9.5.30, Date.prototype.setSeconds (sec [, ms])

V0173:

If *ms* is not specified, this function behaves as if *ms* were specified with the value getMilliseconds( ).

(The bulleted step is added before step 1)

- If the argument *sec* is not present, throw a **TypeError** exception.

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*sec*).
3. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
4. Compute MakeTime(HourFromTime(*t*), MinFromTime(*t*), Result(2), Result(3)).
5. Compute UTC(MakeDate(Day(*t*), Result(4))).
6. Set the **[[Value]]** property of the **this** value to TimeClip(Result(5)).
7. Return the value of the **[[Value]]** property of the **this** value.

The **length** property of the **setSeconds** method is **2**.

2.1.114 [ECMA-262-1999] Section 15.9.5.31, Date.prototype.setUTCSeconds (sec [, ms])

V0174:

If *ms* is not specified, this function behaves as if *ms* were specified with the value getUTCMilliseconds( ).

(The bulleted step is added before step 1)

- If the argument *sec* is not present, throw a **TypeError** exception.

1. Let *t* be this time value.
2. Call ToNumber(*sec*).
3. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
4. Compute MakeTime(HourFromTime(*t*), MinFromTime(*t*), Result(2), Result(3)).
5. Compute MakeDate(Day(*t*), Result(4)).
6. Set the **[[Value]]** property of the **this** value to TimeClip(Result(5)).
7. Return the value of the **[[Value]]** property of the **this** value.

The **length** property of the **setUTCSeconds** method is **2**.

2.1.115 [ECMA-262-1999] Section 15.9.5.33, Date.prototype.setMinutes (min [, sec [, ms]])

V0175:

If *sec* is not specified, this function behaves as if *sec* were specified with the value getSeconds( ).

If *ms* is not specified, this function behaves as if *ms* were specified with the value getMilliseconds( ).

(The bulleted step is added before step 1)

- If the argument *min* is not present, throw a **TypeError** exception.

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*min*).
3. If *sec* is not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).
4. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
5. Compute MakeTime(HourFromTime(*t*), Result(2), Result(3), Result(4)).
6. Compute UTC(MakeDate(Day(*t*), Result(5))).
7. Set the **[[Value]]** property of the **this** value to TimeClip(Result(6)).
8. Return the value of the **[[Value]]** property of the **this** value.

The **length** property of the **setMinutes** method is **3**.

2.1.116 [ECMA-262-1999] Section 15.9.5.34, Date.prototype.setUTCMinutes (min [, sec [, ms]])

V0176:

If *sec* is not specified, this function behaves as if *sec* were specified with the value getUTCSeconds( ).

If *ms* is not specified, this function behaves as if *ms* were specified with the value getUTCMilliseconds( ).

(The bulleted step is added before step 1)

- If the argument *min* is not present, throw a **TypeError** exception.

1. Let *t* be this time value.
2. Call ToNumber(*min*).
3. If *sec* is not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).
4. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
5. Compute MakeTime(HourFromTime(*t*), Result(2), Result(3), Result(4)).
6. Compute MakeDate(Day(*t*), Result(5)).
7. Set the **[[Value]]** property of the **this** value to TimeClip(Result(6)).
8. Return the value of the **[[Value]]** property of the **this** value.

The **length** property of the **setUTCMinutes** method is **3**.

2.1.117 [ECMA-262-1999] Section 15.9.5.35, Date.prototype.setHours (hour [, min [, sec [, ms]]])

V0177:

If *min* is not specified, this function behaves as if *min* were specified with the value getMinutes( ).

If *sec* is not specified, this function behaves as if *sec* were specified with the value getSeconds( ).

If *ms* is not specified, this function behaves as if *ms* were specified with the value getMilliseconds( ).

(The bulleted step is added before step 1)

- <u>If the argument *hour* is not present, throw a **TypeError** exception.</u>

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*hour*).
3. If *min* is not specified, compute MinFromTime(*t*); otherwise, call ToNumber(*min*).
4. If *sec* is not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).
5. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
6. Compute MakeTime(Result(2), Result(3), Result(4), Result(5)).
7. Compute UTC(MakeDate(Day(*t*), Result(6))).
8. Set the **[[Value]]** property of the **this** value to TimeClip(Result(7)).
9. Return the value of the **[[Value]]** property of the **this** value.

The **length** property of the **setHours** method is **4**.

2.1.118 [ECMA-262-1999] Section 15.9.5.36, Date.prototype.setUTCHours (hour [, min [, sec [, ms]]])

V0178:

If *min* is not specified, this function behaves as if *min* were specified with the value getUTCMinutes( ).

If *sec* is not specified, this function behaves as if *sec* were specified with the value getUTCSeconds( ).

If *ms* is not specified, this function behaves as if *ms* were specified with the value getUTCMilliseconds( ).

(The bulleted step is added before step 1)

- <u>If the argument *hour* is not present, throw a **TypeError** exception.</u>

1. Let *t* be this time value.
2. Call ToNumber(*hour*).
3. If *min* is not specified, compute MinFromTime(*t*); otherwise, call ToNumber(*min*).
4. If *sec* is not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).

5. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
6. Compute MakeTime(Result(2), Result(3), Result(4), Result(5)).
7. Compute MakeDate(Day(*t*), Result(6)).
8. Set the **[[Value]]** property of the **this** value to TimeClip(Result(7)).
9. Return the value of the **[[Value]]** property of the **this** value.

The **length** property of the **setUTCHours** method is **4**.

2.1.119 [ECMA-262-1999] Section 15.9.5.36, Date.prototype.setDate (date)

V0179:

(The bulleted step is added before step 1)

- If the argument *date* is not present, throw a **TypeError** exception.

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*date*).
3. Compute MakeDay(YearFromTime(*t*), MonthFromTime(*t*), Result(2)).
4. Compute UTC(MakeDate(Result(3), TimeWithinDay(*t*))).
5. Set the **[[Value]]** property of the **this** value to TimeClip(Result(4)).
6. Return the value of the **[[Value]]** property of the **this** value.

2.1.120 [ECMA-262-1999] Section 15.9.5.37, Date.prototype.setUTCDate (date)

V0180:

(The bulleted step is added before step 1)

- If the argument *date* is not present, throw a **TypeError** exception.

1. Let *t* be this time value.
2. Call ToNumber (*date*).
3. Compute MakeDay(YearFromTime(*t*), MonthFromTime(*t*), Result(2)).
4. Compute MakeDate(Result(3), TimeWithinDay(*t*)).
5. Set the **[[Value]]** property of the **this** value to TimeClip(Result(4)).
6. Return the value of the **[[Value]]** property of the **this** value.

2.1.121 [ECMA-262-1999] Section 15.9.5.38, Date.prototype.setMonth (month [, date])

V0181:

If *date* is not specified, this function behaves as if *date* were specified with the value getDate( ).

(The bulleted step is added before step 1)

- If the argument *month* is not present, throw a **TypeError** exception.

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*month*).
3. If *date* is not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).
4. Compute MakeDay(YearFromTime(*t*), Result(2), Result(3)).
5. Compute UTC(MakeDate(Result(4), TimeWithinDay(*t*))).
6. Set the **[[Value]]** property of the **this** value to TimeClip(Result(5)).
7. Return the value of the **[[Value]]** property of the **this** value.

The **length** property of the **setMonth** method is **2**.

2.1.122 [ECMA-262-1999] Section 15.9.5.39, Date.prototype.setUTCMonth (month [, date])

V0182:

If *date* is not specified, this function behaves as if *date* were specified with the value getUTCDate( ).

(The bulleted step is added before step 1)

- If the argument *month* is not present, throw a **TypeError** exception.

1. Let *t* be this time value.
2. Call ToNumber(*month*).
3. If *date* is not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).
4. Compute MakeDay(YearFromTime(*t*), Result(2), Result(3)).
5. Compute MakeDate(Result(4), TimeWithinDay(*t*)).
6. Set the **[[Value]]** property of the **this** value to TimeClip(Result(5)).
7. Return the value of the **[[Value]]** property of the **this** value.

The **length** property of the **setUTCMonth** method is **2**.

2.1.123 [ECMA-262-1999] Section 15.9.5.40, Date.prototype.setFullYear (year [, month [, date]])

V0183:

If *month* is not specified, this function behaves as if *month* were specified with the value getMonth( ).

If *date* is not specified, this function behaves as if *date* were specified with the value getDate( ).

(The bulleted step is added before step 1)

- If the argument *year* is not present, throw a **TypeError** exception.

1. Let *t* be the result of LocalTime(this time value); but if this time value is **NaN**, let *t* be **+0**.
2. Call ToNumber(*year*).
3. If *month* is not specified, compute MonthFromTime(*t*); otherwise, call ToNumber(*month*).
4. If *date* is not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).
5. Compute MakeDay(Result(2), Result(3), Result(4)).
6. Compute UTC(MakeDate(Result(5), TimeWithinDay(*t*))).
7. Set the **[[Value]]** property of the **this** value to TimeClip(Result(6)).
8. Return the value of the **[[Value]]** property of the **this** value.

The **length** property of the **setFullYear** method is **3**.

2.1.124 [ECMA-262-1999] Section 15.9.5.41, Date.prototype.setUTCFullYear (year [, month [, date]])

V0184:

If *month* is not specified, this function behaves as if *month* were specified with the value getUTCMonth( ).

If *date* is not specified, this function behaves as if *date* were specified with the value getUTCDate( ).

(The bulleted step is added before step 1)

- If the argument *year* is not present, throw a **TypeError** exception.

1. Let *t* be this time value; but if this time value is **NaN**, let *t* be **+0**.
2. Call ToNumber(*year*).
3. If *month* is not specified, compute MonthFromTime(*t*); otherwise, call ToNumber(*month*).
4. If *date* is not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).
5. Compute MakeDay(Result(2), Result(3), Result(4)).
6. Compute MakeDate(Result(5), TimeWithinDay(*t*)).
7. Set the **[[Value]]** property of the **this** value to TimeClip(Result(6)).
8. Return the value of the **[[Value]]** property of the **this** value.

The **length** property of the **setUTCFullYear** method is **3**.

2.1.125 [ECMA-262-1999] Section 15.10.1, Patterns

V0185:

QuantifierPrefix **::**

```
    *
    +
    ?
    { DecimalDigits }
```

{ *DecimalDigits* **, }**
{ *DecimalDigits* **,** *DecimalDigits* **}**
{ *QuantZeroes*$_{opt}$ **1}** *QuantifierPrefix*
{ *QuantZeroes*$_{opt}$ **1,** *QuantZeroes*$_{opt}$ ***1*}** *QuantifierPrefix*

QuantZeroes **::**

QuantZeroes$_{opt}$ **0**

CharacterClass **::**

[]*ClassRanges***]**
[[lookahead ∉ {**^**}] *NonemptyClassRanges* **]**
[^ *NonemptyClassRanges* **]**

2.1.126 [ECMA-262-1999] Section 15.10.2.1, Notation

V0186:

Furthermore, the descriptions below use the following internal data structures:

- A *CharSet* is a mathematical set of characters.
- A *State* is an ordered pair (*endIndex*, *captures*) where *endIndex* is an integer and *captures* is an internal array of *NCapturingParens* values. States are used to represent partial match states in the regular expression matching algorithms. The *endIndex* is one plus the index of the last input character matched so far by the pattern, while *captures* holds the results of capturing parentheses. The *n*th element of *captures* is either a string that represents the value obtained by the *n*th set of capturing parentheses or ~~**undefined**~~ the empty string if the *n*th set of capturing parentheses hasn't been reached yet. Due to backtracking, many states may be in use at any time during the matching process.

2.1.127 [ECMA-262-1999] Section 15.10.2.2, Pattern

V0187:

The production *Pattern* **::** *Disjunction* evaluates as follows:

1. Evaluate *Disjunction* to obtain a Matcher *m*.
2. Return an internal closure that takes two arguments, a string *str* and an integer *index*, and performs the following:
    1. Let *Input* be the given string *str*. This variable will be used throughout the functions in [ECMA-262-1999] section 15.10.2.
    2. Let *InputLength* be the length of *Input*. This variable will be used throughout the functions in [ECMA-262-1999] section 15.10.2.
    3. Let *c* be a Continuation that always returns its State argument as a successful MatchResult.
    4. Let *cap* be an internal array of *NCapturingParens* ~~**undefined**~~ empty string values, indexed 1 through *NCapturingParens*.
    5. Let *x* be the State (*index*, *cap*).
    6. Call *m*(*x*, *c*) and return its result.

Informative comments: A *Pattern* evaluates ("compiles") to an internal function value. **RegExp.prototype.exec** can then apply this function to a string and an offset within the string to determine whether the pattern would match starting at exactly that offset within the string, and, if it does match, what the values of the capturing parentheses would be. The algorithms in [ECMA-262-1999] section 15.10.2 are designed so that compiling a pattern may throw a ~~**SyntaxError**~~ **RegExpError** exception; on the other hand, once the pattern is successfully compiled, applying its result function to find a match in a string cannot throw an exception (except for any host-defined exceptions that can occur anywhere such as out-of-memory).

2.1.128 [ECMA-262-1999] Section 15.10.2.3, Disjunction

V0188:

Informative comments: The **|** regular expression operator separates two alternatives. The pattern first tries to match the left *Alternative* (followed by the sequel of the regular expression); if it fails, it tries to match the right *Disjunction* (followed by the sequel of the regular expression). If the left *Alternative*, the right *Disjunction*, and the sequel all have choice points, all choices in the sequel are tried before moving on to the next choice in the left *Alternative*. If choices in the left *Alternative* are exhausted, the right *Disjunction* is tried instead of the left *Alternative*. Any capturing parentheses inside a portion of the pattern skipped by **|** produce ~~**undefined**~~ empty string values ~~instead of strings~~. Thus, for example,

/a|ab/.exec("abc")

returns the result **"a"** and not **"ab"**. Moreover,

/((a)|(ab))((c)|(bc))/.exec("abc")

returns the array

["abc", "a", "a", ~~undefined~~ "", "bc", ~~undefined~~ "", "bc"]

and not

["abc", "ab", ~~undefined~~ "", "ab", "c", "c", ~~undefined~~ ""]

2.1.129 [ECMA-262-1999] Section 15.10.2.5, Term

V0189:

The production *Term* **::** *Atom Quantifier* evaluates as follows:

1. Evaluate *Atom* to obtain a Matcher *m*.
2. Evaluate *Quantifier* to obtain the three results: an integer *min*, an integer (or ∞) *max*, and boolean *greedy*.
3. If *max* is finite and less than *min*, then throw a ~~**SyntaxError**~~ **RegExpError** exception.
4. Let *parenIndex* be the number of left capturing parentheses in the entire regular expression that occur to the left of this production expansion's *Term*. This is the total number of times the *Atom* **:: (** *Disjunction* **)** production is expanded prior to this production's *Term* plus the total number of *Atom* **:: (** *Disjunction* **)** productions enclosing this *Term*.
5. Let *parenCount* be the number of left capturing parentheses in the expansion of this production's *Atom*. This is the total number of *Atom* **:: (** *Disjunction* **)** productions enclosed by this production's *Atom*.

6. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:

    1. Call RepeatMatcher(*m*, *min*, *max*, *greedy*, *x*, *c*, *parenIndex*, *parenCount*) and return its result.

V0190:

The internal helper function *RepeatMatcher* takes eight parameters, a Matcher *m*, an integer *min*, an integer (or ∞) *max*, a boolean *greedy*, a State *x*, a Continuation *c*, an integer *parenIndex*, and an integer *parenCount*, and performs the following:

1. If *max* is zero, then call *c*(*x*) and return its result.

2. Create an internal Continuation closure *d* that takes one State argument *y* and performs the following:

    1. If *min* is zero and *y*'s *endIndex* is equal to *x*'s *endIndex*, then return **failure**.

    2. If *min* is zero then let *min2* be zero; otherwise let *min2* be *min*-1.

    3. If *max* is ∞, then let *max2* be ∞; otherwise let *max2* be *max*-1.

    4. Call RepeatMatcher(*m*, *min2*, *max2*, *greedy*, *y*, *c*, *parenIndex*, *parenCount*) and return its result.

3. Let *cap* be a fresh copy of *x*'s *captures* internal array.

4. ~~For every integer *k* that satisfies *parenIndex* < *k* and *k* ≤ *parenIndex*+*parenCount*, set *cap*[*k*] to **undefined**.~~

5. Let *e* be *x*'s *endIndex*.

6. Let *xr* be the State (*e*, *cap*).

7. If *min* is not zero, then call *m*(*xr*, *d*) and return its result.

8. If *greedy* is **true**, then go to step 12.

9. Call *c*(*x*) and let *z* be its result.

10. If *z* is not **failure**, return *z*.

11. Call *m*(*xr*, *d*) and return its result.

12. Call *m*(*xr*, *d*) and let *z* be its result.

13. If *z* is not **failure**, return *z*.

14. Call *c*(*x*) and return its result.

V0191:

The above ordering of choice points can be used to write a regular expression that calculates the greatest common divisor of two numbers (represented in unary notation). The following example calculates the gcd of 10 and 15:

```
"aaaaaaaaaa,aaaaaaaaaaaaaaa".replace(/^(a+)\1*,\1+$/,"$1")
```

which returns the gcd in unary notation **"aaaaa"**.

~~Step 4 of the *RepeatMatcher* clears *Atom's* captures each time *t* is repeated. We can see its 106oolean106 in the regular expression~~

~~/(z)((a+)?(b+)?(c))*/.exec("zaacbbbcac")~~

~~which returns the array~~

~~["zaacbbbcac", "z", "ac", "a", undefined, "c"]~~

~~and not~~

~~["zaacbbbcac", "z", "ac", "a", "bbb", "c"]~~

~~because each iteration of the outermost * clears all captured strings contained in the quantified *Atom*, which in this case includes capture strings numbered 2, 3, and 4.~~

Jscript 5.x does not clear an *Atom's* captures each time the *Atom* is repeated.

2.1.130 [ECMA-262-1999] Section 15.10.2.7, Quantifier

V0192:

The productions *QuantifierPrefix* **:: {** *QuantZeroes*$_{opt}$ **1}** *QuantifierPrefix* and *QuantifierPrefix* **:: {** *QuantZeroes*$_{opt}$ **1,** *QuantZeroes*$_{opt}$ **1}** evaluate by returning the result of evaluating *QuantifierPrefix*.

2.1.131 [ECMA-262-1999] Section 15.10.2.8, Atom

V0193:

The form **(?!** *Disjunction* **)** specifies a zero-width negative lookahead. In order for it to succeed, the pattern inside *Disjunction* must fail to match at the current position. The current position is not advanced before matching the sequel. *Disjunction* can contain capturing parentheses, but backreferences to them only make sense from within *Disjunction* itself. Backreferences to these capturing parentheses from elsewhere in the pattern always return **~~undefined~~** the empty string because the negative lookahead must fail for the pattern to succeed. For example,

/(.*?)a(?!(a+)b\2c)\2(.*)/.exec("baaabaac")

looks for an **a** not immediately followed by some positive number *n* of **a**'s, a **b**, another *n* **a**'s (specified by the first **\2**) and a **c**. The second **\2** is outside the negative lookahead, so it matches against **undefined** and therefore always succeeds. The whole expression returns the array:

["baaabaac", "ba", ~~undefined~~ "", "abaac"]

2.1.132 [ECMA-262-1999] Section 15.10.2.9, AtomEscape

V0194:

The production *AtomEscape* **::** *DecimalEscape* evaluates as follows:

1. Evaluate *DecimalEscape* to obtain an EscapeValue *E*.
2. If *E* is not a character then go to step 6.
3. Let *ch* be *E*'s character.

4. Let *A* be a one-element CharSet containing the character *ch*.

5. Call *CharacterSetMatcher*(*A*, **false**) and return its Matcher result.

6. *E* must be an integer. Let *n* be that integer.

7. If *n*=0 or *n*>*NCapturingParens* then return **failure** ~~throw a **SyntaxError** exception~~.

8. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:

    1. Let *cap* be *x*'s *captures* internal array.

    2. Let *s* be *cap*[*n*].

    3. If *s* is ~~**undefined**~~ the empty string, return **failure** ~~then call *c*(*x*) and return its result~~.

    4. Let *e* be *x*'s *endIndex*.

    5. Let *len* be *s*'s length.

    6. Let *f* be *e*+*len*.

    7. If *f*>*InputLength*, return **failure**.

    8. If there exists an integer *i* between 0 (inclusive) and *len* (exclusive) such that *Canonicalize*(*s*[*i*]) is not the same character as *Canonicalize*(*Input* [*e*+*i*]), then return **failure**.

    9. Let *y* be the State (*f*, *cap*).

    10. Call *c*(*y*) and return its result.

V0195:

Informative comments: An escape sequence of the form \ followed by a nonzero decimal number *n* matches the result of the *n*th set of capturing parentheses (see [ECMA-262-1999] section 15.10.2.11). It is an error if the regular expression has fewer than *n* capturing parentheses. If the regular expression has *n* or more capturing parentheses but the *n*th one is ~~**undefined**~~ the empty string because it hasn't captured anything, then the backreference always succeeds.

2.1.133 [ECMA-262-1999] Section 15.10.2.12, CharacterClassEscape

V0196:

The production *CharacterClassEscape* **::** **s** evaluates by returning the set of characters containing the characters that are on the right-hand side of the ~~*WhiteSpace* (7.2) or~~ *LineTerminator* ([ECMA-262-1999] section 7.3) production~~s~~ plus the characters <TAB>, <FF>, and <SP> < (characters \u0009, \u000C, and \u0020).

In JScript 5.x, the regular expression \s does not match any Unicode category Zs characters other than those explicitly listed in the preceding paragraph.

2.1.134 [ECMA-262-1999] Section 15.10.2.13, CharacterClass

V0197:

The production *CharacterClass* **::** **[]** *ClassRanges* **]** evaluates by returning the result of unioning the CharSet containing the one character **]** with the result of evaluating *ClassRanges*.

V0198:

The production *CharacterClass* **:: [** [lookahead ∉ {**^**}] *NonemptyClassRanges* **]** evaluates by evaluating *NonemptyClassRanges* to obtain a CharSet and returning that CharSet and the boolean **false**.

V0199:

The production *CharacterClass* **:: [^** *NonemptyClassRanges* **]** evaluates by evaluating *NonemptyClassRanges* to obtain a CharSet and returning that CharSet and the boolean **true**.

2.1.135 [ECMA-262-1999] Section 15.10.2.15, NonemptyClassRanges

V0200:

The internal helper function *CharacterRange* takes two CharSet parameters *A* and *B* and performs the following:

1. If *A* does not contain exactly one character or *B* does not contain exactly one character then throw a ~~**SyntaxError**~~ **RegExpError** exception.
2. Let *a* be the one character in CharSet *A*.
3. Let *b* be the one character in CharSet *B*.
4. Let *i* be the code point value of character *a*.
5. Let *j* be the code point value of character *b*.
6. If *i* > *j* then throw a ~~**SyntaxError**~~ **RegExpError** exception.
7. Return the set containing all characters numbered *i* through *j*, inclusive.

2.1.136 [ECMA-262-1999] Section 15.10.2.19, ClassEscape

V0201:

The production *ClassEscape* **::** *DecimalEscape* evaluates as follows:

1. Evaluate *DecimalEscape* to obtain an EscapeValue *E*.
2. If *E* is not a character then return the empty CharSet ~~throw a **SyntaxError** exception~~.
3. Let *ch* be *E*'s character.
4. Return the one-element CharSet containing the character *ch*.

V0202:

Informative comments: A *ClassAtom* can use any of the escape sequences that are allowed in the rest of the regular expression except for **\b**, **\B**, and backreferences. Inside a *CharacterClass*, **\b** means the backspace character, while **\B** and backreferences are ignored ~~raise errors. Using a backreference inside a ClassAtom causes an error~~.

2.1.137 [ECMA-262-1999] Section 15.10.4.1, new RegExp (pattern, flags)

V0203:

If *pattern* is an object *R* whose **[[Class]]** property is **"RegExp"** and *flags* is **undefined**, then 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* whose **[[Class]]** property is **"RegExp"** and *flags* is not **undefined**, then throw a ~~**TypeError**~~ **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.

V0204:

If *F* contains any character other than **"g"**, **"i"**, or **"m"**, ~~or if it contains the same one more than once,~~ then throw a ~~**SyntaxError**~~ **RegExpError** exception.

V0205:

If *P*'s characters do not have the form *Pattern,* then throw a ~~**SyntaxError**~~ **RegExpError** exception. Otherwise let the newly constructed object have a **[[Match]]** property obtained by evaluating ("compiling") *Pattern*. Note that evaluating *Pattern* may throw a ~~**SyntaxError**~~ **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 recognised by **RegExp**, the "\" character must be escaped within the *StringLiteral* to prevent its being removed when the contents of the *StringLiteral* are formed.)

V0206:

The **source** property of the newly constructed object is set to an implementation-defined string value in the form of a *Pattern* based on *P*.

For JScript 5.x, when *pattern* is an object *R* whose **[[Class]]** property is **RegExp** the **source** property of the newly constructed object is set to the same string value as the value of the **source** property of *pattern*. Otherwise, the **source** property of the newly constructed object is set to *P*.

2.1.138 [ECMA-262-1999] Section 15.10.6, Properties of the RegExp Prototype Object

V0208:

The value of the internal **[[Prototype]]** property of the RegExp prototype object is the Object prototype. The value of the internal **[[Class]]** property of the RegExp prototype object is **"RegExp"**~~**"Object"**~~**.**

2.1.139 [ECMA-262-1999] Section 15.10.6.2, RegExp.prototype.exec (string)

V0209:

Performs a regular expression match of *string* against the regular expression and returns an Array object containing the results of the match, or **null** if the string did not match.

The string ToString(*string*) is searched for an occurrence of the regular expression pattern as follows:

1. Let *S* be the value of ToString(*string*).
2. Let *length* be the length of *S*.
3. Let *lastIndex* be the value of the **lastIndex** property.

4. Let *i* be the value of ToInteger(*lastIndex*). However if an exception is thrown while evaluating ToInteger, let *I* = 0, or if Result(1) of the ToInteger algorithm is **NaN**, let *I* = -1.

5. If the **global** property is **false**, let *i* = 0.

6. If ~~*I*~~ *i* < 0 or ~~*I*~~ *i* > *length* then set **lastIndex** to 0 and return **null**.

7. Call **[[Match]]**, giving it the arguments *S* and *i*. If **[[Match]]** returned **failure**, go to step 8; otherwise let *r* be its State result and go to step 10.

8. Let *i* = *i*+1.

9. Go to step 6.

10. Let *e* be *r*'s *endIndex* value.

11. ~~If the **global** property is **true**, set~~ Set **lastIndex** to *e*.

12. Let *n* be the length of *r*'s *captures* array. (This is the same value as [ECMA-262-1999] section 15.10.2.1's *NCapturingParens*.)

    1. The values of the **RegExp.input** and **RegExp.$_** properties are set to *S*.

    2. The value of the **RegExp.index** property is set to the position of the matched substring within the complete string *S*.

    3. The value of the **RegExp.lastIndex** property is set to *e*.

    4. The values of the **RegExp.lastMatch** and **RegExp['$&']** properties are set to the substring of S that was matched.

    5. If *n* is 0, set the values of the **RegExp.lastParen** and **RegExp['$+']** properties are set to the empty string, otherwise set them to the result of calling ToString on the last element of *r*'s *captures* array.

    6. The values of the **RegExp.leftContext** and **RegExp["$`"]** properties are set to the substring of *S*, starting at character position 0 and continuing up to but not including the position of the matched substring within the complete string *S*.

    7. The values of the **RegExp.rightContext** and **RegExp["$'"]** properties are set to the substring of *S*, starting at character position *e* and continuing to the last character of *S*.

    8. The 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 set to the empty string.

    9. For each integer *i* such that *i* > 0 and *i* ≤ min(9,*n*), set the property of **RegExp** that has the name of the string **'$'** concatenated with ToString(*i*) to the *i*th element of *r*'s captures array.

13. Return a new array with the following properties:

    - The **index** property is set to the position of the matched substring within the complete string *S*.

    - The **input** property is set to *S*.

    - The **lastIndex** property is set to *e*.

    - The **length** property is set to *n* + 1.

- The **0** property is set to the matched substring (i.e. the portion of *S* between offset *i* inclusive and offset *e* exclusive).
- For each integer *i* such that ~~*I*~~ *i* > 0 and ~~*I*~~ *i* ≤ *n*, set the property named ToString(*i*) to the *i*th element of *r*'s *captures* array.

2.1.140 [ECMA-262-1999] Section 15.10.6.4, RegExp.prototype.toString ()

V0210:

Let *src* be a string in the form of a *Pattern* representing the current regular expression. *src* may or may not be identical to the **source** property or to the source code supplied to the RegExp constructor; however, if *src* were supplied to the RegExp constructor along with the current regular expression's flags, the resulting regular expression must behave identically to the current regular expression.

For JScript 5.x, *src* is identical to the value of the **source** property.

V0211:

toString returns a string value formed by concatenating the strings "**/**", *src*, and "**/**"; plus "**g**" if the **global** property is **true**, "**i**" if the **ignoreCase** property is **true**, and "**m**" if the **multiline** property is **true.**

For JScript 5.x, the flag characters appear in the order "igm" rather than the order "gim".

V0212:

NOTE

*An implementation may choose to take advantage of src being allowed to be different from the source passed to the RegExp constructor to escape special characters in src. For example, in the regular expression obtained from **new RegExp("/")**, src could be, among other possibilities, **"/"** or **"\/"**. The latter would permit the entire result (**"/\//"**) of the **toString** call to have the form RegularExpressionLiteral.*

JScript 5.x does not do such escaping.

2.1.141 [ECMA-262-1999] Section 15.11.1.1, Error (message)

V0213:

When **Error** is called as a function the call is equivalent to calling the **Error** constructor passing the same arguments. ~~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** (15.11.3.1).~~

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

~~If the argument *message* is not **undefined**, the **message** property of the newly constructed object is set to ToString(*message*).~~

2.1.142 [ECMA-262-1999] Section 15.11.2.1, new Error (message)

V0214:

When the **Error** constructor is called with one argument the following steps are taken:

1. 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).
2. The **[[Class]]** property of the newly constructed Error object is set to **"Error"**.
3. Let *message* be the empty string.
4. Let *number* be **NaN**.
5. If *messageOrNumber* is **undefined**, then go to step 8.
6. Let *number* be ToNumber(*messageOrNumber*).
7. If *number* is not **NaN**, then go to step 9.
8. Let *message* be ToString(*messageOrNumber*).
9. The **description** property of the newly constructed object is set to *message*.
10. ~~If the argument *message* is not **undefined**, the~~ The **message** property of the newly constructed object is set to ~~ToString(*message*)~~ *message*.
11. The **name** property of the newly constructed object is set to "**Error**".
12. If *number* is **NaN**, then go to step 14.
13. The **number** property of the newly constructed object is set to *number*.
14. Return the newly constructed object.

2.1.143 [ECMA-262-1999] Section 15.11.4, Properties of the Error Prototype Object

V0215:

The Error prototype object is itself an Error object (its **[[Class]]** is ~~**"Error"**~~ **"Object"**).

In JScript 5.x the **[[Class]]** of the Error prototype object is "**Object**".

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

2.1.144 [ECMA-262-1999] Section 15.11.4.3, Error.prototype.message

V0216:

The initial value of **Error.prototype.message** is an implementation-defined string.

In JScript 5.x the initial value is the empty string.

2.1.145 [ECMA-262-1999] Section 15.11.4.4, Error.prototype.toString ()

Returns an implementation defined string.

In JScript 5.8 under Internet Explorer 7 or 8, the returned string is determined as follows:

1. Let *name* be the result of calling the **[[Get]]** method of the **this** object with argument "**name**".
2. If *name* is not **undefined**, let *name* be ToString(*name*). If ToString throws an exception, ignore the exception and set *name* to **undefined**.
3. Let *message* be the result of calling the **[[Get]]** method of the **this** object with argument **"message"**.
4. If *message* is not **undefined**, let *message* be ToString(*message*). If ToString throws an exception ignore the exception and set *message* to **undefined**.
5. if *name* and *message* are both **undefined**, then return the string value **"[object Error]".**
6. If *name* is **undefined**, return *message*.
7. If *message* is **undefined**, return *name*.
8. Concatenate *name* and the string value **": "**.
9. Concatenate Result(8) and *message*.
10. Return Result(9)

In JScript 5.7 the returned string is determined as follows:

1. Return the string value **"[object Error]".**

2.1.146 [ECMA-262-1999] Section 15.11.5, Properties of Error Instances

V0217:

Error instances inherit properties from their **[[Prototype]]** object as specified above and also have the following properties. ~~Error instances have no special properties beyond those inherited from the Error prototype object.~~

2.1.147 [ECMA-262-1999] Section 15.11.6.2, RangeError

V0218:

Indicates a numeric value has exceeded the allowable range. See [ECMA-262-1999] sections 15.4.2.2, 15.4.5.1, 15.7.4.5, 15.7.4.6, and 15.7.4.7.

Also see the following sections in [MS-ES3EX]:

- VBArray.prototype.getItem (dim1 [, dim2, [dim3, ...]])
- VBArray.prototype.lbound ([dimension])
- VBArray.prototype.ubound ([dimension])

2.1.148 [ECMA-262-1999] Section 15.11.6.4, SyntaxError

V0219:

Indicates that a parsing error has occurred. See [ECMA-262-1999] sections 7.9, 7.9.1, 7.9.2, 15.1.2.1, 15.3.2.1, ~~15.10.2.5, 15.10.2.9, 15.10.2.15,~~ and 15.10.2.19~~, and 15.10.4.1~~.

Also see the following section in [MS-ES3EX]:

- parse (text [, reviver])

2.1.149 [ECMA-262-1999] Section 15.11.6.5, TypeError

V0220:

Indicates the actual type of an operand is different than the expected type. See [ECMA-262-1999] sections 8.6.2, 8.6.2.6, 8.7.1, 9.9, 11.2.2, 11.2.3, 11.4.1, 11.8.6, 11.8.7, 15.2.4.7, 15.3.4, 15.3.4.2, 15.3.4.3, 15.3.4.4, 15.3.5.3, 15.4.4.2, 15.4.4.3, 15.4.4.5, 15.4.4.6, 15.4.4,7, 15.4.4.8, 15.4.4.9, 15.4.4.10, 15.4.4.11, 15.4.4.12, 15.4.4.13, 15.5.4.2, 15.5.4.3, 15.6.4, 15.6.4.2, 15.6.4.3, 15.7.4, 15.7.4.2, 15.7.4.4, 15.9.5, 15.9.5.9, 15.9.5.27, 15.9.5.28, 15.9.5.29, 15.9.5.30, 15.9.5.31, 15.9.5.33, 15.9.5.34, 15.9.5.35, 15.9.5.36, 15.9.5.37, 15.9.5.38, 15.9.5.39, 15.9.5.40, 15.9.5.41, ~~15.10.4.1, and~~ 15.10.6.

Also see the following sections in [MS-ES3EX]:

- RuntimeObject
- GetObject
- stringify (value [, replacer [, space]])
- new Enumerator ([collection])
- Enumerator.prototype.atEnd ( )
- Enumerator.prototype.item ( )
- Enumerator.prototype.moveFirst ( )
- Enumerator.prototype.moveNext ( )
- VBArray (value)
- new VBArray (value)
- VBArray.prototype.dimensions ( )
- VBArray.prototype.getItem (dim1 [, dim2, [dim3, ...]])
- VBArray.prototype.lbound ([dimension])
- VBArray.prototype.toArray ( )
- VBArray.prototype.ubound ([dimension])
- VBArray.prototype.valueOf ( )
- ActiveXObject (name [, location]))
- new ActiveXObject (name [, location]))

2.1.150 [ECMA-262-1999] Section 15.11.7, NativeError Object Structure

V0221:

When an ECMAScript implementation detects a runtime error, it throws an instance of one of the *NativeError* objects defined in [ECMA-262-1999] section 15.11.6. Each of these objects has the structure described below, differing only in the name used as the constructor name instead of *NativeError*, in the **name** property of the prototype object~~, and in the implementation-defined **message** property of the prototype object~~.

For each error object, references to *NativeError* in the definition should be replaced with the appropriate error object name from [ECMA-262-1999] section 15.11.6.

2.1.151 [ECMA-262-1999] Section 15.11.7.2, NativeError (message)

V0222:

The **[[Prototype]]** property of the newly constructed object is set to the prototype object for this error constructor. The **[[Class]]** property of the newly constructed object is set to **"Error"**.

If the argument *message* is not **undefined**, the **message** property of the newly constructed object is set to ToString(*message*). If the argument *message* is **undefined** the **message** property of the newly constructed property is set to the empty string value.

2.1.152 [ECMA-262-1999] Section 15.11.7.4, New NativeError (message)

V0223:

The **[[Prototype]]** property of the newly constructed object is set to the prototype object for this *NativeError* constructor. The **[[Class]]** property of the newly constructed object is set to **"Error"**.

If the argument *message* is not **undefined**, the **message** property of the newly constructed object is set to ToString(*message*). If the argument *message* is **undefined** the **message** property of the newly constructed property is set to the empty string value.

2.1.153 [ECMA-262-1999] Section 15.11.7.10, NativeError.prototype.message

V0224:

~~The initial value of the **message** property of the prototype for a given *NativeError* constructor is an implementation-defined string.~~

In JScript 5.x *NativeError* prototype objects do not have their own **message** property. Instead they inherit their **message** property from **Error.prototype**.

2.1.154 [ECMA-262-1999] Section 16, Errors

V0266:

An implementation may treat any instance of the following kinds of runtime errors as a syntax error and therefore report it early:

- Improper uses of **return, break,** and **continue.**
- Using the **eval** property other than via a direct call.
- Errors in regular expression literals that are not implementation-defined syntax extensions.
- Attempts to call **PutValue** on a value that is not a reference (for example, executing the assignment statement 3= 4).

2.1.155 [ECMA-262-1999] Section A.1, Lexical Grammar

V0225:

LineTerminator **::** *See* [ECMA-262-1999] *section 7.3*

<LF>
<CR>
~~*<LS>*~~
~~*<PS>*~~

V0226:

MultiLineNotAsteriskChar **::** *See* [ECMA-262-1999] *section 7.4*

SourceCharacter **but not** *asterisk* ***** **or** <NUL>

V0227:

MultiLineNotForwardSlashOrAsteriskChar **::** *See* [ECMA-262-1999] *section 7.4*

SourceCharacter **but not** forward-slash **/** ***or*** asterisk ***** **or** <NUL>

V0228:

FutureReservedWord **:: one of** *See* [ECMA-262-1999] *section 7.5.3*

~~**abstract**~~	**enum**	~~**int**~~	~~**short**~~
~~**boolean**~~	**export**	~~**interface**~~	~~**static**~~
~~**byte**~~	**extends**	~~**long**~~	**super**
~~**char**~~	~~**final**~~	~~**native**~~	~~**synchronized**~~
class	~~**float**~~	~~**package**~~	**throws**
const	~~**goto**~~	~~**private**~~	**transient**
debugger	~~**implements**~~	~~**protected**~~	**volatile**
~~**double**~~	**import**	~~**public**~~	

V0229:

DoubleStringCharacter **::** *See* [ECMA-262-1999] *section 7.8.4*

SourceCharacter **but not** *double-quote* **" or** *backslash* **\\ or** *LineTerminator* **or** <NUL>
\ *EscapeSequence*
LineContinuation

V0230:

SingleStringCharacter **::** *See* [ECMA-262-1999] *section 7.8.4*

SourceCharacter **but not** *single-quote* **' or** *backslash* **\\ or** *LineTerminator* **or** <NUL>
\ *EscapeSequence*
LineContinuation

V0231:

LineContinuation **::** See [ECMA-262-1999] *section 7.8.4*

\ *LineTerminatorSequence*

V0232:

LineTerminatorSequence **::** See [ECMA-262-1999] *section 7.8.4*

<LF>
<CR> [lookahead ∉ *<LF>*]
<CR> <LF>

V0233:

EscapeSequence **::** See [ECMA-262-1999] *section 7.8.4*

CharacterEscapeSequence
OctalEscapeSequence ~~**0** [lookahead ∉ *DecimalDigit*]~~
HexEscapeSequence
UnicodeEscapeSequence
8
9

JScript 5.x also supports *OctalEscapeSequence* as specified in [ECMA-262-1999] section Annex B.1.2. That extension replaces the rule *EscapeSequence* **::** ***0*** [lookahead ∉ *DecimalDigit*] with the rule *EscapeSequence* **::** *OctalEscapeSequence*.

V0234:

SingleEscapeCharacter **:: one of** See [ECMA-262-1999] *section 7.8.4*

' " \ b f n r t ~~**v**~~

V0235:

RegularExpressionFirstChar **::** See [ECMA-262-1999] *section 7.8.5*

NonTerminator **but not * or \ or / or** <NUL>
BackslashSequence
RegularExpressionClass

V0236:

RegularExpressionChar **::** See [ECMA-262-1999] *section 7.8.5*

NonTerminator **but not \ or / or** <NUL>
BackslashSequence
RegularExpressionClass

V0237:

RegularExpressionClass **::** See [ECMA-262-1999] *section 7.8.5*

[*RegularExpressionClassChars* **]**

V0238:

RegularExpressionClassChars **::** See [ECMA-262-1999] *section 7.8.5*

[empty]
RegularExpressionClassChars RegularExpressionClassChar

V0239:

RegularExpressionClassChar **::** *See* [ECMA-262-1999] *section 7.8.5*

NonTerminator **but not] or \ or** <NUL> *BackslashSequence*

V0240:

RegExpFlag **:: one of**

g i m *See* [ECMA-262-1999] *section 7.8.5*

2.1.156 [ECMA-262-1999] Section A.3, Expressions

V0241:

ObjectLiteral **:** *See* [ECMA-262-1999] *section 11.1.5*

{ }
{ *PropertyNameAndValueList* **}**
{ *PropertyNameAndValueList* **, }**

2.1.157 [ECMA-262-1999] Section A.4, Statements

V0242:

Statement **:** *See* [ECMA-262-1999] *section 12*

Block
VariableStatement
EmptyStatement
ExpressionStatement
IfStatement
IterationStatement
ContinueStatement
BreakStatement
ReturnStatement
WithStatement
LabelledStatement
SwitchStatement
ThrowStatement
TryStatement
DebuggerStatement
FunctionDeclaration

V0243:

Block **:** *See* [ECMA-262-1999] *section 12.1*

{ *StatementList*$_{opt}$ **}**
{ *StatementList*$_{opt}$ **};**

V0244:

DebuggerStatement **:** See section 2.1.6

Debugger ***;***

2.1.158 [ECMA-262-1999] Section A.5, Functions and Programs

V0245:

FunctionDeclaration **:** *See* [ECMA-262-1999] *section 13*

function *Identifier$_{opt}$* **(** *FormalParameterList$_{opt}$* **)** **{** *FunctionBody* **}**
JScriptFunction

V0246:

FunctionExpression **:** *See* [ECMA-262-1999] *section 13*

function *Identifier$_{opt}$* **(** *FormalParameterList$_{opt}$* **)** **{** *FunctionBody* **}**
JScriptFunction

V0247:

JScriptFunction **:** *See* [ECMA-262-1999] *section 13*

function *FunctionBindingList* **(** *FormalParameterList$_{opt}$* **)** **{** *FunctionBody* **}**

V0248:

FunctionBindingList **:** *See* [ECMA-262-1999] *section 13*

FunctionBinding
FunctionBindingList, *FunctionBinding*

V0249:

FunctionBinding **:** *See* [ECMA-262-1999] *section 13*

SimpleFunctionBinding
MethodBinding
EventHandlerBinding

V0250:

SimpleFunctionBinding **:** *See* [ECMA-262-1999] *section 13*

Identifier [lookahead ∉ {*NameQualifier, EventDesignator*}]

V0251:

MethodBinding **:** *See* [ECMA-262-1999] *section 13*

ObjectPath NameQualifier Identifier [lookahead ∉ {*NameQualifier, EventDesignator*}]

V0252:

EventHandlerBinding **:** *See* [ECMA-262-1999] *section 13*

ObjectPath EventDesignator Identifier

V0253:

ObjectPath **:** — *See* [ECMA-262-1999] *section 13*

> *Identifier*
> *ObjectPath NameQualifier Identifier*

V0254:

NameQualifier **: .**

V0255:

EventDesignator **: ::**

See [ECMA-262-1999] *section 13*

2.1.159 [ECMA-262-1999] Section A.7, Regular Expressions

V0256:

QuantifierPrefix **::** — *See* [ECMA-262-1999] *section 15.10.1*

> *****
> **+**
> **?**
> **{** *DecimalDigits* **}**
> **{** *DecimalDigits* **, }**
> **{** *DecimalDigits* **,** *DecimalDigits* **}**
> **{** *QuantZeroes*$_{opt}$ **1}** *QuantifierPrefix*
> **{** *QuantZeroes*$_{opt}$ **1,** *QuantZeroes*$_{opt}$ **1}** *QuantifierPrefix*

V0257:

QuantZeroes **::** — *See* [ECMA-262-1999] *section 15.10.1*

> *QuantZeroes*$_{opt}$ **0**

V0258:

CharacterClass **::** — *See* [ECMA-262-1999] *section 15.10.1*

> **[]***ClassRanges***]**
> **[** [lookahead ∉ {**^**}] *NonemptyClassRanges* **]**
> **[^** *NonemptyClassRanges* **]**

2.1.160 [ECMA-262-1999] Section B.1.2, String Literals

V0259:

OctalEscapeSequence **::**

> *OctalDigit* [lookahead ∉ *OctalDigit* ~~*DecimalDigit*~~]
> *ZeroToThree OctalDigit* [lookahead ∉ *OctalDigit* ~~*DecimalDigit*~~]
> *FourToSeven OctalDigit*
> *ZeroToThree OctalDigit OctalDigit*

V0260:

Semantics

- The CV of *EscapeSequence* **::** *OctalEscapeSequence* is the CV of the *OctalEscapeSequence*.
- The CV of *OctalEscapeSequence* **::** *OctalDigit* [lookahead ∉ OctalDigit ~~DecimalDigit~~] is the character whose code point value is the MV of the *OctalDigit*.
- The CV of *OctalEscapeSequence* **::** *ZeroToThree OctalDigit* [lookahead ∉ OctalDigit ~~DecimalDigit~~] is the character whose code point value is (8 times the MV of the *ZeroToThree*) plus the MV of the *OctalDigit*.

2.1.161 [ECMA-262-1999] Section B.2, Additional Properties

V0261:

Some implementations of ECMAScript have included additional properties for some of the standard native objects. This non-normative annex suggests uniform semantics for such properties without making the properties or their semantics part of this standard.

JScript 5.x implements all of the properties listed in [ECMA-262-1999] section B.2. However, in some cases identified below, the definition used by JScript 5.x differs from that in the base specification.

2.1.162 [ECMA-262-1999] Section B.2.3, String.prototype.substr (start, length)

V0262:

The **substr** method takes two arguments, *start* and *length*, and returns a substring of the result of converting this object to a string, starting from character position *start* and running for *length* characters (or through the end of the string is *length* is **undefined**). If *start* is negative, it is treated as zero ~~(sourceLength+start) where sourceLength is the length of the string~~. The result is a string value, not a String object.

V0263:

1. Call ToString, giving it the **this** value as its argument.
2. Call ToInteger(*start*).
3. If *length* is **undefined**, use **+∞**; otherwise call ToInteger(*length*).
4. Compute the number of characters in Result(1).
5. If Result(2) is positive or zero, use Result(2); else use zero ~~max(Result(4)+Result(2),0)~~.
6. Compute min(max(Result(3),0), Result(4)-Result(5)).
7. If Result(6) ≤ 0, return the empty string **""**.
8. Return a string containing Result(6) consecutive characters from Result(1) beginning with the character at position Result(5).

The **length** property of the **substr** method is **2**.

2.1.163 [ECMA-262-1999] Section B.2.4, Date.prototype.getYear ()

V0264:

When the **getYear** method is called with no arguments the following steps are taken:

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return YearFromTime(LocalTime(*t*))~~ − 1900~~.

> For JScript 5.x, **Date.prototype.getYear** is functionally identical to **Date.prototype.getFullYear**.

2.1.164 [ECMA-262-1999] Section B.2.5, Date.prototype.setYear (year)

V0265:

When the **setYear** method is called with one argument *year* the following steps are taken:

(The bulleted step is added before step 1)

- If the argument *year* is not present, throw a **TypeError** exception.

1. Let *t* be the result of LocalTime(this time value); but if this time value is **NaN**, let *t* be **+0**.
2. Call ToNumber(*year*).
3. If Result(2) is **NaN**, set the **[[Value]]** property of the **this** value to **NaN** and return **NaN**.
4. If Result(2) is not **NaN** and 0 ≤ ToInteger(Result(2)) ≤ 99 then Result(4) is ToInteger(Result(2)) ~~+ 1900~~. Otherwise, Result(4) is Result(2).
5. Compute MakeDay(Result(4), MonthFromTime(*t*), DateFromTime(*t*)).
6. Compute UTC(MakeDate(Result(5), TimeWithinDay(*t*))).
7. Set the **[[Value]]** property of the **this** value to TimeClip(Result(6)).
8. Return the value of the **[[Value]]** property of the **this** value.

> For JScript 5.x, **Date.prototype.setYear** is functionally identical to **Date.prototype.setFullYear**.

2.2 Clarifications

The following sub-sections identify clarifications relative to <Target name>.

2.2.1 [ECMA-262-1999] Section 7.8.5, Regular Expression Literals

C0001:

If the call to **new RegExp** generates an error, an implementation may, at its discretion, either report the error immediately while scanning the program, or it may defer the error until the regular expression literal is evaluated in the course of program execution.

> JScript 5.x reports any errors **new RegExp** errors relating to a regular expression literal while scanning the program.

2.2.2 [ECMA-262-1999] Section 8.6.2, Internal Properties and Methods

C0002:

The value of the **[[Prototype]]** property must be either an object or **null**, and every **[[Prototype]]** chain must have finite length (that is, starting from any object, recursively accessing the **[[Prototype]]** property must eventually lead to a **null** value). Whether or not a native object can have a host object as its **[[Prototype]]** depends on the implementation.

JScript 5.x does not permit a native object to have a host object as its **[[Prototype]]**.

2.2.3 [ECMA-262-1999] Section 10.1.1, Function Objects

C0003:

- Internal functions are built-in objects of the language, such as **parseInt** and **Math.exp**. An implementation may also provide implementation-dependent internal functions that are not described in this specification. These functions do not contain executable code defined by the ECMAScript grammar, so they are excluded from this discussion of execution contexts.

In the above paragraph the phrase "internal function" is actually being used as a synonym for "built-in objects" (as defined in [ECMA-262-1999] section 4.3.7) that are functions. The implementation-dependent built-in functions provided by JScript 5.x are described in [ECMA-262-1999] section 15.

2.2.4 [ECMA-262-1999] Section 15.1.2.2, parseInt (string, radix)

C0004:

NOTE

parseInt *may interpret only a leading portion of the string as an integer value; it ignores any characters that cannot be interpreted as part of the notation of an integer, and no indication is given that any such characters were ignored.*

When radix is 0 or ***undefined*** *and the string's number begins with a* ***0*** *digit not followed by an* ***x*** *or* ***X****, then the implementation may, at its discretion, interpret the number either as being octal or as being decimal. Implementations are encouraged to interpret numbers in this case as being decimal.*

Jscript 5.x interprets numbers in this case as being octal.

2.3 Error Handling

There are no additional considerations for error handling.

2.4 Security

There are no additional security considerations.

Copyright © 2017 Microsoft Corporation

3 Change Tracking

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

4 Index