[MS-ES5EX]: Internet Explorer Extensions to the ECMA-262 ECMAScript Language Specification (Fifth Edition)

Revision Summary

Date	Revision History	Revision Class	Comments
09/08/2010	0.1	New	Released new document.
10/13/2010	0.2	Minor	Clarified the meaning of the technical content.
02/10/2011	1.0	Minor	Clarified the meaning of the technical content.
02/28/2011	1.1	Minor	Clarified the meaning of the technical content.
02/22/2012	2.0	Major	Significantly changed the technical content.
07/25/2012	2.1	Minor	Clarified the meaning of the technical content.

Table of Contents

1 Introduction

This document describes extensions provided to the ECMAScript language that are implemented in IE9 Mode in Windows® Internet Explorer® 9 based on the ECMAScript Language Specification 5th Edition [ECMA-262/5], published December 2009. These are described in sections 2.1 through 2.8.

The implementation of the ECMAScript language in IE10 Mode in Windows® Internet Explorer® 10 is based on ECMAScript Language Specification 5.1 Edition [ECMA-262/51], published June 2011. All of the extensions to [ECMA-262/5] described in this document for IE9 Mode also apply to IE10 Mode in Internet Explorer 10. In addition, section 2.9 describes extensions to the ECMAScript Language Specification 5.1 Edition [ECMA-262/51] in IE10 Mode; these extensions are not available in versions prior to Internet Explorer 10.

Sections 1.7 and 2 of this specification are normative and can contain the terms MAY, SHOULD, MUST, MUST NOT, and SHOULD NOT as defined in RFC 2119. All other sections and examples in this specification are informative.

1.1 Glossary

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

1.2 References

References to Microsoft Open Specifications documentation do not include a publishing year because links are to the latest version of the technical documents, which are updated frequently. References to other documents include a publishing year when one is available.

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. Please check the archive site, http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624, as an additional source.

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

[ECMA-262/51] ECMA International, "Standard ECMA-262 ECMAScript Language Specification", 5.1 Edition (June 2011), http://www.ecma-international.org/publications/standards/Ecma-262.htm

[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-ES3] Microsoft Corporation, "Microsoft JScript ECMAScript Language Specification 3rd Edition Standards Support Document".

[MS-ES3EX] Microsoft Corporation, "Microsoft JScript Extensions to the ECMAScript Language Specification 3rd Edition".

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

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

1.3 Extension Overview (Synopsis)

The extensions described in this document were selected for their applicability to [ECMA-262/5].

These extensions are organized based on sections of [ECMA-262/5] as follows.

Section 7, Lexical Conventions

1. Conditional Source Text Processing
2. 7.8.3, Numeric Literals
3. 7.8.4, String Literals

Section 8, Types

Section 9, Type Conversion and Testing

Section 10, Executable Code and Execution Contexts

Section 11, Expressions

Section 12, Statements

Section 13, Function Definition

Section 15, Standard Built-in ECMAScript Objects

1. 15.1.2, Function Properties of the Global Object
2. 15.1.4, Constructor Properties of the Global Object
3. 15.3.5, Properties of Function Instances
4. String.prototype HTML Wrapper Properties
5. 15.9.5, Properties of the Date Prototype Object
6. 15.10.5, Properties of the RegExp Constructor
7. 15.10.6, Properties of the RegExp Prototype Object
8. 15.10.7, Properties of RegExp Instances
9. 15.11.2, The Error Constructor
10. 15.11.5, Properties of Error Instances
11. 15.11.6, Native Error Types Used in This Standard
12. The Debug Object
13. Enumerator Objects

14.[VBArray Objects]

15.[ActiveXObject Objects]

1.3.1 Organization of This Documentation

This document is organized as follows:

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

1.4 Relationship to Standards and Other Extensions

This document defines extensions to [ECMA-262/5]. Variations from [ECMA-262/5] are defined in [MS-ES5].

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

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

1.5 Applicability Statement

This document specifies a set of extensions to the [ECMA-262/5] specifications. The extensions in this document provide access to some features that are unique to Windows® Internet Explorer® 9 and Windows® Internet Explorer® 10 when it is operating in IE9 Mode.

Section 2.9 specifies extensions to the ECMAScript Language Specification 5.1 Edition [ECMA-262/51] in IE10 Mode.

2 Extensions

This section specifies extensions to [ECMA-262/5] that are available in Windows® Internet Explorer® 9 in IE9 Mode.

Section 2.9 specifies extensions to the [ECMA-262/51] in Windows® Internet Explorer® 10 for IE10 Mode.

The extensions are as follows:

- Extensions to Lexical Conventions
- Extensions to Types
- Extensions to Type Conversion and Testing
- Extensions to Executable Code and Execution Contexts
- Extensions to Expressions
- Extensions to Statements
- Extensions to Function Definition
- Extensions to Native ECMAScript Objects

2.1 Extensions to Lexical Conventions

The following section defines Internet Explorer ECMAScript extensions to [ECMA-262/5] lexical conventions.

The extensions are as follows:

- Conditional Source Text Processing
- Global State
- Conditional Processing Algorithm
- Extensions to Numeric Literals
- Extensions to String Literals

2.1.1 Conditional Source Text Processing

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

Each *Program* (see [ECMA-262/5] section 14), whether presented as either a discrete source text or as the argument to the eval built-in function, and each *FunctionBody* (see [ECMA-262/5] section 13) processed by the standard built-in Function constructor ([ECMA-262/5] section 15.3.2.1) has conditional source text processing performed independently upon it.

NOTE

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

2.1.1.1 Global State

The following state is shared by the conditional source text processing of all independent source texts that make up an ECMAScript program (see [ECMA-262/5] section 14). The state is initialized prior to the first such processing as follows:

1. *SubstitutionEnabled* **Boolean** flag with an initial value of **false**.

2. *CCvariables* A set of associations between string valued keys and values. The keys are strings. The values may be either ECMAScript **Number** ([ECMA-262/5] section 8.5) or **Boolean** ([ECMA-262/5] section 8.3) values. The initial associations are defined in the following table.

Key	Initial Value
"_win32"	Defined as **true** if this Internet Explorer ECMAScript implementation is a Microsoft 32-bit–based implementation. Otherwise, this association is not initially defined.
"_win64"	Defined as **true** if this Internet Explorer ECMAScript implementation is a Microsoft 64-bit–based implementation. Otherwise, this association is not initially defined.
"_x86"	Defined as **true** when running on a processor using the x86-based architecture. Otherwise, this association is not initially defined.
"_ia64"	Defined as **true** when running on a processor using the Itanium 64-bit architecture. Otherwise, this association is not initially defined.
"_amd64"	Defined as **true** when running on a processor using the x64 architecture. Otherwise, this association is not initially defined.
"_jscript"	**true**
"_jscript_build"	**Number** value that identifies the specific build of the Internet Explorer ECMAScript implementation that is running.
"_jscript_version"	**Number** value that represents the version of the Internet Explorer ECMAScript language implementation. The value `9` indicates that the implementation only supports features of the Internet Explorer 9 ECMAScript language.
"_microsoft"	Defined as `true` when running on a Microsoft ECMAScript implementation provided by Microsoft. Otherwise, this association is not initially defined.

2.1.1.2 Conditional Processing Algorithm

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

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

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

Syntax

NOTE:

CCInputElementState0 is recognized during top-level conditional processing when *SubstitutionEnabled* is **false**. When recognizing a *RegularExpressionLiteral* in this state, the contextual distinction between *RegularExpressionLiteral* and *DivPunctuator* (see [ECMA-262/5] section 7) must be respected.

CCInputElementState0 **::**

RegularExpressionLiteralStringLiteralCCOnCCSet0CCIf0CCMultiLineComment0CCSingleLinecomment0SourceCharacter

CCOn **::**

@ *CCOnId***/*@** *CCOnId***//@** *CCOnId*

CCOnId **::**

cc_on [lookahead *IdentifierPart*]

CCSet0 **::**

@set [lookahead *IdentifierPart*]

CCIf0 **::**

@if [lookahead *IdentifierPart*]

CCMultiLineComment0 **::**

/* [lookahead ≠ *CCOnId*] *MultiLineCommentChars*$_{opt}$ ***/**

SingleLineComment0 **::**

// [lookahead ≠ *CCOnId*] *SingleLineCommentChars*$_{opt}$

Semantics

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

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

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

2. Use *CCInputElementState0* to recognize the next input element from *source*.

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

1. Set SubstitutionEnable to **true**.

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

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

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

1. Set SubstitutionEnable to **true**.

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

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

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

1. Set SubstitutionEnable to **true**.

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

3. Increment the value of IfNestingLevel by 1.

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

Syntax

NOTE:

CCInputElementState1 is recognized during active conditional processing when *SubstitutionEnabled* is **true**. This may be at the top level or in the clause of an **@if** statement that represents the "true" condition. When recognizing a *RegularExpressionLiteral* in this state the contextual distinction between *RegularExpressionLiteral* and *DivPunctuator* (see [ECMA-262/5] section 7) must be respected.

CCInputElementState1 **::**

*RegularExpressionLiteralStringLiteralCCOnCCSet1CCIf1CCElif1CCElse1CCEnd1CC*Substitution*1CCStartMarkerCCEndMarkerCCMultiLineComment1CCSingleLinecomment1SourceCharacter*

CCSet1 **::**

```
@set [lookahead  IdentifierPart ]
/*@set [lookahead  IdentifierPart ]
//@set [lookahead  IdentifierPart ]
```

CCIf1 **::**

```
@if [lookahead  IdentifierPart ]
/*@if [lookahead  IdentifierPart ]
//@if [lookahead  IdentifierPart ]
```

CCElif1 **::**

```
@elif [lookahead  IdentifierPart ]
/*@elif [lookahead  IdentifierPart ]
//@elif [lookahead  IdentifierPart ]
```

CCElse1 **::**

```
@else [lookahead  IdentifierPart ]
/*@else [lookahead  IdentifierPart ]
//@else [lookahead  IdentifierPart ]
```

CCEnd1 **::**

```
@end [lookahead  IdentifierPart ]
/*@end [lookahead  IdentifierPart ]
//@end [lookahead  IdentifierPart ]
```

CCSubstitution1 **::**

```
@ CCSubIdentifier
/*@ CCSubIdentifier
//@ CCSubIdentifier
```

CCStartMarker **::**

```
/*@
//@
```

CCEndMarker **::**

```
@*/
```

CCMultiLineComment1 **::**

```
/* [lookahead ≠ @ ] MultiLineCommentCharsopt */
```

SingleLineComment1 **::**

```
// [lookahead ≠ @] SingleLineCommentCharsopt
```

CCSubIdentifer **::**

```
 [lookahead  CCKeyword ] IdentifierName
```

CCKeyword **::**

```
  cc_on setifelif
  elseend
```

Semantics

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

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

1. Append to the end of output, in left-to-right sequence, the Unicode characters from *source* that were recognized by the production. Remove the recognized characters from *source*.
2. Use *CCInputElementState1* to recognize the next input element from *source*.

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

1. Append a <SP> character to the end of output. Remove the recognized characters from *source*.
2. Use *CCInputElementState1* to recognize the next input element from *source*.

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

1. Append a <SP> character to the end of output. Remove the recognized characters from *source*.
2. Use *CCInputElementStateSetLHS* to recognize the next input element from *source*.

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

1. Append a <SP> character to the end of output. Remove the recognized characters from *source*.
2. Increment the value of *IfNestingLevel* by 1.
3. Use *CCInputElementStateIfPredicate* to recognize the next input element from *source*.

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

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

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

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

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

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

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

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

Syntax

NOTE:

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

CCInputElementStateSetLHS **::**

WhiteSpace$_{opt}$ **@** *IdentifierName WhiteSpace*$_{opt}$ **=** *CCExpression*

Semantics

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

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

1. Let *setName* be the string of characters recognized as the *IdentifierName* element of *CCSubstitution1*.
2. Let *value* be the result of evaluating *CCExpression*.
3. Create an association within *CCVariables* where the key is the string value of *setName* and where the value is *value*. If an association with that key already exists, replace it.
4. Remove the recognized characters from *source*.
5. Use *CCInputElementState1* to recognize the next input element from *source*.

Syntax

NOTE:

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

CCInputElementStateIfPredicate **::**

WhiteSpace$_{\text{opt}}$ **(** *CCExpression WhiteSpace*$_{\text{opt}}$ **)**

Semantics

If *CCInputElementStateIfPredicate* cannot be recognized, a **SyntaxError** exception is thrown.

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

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

Syntax

NOTE:

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

CCInputElementStateFalseThen **::**

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

Semantics

If *CCInputElementStateFalseThen* cannot be recognized, a **SyntaxError** exception is thrown.

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

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

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

1. Remove the recognized characters from *source*.
2. If *SkippedIfNestingLevel* > 0, then use *CCInputElementStateFalseThen* to recognize the next input element from *source*.
3. Otherwise, use *CCInputElementStateIfPredicate* to recognize the next input element from *source*.

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

1. Remove the recognized characters from *source*.
2. If *SkippedIfNestingLevel* > 0, then use *CCInputElementStateFalseThen* to recognize the next input element from *source*.
3. Otherwise, use *CCInputElementState1* to recognize the next input element from *source*.

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

1. Remove the recognized characters from *source*.
2. If *SkippedIfNestingLevel* is 0, then go to step 6.
3. Decrement the value of *SkippedIfNestingLevel* by 1.
4. Use *CCInputElementStateFalseThen* to recognize the next input element from *source*.
5. Return.
6. Decrement the value of *IfNestingLevel* by 1.
7. Use *CCInputElementState1* to recognize the next input element from *source*.

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

1. Remove the recognized characters from *source*.
2. Use *CCInputElementStateFalseThen* to recognize the next input element from *source*.

Syntax

NOTE:

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

CCInputElementStateFalseIfTail **::**

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

Semantics

If *CCInputElementStateFalseIfTail* cannot be recognized, a **SyntaxError** exception is thrown.

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

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

2. Remove the recognized characters from *source*.

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

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

1. Remove the recognized characters from *source*.

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

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

1. Remove the recognized characters from *source*.

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

3. Decrement the value of Skipped*IfNestingLevel* by 1.

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

5. Return.

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

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

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

1. Remove the recognized characters from *source*.

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

Syntax

CCExpression ::

```
                CCLogicalANDExpression                CCExpression WhiteSpaceopt  ||
CCLogicalANDExpression
```

CCLogicalANDExpression ::

```
CCBitwiseORExpressionCCcLogicalANDExpression  WhiteSpaceopt
&&  CCBitwiseORExpression
```

CCBitwiseORExpression **::**

```
CCBitwiseXORExpressionCCBitwiseORExpression  WhiteSpaceopt  | CCBitwiseXORExpression
```

CCBitwiseXORExpression **::**

```
CCBitwiseANDExpressionCCBitwiseXORExpression  WhiteSpaceopt ^ CCBitwiseANDExpression
```

CCBitwiseANDExpression **::**

```
CCEqualityExpressionCCBitwiseANDExpression  WhiteSpaceopt & CCEqualityExpression
```

CCEqualityExpression **::**

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

CCRelationalExpression **::**

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

CCShiftExpression **::**

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

CCAdditiveExpression **::**

```
CCMultiplicativeExpressionCCAdditiveExpression  WhiteSpaceopt +
CCMultiplicativeExpressionCCAdditiveExpression  WhiteSpaceopt - CCMultiplicativeExpression
```

CCMultiplicativeExpression **::**

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

UnaryExpression **::**

```
CCPrimaryExpressionWhiteSpaceopt + CCUnaryExpressionWhiteSpaceopt -
CCUnaryExpressionWhiteSpaceopt ~ CCUnaryExpressionWhiteSpaceopt! CCUnaryExpression
```

CCPrimaryExpression **::**

```
  CCVariableCCLiteralWhiteSpaceopt ( Expression )
```

CCLiteral **::**

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

CCVariable **::**

```
  WhiteSpaceopt @ IdentifierName
```

Semantics

Unless otherwise specified in this section, the productions of *CCExpression* are evaluated using the same semantic rules as the analogous productions of the ECMAScript syntactic grammar for Expression in [ECMA-262/5] section 11. However, only values of types **Number** and **Boolean** can occur during the evaluation of *CCExpression* productions, so any semantic steps that are relative to other types of values are not relevant.

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

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

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

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

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

2.1.2 Extensions to Numeric Literals

Internet Explorer ECMAScript supports the Numeric Literal extensions that are defined by [ECMA-262/5] Annex B, section B.1.1.

2.1.3 Extensions to String Literals

Internet Explorer ECMAScript supports the String Literal extensions that are defined by [ECMA-262/5] Annex B, section B.1.2.

In addition, the production *EscapeSequence* is extended to include the characters 8 and 9 as right-hand-side alternatives, as follows:

EscapeSequence **::**

```
CharacterEscapeSequence
OctalEscapeSequence
HexEscapeSequence
UnicodeEscapeSequence
8
9
```

The character values (CV) are defined as follows:

1. The CV of *EscapeSequence* **::** `8` is a character 8 (Unicode value 0038).
2. The CV of *EscapeSequence* **::** `9` is a character 9 (Unicode value 0039).

2.2 Extensions to Types

The following section defines an Internet Explorer ECMAScript extension to [ECMA-262/5] types.

2.2.1 SafeArray Type

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

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

2.2.2 VarDate Type

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

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

2.3 Extensions to Type Conversion and Testing

The following extensions to [ECMA-262/5] are necessary to support the **SafeArray** and **VarDate** extended types.

Conversion operation	*Argument type*	*Operation*
ToPrimitive	**SafeArray**	Returns the input argument (no conversion is applied).
ToPrimitive	**VarDate**	Returns the input argument (no conversion is applied).
ToBoolean	**SafeArray**	Returns a value of **false**.
ToBoolean	**VarDate**	Returns a value of **false**.
ToNumber	**SafeArray**	Throws a **TypeError** exception.
ToNumber	**VarDate**	Returns the **Number** value that represents the internal numerical value of the **VT_Date** value.
ToString	**SafeArray**	Applies the following steps:

Conversion operation	*Argument type*	*Operation*
		1. Let *objValue* be **ToObject**(input *argument*). 2. Returns the value of **ToString**(*objValue*).
ToString	**VarDate**	Returns a **String** value that contains a representation of the **VarDate** value in the same representational format as **Date**.**prototype**.**toString**. For more information, see[ECMA-262/5], Section 15.9.5.2.
ToObject	**SafeArray**	Creates a new **VBArray** object in the same manner as the following ECMAScript expression: `new VBArray(argument)` In this case, *argument* is a value of type **SafeArray**.
ToObject	**VarDate**	Throws a **TypeError** exception.
CheckObjectCoercible	**SafeArray**	Returns with no return value.
CheckObjectCoercible	**VarDate**	Throws a **TypeError** exception.

2.4 Extensions to Executable Code and Execution Contexts

The following section defines Internet Explorer ECMAScript extensions to [ECMA-262/5] executable code and execution contexts.

The extensions are as follows:

- Extensions to Declaration Binding Instantiation

2.4.1 Extensions to Declaration Binding Instantiation

Internet Explorer ECMAScript allows a *FunctionDeclaration* language syntactic element to appear anywhere that a *Statement* can appear. *FunctionDeclaration* items are processed during step 5 of the Declaration Binding Instantiation algorithm (which is defined by [ECMA-262/5], section 10.5). However, a *FunctionDeclaration* item that defines an event handler is excluded from the processing of step 5. Such a *FunctionDeclaration* item is evaluated when an ECMAScript *SourceElement* production is evaluated.

2.5 Extensions to Expressions

The following section defines Internet Explorer ECMAScript extensions to [ECMA-262/5] expressions.

2.5.1 Extensions to typeof Operator

Internet Explorer ECMAScript adds the following **typeof** operator results to Table 20 in [ECMA-262/5], section 11.4.3.

*Typeof **val***	*Result*
SafeArray	`"unknown"`
VarDate	`"date"`

2.6 Extensions to Statements

The following section defines an Internet Explorer ECMAScript extension to [ECMA-262/5] statements.

2.6.1 Extension Grammar Production for Statement

Internet Explorer ECMAScript adds a *FunctionDeclaration* language syntactic element as an additional alternative to the *Statement* grammar production of [ECMA-262/5], section 12:

Syntax Extension

Statement**:**

FunctionDeclaration

A *FunctionDeclaration* element can occur in any context where a *Statement* production is required. The semantics of such declarations are specified in section 2.4.1 of this document.

2.7 Extensions to Function Definition

The following section defines an Internet Explorer ECMAScript extension to [ECMA-262/5] functions.

2.7.1 Function Definition Used As a Statement

Semantic Extensions

Internet Explorer ECMAScript allows a *FunctionDeclaration* element to be evaluated as a *Statement* production, as follows:

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

When this production is evaluated, the following step is performed:

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

2.7.2 Event Handler Function Definitions

Internet Explorer ECMAScript adds an additional alternative to the *FunctionDeclaration* grammar production of [ECMA-262/5], section 13, as follows:

Syntax Extension

FunctionDeclaration **:**

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

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

ObjectPath **:**

Identifier

ObjectPath NameQualifier Identifier

NameQualifier **:** .

Semantic Extensions

Internet Explorer ECMAScript allows a *FunctionDeclaration* element to be evaluated as a *Statement* production, as follows:

FunctionDeclaration **: function** *ObjectPath* **::** *Identifier* **(** *FormalParameterList*$_{opt}$ **) {** *FunctionBody* **}**

When this production is evaluated, the following steps are performed:

1. Let *p* be the result of the evaluation of *ObjectPath*.
2. Let *o* be **ToObject**(**GetValue**(*p*)).
3. If *o* is not a host object that supports event attachment, throw a **TypeError** exception.
4. Let *eventName* be a string that contains the text of *Identifier*.
5. Let *h* be the result of creating a new **Function** object, as specified in [ECMA-262/5], section 13.2, with the parameters specified by *FormalParameterList*$_{opt}$ and the body specified by *FunctionBody*.
    1. Pass in the **VariableEnvironment** component of the running execution context as the *Scope*.
    2. Pass in a value of **true** as the *Strict* flag if the *FunctionDeclaration* element is contained in strict code or if its *FunctionBody* element is strict code.
6. Perform event handler attachment of *h* to *o* by using *eventName* as the event name.
7. Return (**normal**, **empty**, **empty**).

An event handler function *h* is attached to a host object *o* with *eventName* *n* as follows:

1. If *o* implements the IBindEventHandler COM interface (http://msdn.microsoft.com/en-us/library/56zc7scb(VS.85).aspx), perform the following actions:
    1. Call the **BindHandler** COM method of *o*, passing arguments *n* and the function entry point *h*. This call hooks up the direct event.
    2. Return.
2. If *o* does not implement the **IBindEventHandler** COM interface, retain the information (*o*, *n*, and *h*), and defer the event binding until the script engine is placed into "connected" mode, as defined by the SCRIPTSTATE_CONNECTED constant value of the Microsoft Windows Script Technologies SCRIPTSTATE enumeration (http://msdn.microsoft.com/en-us/library/f7z7cxxa(VS.85).aspx). When the script engine is placed into the connected mode, the retained information is used to bind the event with an event sinking process. The event binding is performed immediately if the script is already in connected mode.
3. Return.

The **IConnectionPointContainer** COM interface (http://msdn.microsoft.com/en-us/library/ms683857(VS.85).aspx) is used to perform the event binding in step 2, regardless of whether the binding is performed immediately or is deferred.

2.8 Extensions to Native ECMAScript Objects

Internet Explorer ECMAScript defines extensions to the native ECMAScript objects of [ECMA-262/5]. These extensions are described in the following sections.

2.8.1 Function Properties of the Global Object

Internet Explorer ECMAScript defines additional properties of the Global object of [ECMA-262/5]. These properties are described in the following sections.

2.8.1.1 ScriptEngine

When the ScriptEngine function is called, it returns a string value that specifies the implementation-defined name of the ECMAScript implementation that is executing the call. The Internet Explorer ECMAScript implementations within Internet Explorer 9 always return the string "JScript".

2.8.1.2 ScriptEngineBuildVersion

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

2.8.1.3 ScriptEngineMajorVersion

When the **ScriptEngineMajorVersion** function is called, it returns a value that identifies the major revision level of the implementation, not the revision level of the ECMAScript or JavaScript language specification that is currently supported by the implementation.

An implementation of Internet Explorer ECMAScript that supports distinct document modes (that separately implement other versions of the language, such as JScript 5.7 and JScript 5.8 functionality) can return a single value that does not vary among modes. The return value cannot be used as a reliable indicator of the availability or lack of availability of specific language features.

The ECMAScript implementations within Internet Explorer 9 always return a value of `9`, even when Internet Explorer 9 is operating in Quirks, IE7, or IE8 document modes.

2.8.1.4 ScriptEngineMinorVersion

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

The ECMAScript implementation within Microsoft Internet Explorer 9 always returns a value of `0`, even when Internet Explorer 9 is operating in Quirks, IE7, or IE8 document modes.

2.8.1.5 CollectGarbage

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

2.8.2 Constructor Properties of the Global Object

Internet Explorer ECMAScript defines the following additional constructor properties of the Global object:

1. Debug
2. Enumerator
3. VBArray
4. ActiveXObject

2.8.3 Properties of Function Instances

Internet Explorer ECMAScript defines additional properties of Function instances of [ECMA-262/5]. These properties are described in the following sections.

2.8.3.1 The arguments Property

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

2.8.3.2 The caller Property

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

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

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

1. If *P* is the string 'arguments', take the following steps:
    1. If an active execution context for *F* does not exist, go to step 3.
    2. Let *X* be the most recently created active execution context for *F*.
    3. If *X* is marked as having a partially accessible arguments object, let *A* be the original arguments object for *X*; otherwise, let *A* be the value of the property named 'arguments' of the variable object of *X*.

       Note: JScript 5.x under Internet Explorer 9 (in all document modes) marks the current execution context as having a partially accessible arguments object when the function's *FormalParameterList* contains the name 'arguments' or the function's *FunctionBody* contains a direct reference to the function's original arguments object or the function's *FunctionBody* contains a direct call to **eval**.
    4. Return *A*.
2. If *P* is the string 'caller', take the following steps:
    1. Let *X* be the most recently created active execution context for *F*.
    2. If *X* does not have an execution context to which it could normally exit, return **null**.
    3. Let *R* be the execution context which would become the current execution context if *X* exited normally (not via an exception).

   4. If *R* is an execution context for a built-in function or a host object function, return **null**.

   5. If *R* is an execution context for global code or for eval code, return **null**.

   6. *R* must be an execution context for function code, so let *rf* be the function object that contains the call that caused *R* to be created.

   7. If *rf* is a strict mode **Function** object, throw a **TypeError** exception.

   8. Return *rf*.

3. Return the result of calling the default **[[Get]]** method ([ECMA-262/5] section 8.12.3), passing *P* as the argument.

2.8.4 String.prototype HTML Wrapper Properties

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

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

1. Append the character "<" to the characters of *tag*.

2. If *attribute* is not present, go to Step 7.

3. Append to *Result(1)* a single-space character followed by the characters of *attribute*.

4. Append to *Result(3)* the characters "=" and """.

5. Append to *Result(4)* the characters of the string returned by **ToString**(*data*).

6. Append to *Result(5)* the character """.

7. If *attribute* is present, use Result(6); otherwise, use Result(1).

8. Append to *Result(7)* the character ">".

9. Append to *Result(8)* the characters of the string returned by **ToString**(*body*).

10. Append to *Result(9)* the characters "<" and "/".

11. Append to *Result(10)* the characters of *tag*.

12. Append to *Result(11)* the character ">".

13. Return the string value of the characters from *Result(12)*.

2.8.4.1 String.prototype.anchor(name)

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

2.8.4.2 String.prototype.big( )

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

2.8.4.3 String.prototype.blink( )

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

2.8.4.4 String.prototype.bold( )

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

2.8.4.5 String.prototype.fixed( )

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

2.8.4.6 String.prototype.fontcolor(color)

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

2.8.4.7 String.prototype.fontsize(size)

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

2.8.4.8 String.prototype.italics( )

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

2.8.4.9 String.prototype.link(url)

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

2.8.4.10 String.prototype.small( )

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

2.8.4.11 String.prototype.strike( )

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

2.8.4.12 String.prototype.sub( )

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

2.8.4.13 String.prototype.sup( )

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

2.8.5 Properties of the Date Prototype Object

Internet Explorer ECMAScript defines an additional method of the **Date** prototype object of [ECMA-262/5]. This method is described in the following section.

2.8.5.1 Date.prototype.getVarDate ( )

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

1. Let *t* be the time value.

2. It the value of *t* is "NaN", return a **VarDate** value for which the value of **ToNumber** is "NaN".

3. Otherwise, return a **VarDate** value that corresponds to the time value *t*.

2.8.6 Properties of the RegExp Constructor

Internet Explorer ECMAScript defines additional properties of the RegExp constructor of [ECMA-262/5]. These properties are described in the following sections.

2.8.6.1 RegExp.input

The initial value of **RegExp**.**input** is the empty string. This property shall have the attributes { **[[Enumerable]]**: **false**, **[[Configurable]]**: **false**, **[[Writable]]**: **true** }. The value of this property may be modified by calls to **RegExp.prototype.exec**. The properties **RegExp**.**input** and **RegExp**.**$_** always have the same value. When one is set to some value, the other is automatically also set to that same value. Unlike most other **RegExp** constructor properties, this property is writable.

2.8.6.2 RegExp.lastIndex

The initial value of **RegExp**.**lastIndex** is the number `-1`. This property shall have the attributes { **DontEnum**, **DontDelete**, **ReadOnly** }. Even though this property is {**[[Writable]]**: **false**}, its value may be modified by calls to **RegExp**.**prototype**.**exec**.

2.8.6.3 RegExp.lastMatch

The initial value of **RegExp**.**lastMatch** is the empty string. This property shall have the attributes { **[[Enumerable]]**: **false**, **[[Configurable]]**: **false**, **[[Writable]]**: **false** }. Even though this property is {**[[Writable]]**: **false**}, its value may be modified by calls to **RegExp**.**prototype**.**exec**.

2.8.6.4 RegExp.lastParen

The initial value of **RegExp**.**lastParen** is the empty string. This property shall have the attributes { **[[Enumerable]]**: **false**, **[[Configurable]]**: **false**, **[[Writable]]**: **false** }. Even though this property is {**[[Writable]]**: **false**}, its value may be modified by calls to **RegExp**.**prototype**.**exec**.

2.8.6.5 RegExp.leftContext

The initial value of **RegExp**.**leftContext** is the empty string. This property shall have the attributes { **[[Enumerable]]**: **false**, **[[Configurable]]**: **false**, **[[Writable]]**: **false** }. Even though this property is {**[[Writable]]**: **false**}, its value may be modified by calls to **RegExp**.**prototype**.**exec**.

2.8.6.6 RegExp.rightContext

The initial value of **RegExp**.**rightContext** is the empty string. This property shall have the attributes { **[[Enumerable]]**: **false**, **[[Configurable]]**: **false**, **[[Writable]]**: **false** }. Even though this property is {**[[Writable]]: false**}, its value may be modified by calls to **RegExp**.**prototype**.**exec**.

2.8.6.7 RegExp.$1 - RegExp.$9

The initial value of **RegExp**.**rightContext** is the empty string. This property shall have the attributes { **[[Enumerable]]**: **false**, **[[Configurable]]**: **false**, **[[Writable]]**: **false** }. Even though these are {**[[Writable]]: false**} properties, their values may be modified by calls to **RegExp**.**prototype**.**exec**.

2.8.6.8 RegExp.$_

The initial value of each of the properties **RegExp**.**$1**, **RegExp**.**$2**, **RegExp**.**$3**, **RegExp**.**$4**, **RegExp**.**$5**, **RegExp**.**$6**, **RegExp**.**$7**, **RegExp**.**$8**, and **RegExp**.**$9** is the empty string. These properties shall have the attributes { **[[Enumerable]]**: **false**, **[[Configurable]]**: **false**, **[[Writable]]**: **false** }. The value of this property may be modified by calls to **RegExp**.**prototype**.**exec**. The properties **RegExp**.**input** and **RegExp**.**$_** always have the same value. When one of these properties is set to some value, the other is automatically also set to that same value. Unlike most other **RegExp** constructor properties, this property is writable.

2.8.6.9 RegExp['$&']

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

2.8.6.10 RegExp['$+']

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

2.8.6.11 RegExp["$`"]

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

2.8.6.12 RegExp["$'"]

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

2.8.7 Properties of the RegExp Prototype Object

Internet Explorer ECMAScript defines additional properties of the **RegExp** prototype object of [ECMA-262/5] (see section 15.10.6). These properties are described in the following sections.

2.8.7.1 RegExp.prototype.compile(pattern, flags)

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

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

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

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

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

If the characters in *P* do not have the form *Pattern*, throw a **SyntaxError** exception. Otherwise, let the newly constructed object have a **[[Match]]** property obtained by evaluating ("compiling") *Pattern*.

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

1. When pattern is an object *R* that has a **[[Class]]** property of "RegExp", this **RegExp** object is set to the same string value as the value of the source property of pattern. Otherwise, the source property of this **RegExp** object is set to *P*.
2. The **lastIndex** property of this **RegExp** object is set to 0.
3. The **options** property of this **RegExp** object is set as described in section 2.8.8.1 of this document.
4. This **RegExp** object is optimized using the assumption that it will be executed multiple times.

2.8.8 Properties of RegExp Instances

Internet Explorer ECMAScript defines an additional property of the **RegExp** instances of [ECMA-262/5]. This property is described in the following section.

2.8.8.1 options

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

2.8.9 The Error Constructor

Internet Explorer ECMAScript defines additional behaviors of the **Error** constructor of [ECMA-262/5].

These behaviors are described in the following sections:

- new Error ()
- new Error (number, message)

2.8.9.1 new Error ()

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

2.8.9.2 new Error(number, message)

When the **Error** constructor is called with two or more arguments, 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** (see [ECMA-262/5] section 15.11.3.1).
2. The **[[Class]]** property of the newly constructed **Error** object is set to "Error".
3. Let *num* be **ToNumber**(*number*).
4. Let *msg* be **ToString**(*message*).
5. The **description** property of the newly constructed object is set to *msg*.
6. The **message** property of the newly constructed object is set to *msg*.
7. The **name** property of the newly constructed object is set to "Error".
8. The **number** property of the newly constructed object is set to *num*.
9. Return the newly constructed object.

2.8.10 Properties of Error Instances

Internet Explorer ECMAScript defines additional error instances inherited from the **[[Prototype]]** object of [ECMA-262/5]. These error instances are described in the following sections.

2.8.10.1 description

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

2.8.10.2 number

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

2.8.11 Properties of NativeError Instances

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

2.8.11.1 description

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

2.8.11.2 number

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

2.8.12 The Debug Object

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

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

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

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

2.8.12.1 Function Properties of the Debug Object

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

2.8.12.1.1 write ([item1 [, item2 [, …]]])

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

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

2.8.12.1.2 writeln ([item1 [, item2 [, …]]]))

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

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

2.8.13 Enumerator Objects

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

For such objects, the order of enumeration is the same as occurs for the **for-in** statement (see [ECMA-262/5] section 12.6.4).

2.8.13.1 The Enumerator Constructor Called as a Function

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

2.8.13.2 The Enumerator Constructor

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

2.8.13.2.1 new Enumerator ([collection])

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

1. If *collection* is not present, let *collection* be **undefined**, and then go to step 6.
2. If *collection* is an **Array** instance, go to step 5.
3. If *collection* is a host object that supports an implementation-dependent enumeration protocol, go to step 5.
4. Throw a **TypeError** exception.
5. The **[[EnumerationState]]** property of the newly created object is set to a state indicating that the enumeration is at the first item of the enumeration of *collection*. If *collection* has no enumerable items, the state will indicate that the end of the enumeration has been reached.
6. The **[[Collection]]** property of the newly created object is set to *collection*.
7. The **[[Prototype]]** property of the newly constructed object is set to the original **Error** prototype object, the one that is the initial value of **Enumerator**.**prototype** (see section 2.8.13.3.1 of this document).
8. The **[[Class]]** property of the newly constructed **Enumerator** object is set to "Object".
9. Return the newly constructed object.

2.8.13.3 Properties of the Enumerator Constructor

The value of the internal **[[Prototype]]** property of the **Enumerator** constructor is the **Function** prototype object (see [ECMA-262/5] section 15.3.4).

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

2.8.13.3.1 Enumerator.prototype

The initial value of **Enumerator**.**prototype** is the **Enumerator** prototype object (see section 2.8.13.4 of this document).

This property has the attributes { **[[Enumerable]]**:**false**, **[[Configurable]]**:**false**, **[[Writable]]**:**false** }.

2.8.13.4 Properties of the Enumerator Prototype Object

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

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

2.8.13.4.1 Enumerator.prototype.constructor

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

2.8.13.4.2 Enumerator.prototype.atEnd ( )

1. If the **this** object is not an **Enumerator** object, throw a **TypeError** exception.
2. Let *collection* be the value of the **this** object's **[[Collection]]** property.

3. If *collection* is undefined, return **true**.
4. Let *state* be the value of the this object's **[[EnumerationState]]** property.
5. If *state* indicates that the end of the enumeration has been reached, return **true**.
6. Return **false**.

2.8.13.4.3 Enumerator.prototype.item ( )

1. If the **this** object is not an **Enumerator** object, throw a **TypeError** exception.
2. Let *collection* be the value of the **this** object's **[[Collection]]** property.
3. If *collection* is undefined, return **undefined**.
4. Let *state* be the value of the this object's **[[EnumerationState]]** property.
5. If *state* indicates that the end of the enumeration has been reached, return **undefined**.
6. Return the current enumeration item as indicated by **state**.

2.8.13.4.4 Enumerator.prototype.moveFirst ( )

1. If the **this** object is not an **Enumerator** object, throw a **TypeError** exception.
2. Let *collection* be the value of the **this** object's **[[Collection]]** property.
3. If *collection* is undefined, return **undefined**.
4. Modify the **[[EnumerationState]]** property of the this object to a state indicating that the current enumeration of *collection* is now positioned at the original first item of the enumeration. If the current **[[EnumerationState]]** property indicates that *collection* has no enumerable items, the new *state* will indicate that the end of the enumeration has been reached.
5. Return **undefined**.

2.8.13.4.5 Enumerator.prototype.moveNext ( )

1. If the **this** object is not an **Enumerator** object, throw a **TypeError** exception.
2. Let *collection* be the value of the **this** object's **[[Collection]]** property.
3. If *collection* is undefined, return **undefined**.
4. Let *state* be the value of the this object's **[[EnumerationState]]** property.
5. If *state* indicates that the end of the enumeration has been reached, return **undefined**.
6. Modify *state* to a state indicating that the current enumeration of *collection* is now positioned at the next item beyond the current item of the enumeration. The new *state* may indicate that the end of the enumeration has been reached.
7. Update the **[[EnumerationState]]** property of the this object to *state*.
8. Return **undefined**.

2.8.13.5 Properties of Enumerator Instances

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

2.8.14 VBArray Objects

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

For such objects, the order of enumeration is the same as the for-in statement (see [ECMA-262/5] section 12.6.4).

2.8.14.1 The VBArray Constructor Called as a Function

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

2.8.14.1.1 VBArray (value)

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

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

2. Throw a **TypeError** exception.

2.8.14.2 The VBArray Constructor

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

2.8.14.2.1 new VBArray (value)

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

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

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

3. The **[[Prototype]]** property of the newly constructed object is set to the initial value of the **VBArray prototype** object (see section 2.8.14.3.1 of this document).

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

5. Return the newly constructed object.

2.8.14.3 Properties of the VBArray Constructor

The value of the internal **[[Prototype]]** property of the **VBArray constructo**r is the **Function** prototype object (see [ECMA-262/5] section 15.3.4).

The value of the **length** property is 1. In addition, the **VBArray constructor** has the **VBArray.prototype** property (see section 2.8.14.3.1 of this document).

2.8.14.3.1 VBArray.prototype

The initial value of **VBArray.prototype** is the **VBArray prototype object** (see section 2.8.14.4 of this document).

This property has the attributes { **[[Enumerable]]**: **false**, **[[Configurable]]**: **false**, **[[Writable]]**: **true** }.

2.8.14.4 Properties of the VBArray Prototype Object

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

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

2.8.14.4.1 VBArray.prototype.constructor

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

2.8.14.4.2 VBArray.prototype.dimensions ( )

1. Call **ToObject**, passing the **this** value as the argument.
2. If *Result(1)* is not a **VBArray** instance, throw a **TypeError** exception.
3. Get the value of the **[[SArray]]** property of *Result(1)*.
4. Return the **Number** that is the number of dimensions of the **COM SAFEARRAY** referenced by Result(3).

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

1. Call **ToObject**, passing the **this** value as the argument.
2. If *Result(1)* is not a **VBArray** instance, throw a **TypeError** exception.
3. Get the value of the **[[SArray]]** property of *Result(1)*.
4. If no arguments were passed to this call, or if the number of arguments passed is greater than *Result(3)*, throw a **RangeError** exception.
5. For each argument *dim1* through *dimN*, let *IdimX* be **ToInteger**(*dimX*) where *X* is the numeric suffix of the argument name.
6. For each of *Idim1* through *IdimN*, if *IdimX* is less than the **lower** bound of dimension *X* of the **COM SAFEARRAY** referenced by *Result(3)*, or if *IdimX* is greater than the **upper** bound of dimension *X*, throw a **RangeError** exception.
7. Return the value of the element identified by array indices *Idim1* through *IdimN* in the **COM SAFEARRAY** referenced by *Result(3)*.

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

2.8.14.4.4 VBArray.prototype.lbound ([dimension])

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

2. If *Result(1)* is not a **VBArray** instance, throw a **TypeError** exception.
3. Get the value of the **[[SArray]]** property of *Result(1)*.
4. If *dimension* is not defined, use a value of `1`; otherwise, use **ToInteger**(*dimension*).
5. Get the **Number** that is the number of dimensions of the **COM SAFEARRAY** referenced by *Result(3)*.
6. If *Result(4)* is less than `1` or greater than *Result(5)*, throw a **RangeError** exception.
7. Return the **Number** that is the lower bound of dimension number *Result(4)* of the **COM SAFEARRAY** referenced by *Result(3*).

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

2.8.14.4.5 VBArray.prototype.toArray ( )

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

1. Call **ToObject**, passing the **this** value as the argument.
2. If Result(1) is not a **VBArray** instance, throw a **TypeError** exception.
3. Get the value of the **[[SArray]]** property of *Result(1)*.
4. Let *SA* be the **COM SAFEARRAY** referenced by *Result(3)*.
5. Let *dim* be the number of dimensions of *SA*.
6. If *dim* is zero, return a new **Array** object that is created as if by evaluating the expression new **Array**(0) using the original **Array** constructor object.
7. Let *size* be the total number of array elements of *SA*.
8. Let *A* be a new **Array** object that is created as if by evaluating the expression new **Array**(*size*) using the original **Array** constructor object.
9. Access the elements of *SA* in row-major order, and store the elements in the array-indexed properties for *A* starting with property 0.
10. Return *A*.

2.8.14.4.6 VBArray.prototype.ubound ([dimension])

1. Call **ToObject**, passing the **this** value as the argument.
2. If *Result(1)* is not a **VBArray** instance, throw a **TypeError** exception.
3. Get the value of the **[[SArray]]** property of *Result(1)*.
4. If *dimension* is not defined, use a value of `1`; otherwise, use **ToInteger**(*dimension*).
5. Get the **Number** that is the number of dimensions of the **COM SAFEARRAY** referenced by *Result(3)*.

6. If *Result(4)* is less than 1 or greater than *Result(5)*, throw a **RangeError** exception.
7. Return the **Number** that is the upper bound of dimension number *Result(4)* of the **COM SAFEARRAY** referenced by *Result(3)*.

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

2.8.14.4.7 VBArray.prototype.valueOf ( )

1. Call **ToObject**, passing the **this** value as the argument.
2. If *Result(1)* is not a **VBArray** instance, throw a **TypeError** exception.
3. Get the value of the **[[SArray]]** property of *Result(1)*.
4. Return *Result(3)*.

2.8.14.5 Properties of VBArray Instances

A **VBArray** instance inherits properties from the **[[Prototype]]** object, as specified in **VBArray.prototype.valueOf ( )** (see section 2.8.14.4.7 of this document). In addition, **VBArray** instances have an internal **[[SArray]]** property with a value that is the **SafeArray** from which the instance was constructed.

2.8.15 ActiveXObject Objects

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

2.8.15.1 The ActiveXObject Constructor Called as a Function

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

2.8.15.1.1 ActiveXObject (name [, location]))

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

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

2.8.15.2 The ActiveXObject Constructor

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

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

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

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

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

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

2.8.15.3 Properties of the ActiveXObject Constructor

The value of the internal **[[Prototype]]** property of the **ActiveXObject** constructor is the **Function** prototype object (see [ECMA-262/5] section 15.3.4).

The value of the length property is `1`. In addition, the **ActiveXObject** constructor has the **ActiveXObject.prototype** property (see section 2.8.15.3.1 of this document).

2.8.15.3.1 ActiveXObject.prototype

The initial value of **ActiveXObject.prototype** is the **ActiveXObject** prototype object (see section section 2.8.15.4 of this document).

This property has the attributes { **[[Enumerable]]**: **false**, **[[Configurable]]**: **false**, **[[Writable]]**: **false** }.

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

2.8.15.4 Properties of the ActiveXObject Prototype Object

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

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

2.8.15.4.1 ActiveXObject.prototype.constructor

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

2.8.15.5 Properties of ActiveXObject Instances

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

2.9 Extensions to ECMAScript 5.1

This section describes extensions to [ECMA-262/51] that are not available in versions prior to Windows® Internet Explorer® 10.

2.9.1 Typed Arrays

Typed arrays provide access to raw binary data and enables efficient byte-level programming ability to JavaScript developers. The functionality is implemented by the following three objects.

- **ArrayBuffer**: The ArrayBuffer object provides the ability to create and work with an opaque buffer of native memory.
- **TypeArray**: Each of the TypeArray objects provides a view over an ArrayBuffer based on the element Type, allowing typed access to the contents of the native buffer.
- **DataView**: The DataView object provides unstructured access to the contents of an ArrayBuffer, reading and writing basic data types and fixed offsets in the buffer.

2.9.1.1 ArrayBuffer Objects

This section describes ArrayBuffer Objects.

2.9.1.1.1 The ArrayBuffer constructor called as a function

When ArrayBuffer is called as a function rather than as a constructor, it creates and initialises a new ArrayBuffer object. Thus the function call ArrayBuffer(…) is equivalent to the object creation expression new ArrayBuffer (…) with the same arguments.

2.9.1.1.2 The ArrayBuffer constructor

When ArrayBuffer is called as part of a new expression, it is a constructor: it initialises the newly created object.

2.9.1.1.2.1 New Array (len)

The [[Prototype]] internal property of the newly constructed object is set to the original ArrayBuffer prototype object, the one that is the initial value of `ArrayBuffer.prototype` (16.1.3.1). The [[Class]] internal property of the newly constructed object is set to "`ArrayBuffer`". The [[Extensible]] internal property of the newly constructed object is set to `true`.

The length property of the newly constructed object is set to ToUInt32(len).

A fresh native buffer nativeBuffer of `length` bytes is allocated. The contents of this native buffer are zero initialized. If the requested number of bytes could not be allocated, a RangeError is raised. The [[NativeBuffer]] internal property of the newly constructed object is set to nativeBuffer.

2.9.1.1.3 Properties of the ArrayBuffer constructor

The value of the [[Prototype]] internal property of the ArrayBuffer constructor is the Function prototype object (15.3.4).

Besides the internal properties and the length property (whose value is 1), the ArrayBuffer constructor has the following properties:

2.9.1.1.3.1 ArrayBuffer.Prototype

The initial value of ArrayBuffer.prototype is the ArrayBuffer prototype object (16.1.4).

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

2.9.1.1.4 Properties of the ArrayBuffer Prototype Object

The value of the [[Prototype]] internal property of the Array prototype object is the standard built-in Object prototype object (15.2.4). The [[Class]] internal property of the newly constructed object is set to "Object". The [[Extensible]] internal property of the newly constructed object is set to true.

2.9.1.1.4.1 ArrayBuffer.prototype.constructor

The initial value of ArrayBuffer.prototype.constructor is the standard built-in ArrayBuffer constructor.

2.9.1.1.5 Properties of ArrayBuffer Instances

ArrayBuffer instances inherit properties from the ArrayBuffer prototype object and their [[Class]] internal property value is "ArrayBuffer". ArrayBuffer instances also have the following properties.

2.9.1.1.5.1 byteLength

The byteLength property of this ArrayBuffer object is a data property whose value is the length of the ArrayBuffer in bytes, as fixed at construction time.

The length property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [Configurable]]: false }.

2.9.1.2 TypeArray Objects

For each *Type* in the following table, a separate *Type*Array constructor object, with corresponding prototype and instances as described below is available.

Type	**Array Name**	**Size**	**Description**	**Equivalent C Type**
`Int8`	`Int8Array`	1	8-bit 2's complement signed integer	`signed char`
`Uint8`	`Uint8Array`	1	8-bit unsigned integer	`unsigned char`
`Int16`	`Int16Array`	2	16-bit 2's complement signed integer	`Short`
Uint16	`Uint16Array`	2	16-bit unsigned integer	`unsigned short`
`Int32`	`Int32Array`	4	32-bit 2's complement signed integer	`Int`
`Uint32`	`Uint32Array`	4	32-bit unsigned integer	`unsigned int`
`Float32`	`Float32Array`	4	32-bit IEEE floating point	`Float`
`Float64`	`Float64Array`	8	64-bit IEEE floating point	`Double`

2.9.1.2.1 The TypeArray Constructor Called as a Function

When *Type*Array is called as a function rather than as a constructor, it creates and initialises a new *Type*Array object. Thus the function call *Type*Array(…) is equivalent to the object creation expression new *Type*Array (…) with the same arguments.

2.9.1.2.2 The TypeArray Constructor

When *Type*Array is called as part of a new expression, it is a constructor: it initialises the newly created object.

2.9.1.2.2.1 New TypeArray (arg0 [, arg1, [, arg2])

The [[Prototype]] internal property of the newly constructed object is set to the original *Type*Array prototype object, the one that is the initial value of *Type*Array.prototype (16.2.3.1). The [[Class]] internal property of the newly constructed object is set to "*Type*Array". The [[Extensible]] internal property of the newly constructed object is set to true.

The remaining properties of the newly constructed object are set as follows:

1. If the argument arg0 is a Number:
    1. The length property of the newly constructed object is set to ToUInt32(arg0)
    2. The byteLength property of the newly constructed object is set to length multiplied by the size in bytes of *Type*.
    3. Let arrayBuffer be an object constructed as if by a call to the built-in ArrayBuffer constructor, as "new ArrayBuffer(byteLength)".
    4. The buffer property of the newly constructed object is set to arrayBuffer.
    5. The byteOffset property of the newly constructed object is set to 0.

```
2. Otherwise if the argument arg0 is an Object:
    1. Let O be the result of calling ToObject(arg0).
    2. Let class be the value of the [[Class]] internal property of O.
    3. If class is "ArrayBuffer":
        1. Let byteOffset be the result of calling ToUInt32 on arg1, if provided, or else 0.
        2. If byteOffset is not an integer multiple of the size in byte of Type, raise a RangeError exception.
        3. Let bufferLength be the result of calling [[Get]] on O with property name "byteLength".
        4. Let byteLength be the result of calling ToUInt32 on arg2, if provided, or else bufferLength – byteOffset.
        5. If byteOffset + byteLength is greater than bufferLength, raise a RangeError exception.
        6. Let length be the result of dividing byteLength by the size in bytes of Type.
        7. If ToUInt32(length) !== length, raise a RangeError exception.
        8. The length property of the newly constructed object is set to length.
        9. The byteLength property of the newly constructed object is set to byteLength.
        10.The buffer property of the newly constructed object is set to O.
        11.The byteOffset property of the newly constructed object is set to byteOffset.
    4. Else:
        1. Let n to be the result of calling [[Get]] on V with property name "length".
        2. Let length be the result of calling ToUInt32(n).
        3. The length property of the newly constructed object is set to length.
        4. The byteLength property of the newly constructed object is set to length multiplied by the size in bytes of Type.
        5. Let arrayBuffer be an object constructed as if by a call to the built-in ArrayBuffer constructor, as "new ArrayBuffer(byteLength)".
        6. Initialize i to be 0.
        7. While i < length:
            1. Let x be the result of calling [[Get]] on arrayBuffer with property name ToString(i).
            2. Let indexDesc be a property descriptor.
            3. Set indexDesc.Writable to true.
            4. Set indexDesc.Enumerable to true.
            5. Set indexDesc.Configurable to false.
```

    6. Set indexDesc.Value to x.
    7. Call [[DefineOwnProperty]] on the newly constructed object with arguments ToString(i), indexDesc, and false.
    8. Set i to i + 1.
  8. The buffer property of the newly constructed object is set to arrayBuffer.
  9. The byteOffset property of the newly constructed object is set to 0.

3. Otherwise:

   ▪Throw an exception

2.9.1.2.3 Properties of the TypeArray Constructor

The value of the [[Prototype]] internal property of the *Type*Array constructor is the Function prototype object (15.3.4).

Besides the internal properties and the length property (whose value is 3), the *Type*Array constructor has the following properties:

2.9.1.2.3.1 TypeArray.prototype

The initial value of *Type*Array.prototype is the *Type*Array prototype object (16.2.4).

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

2.9.1.2.3.2 typeArray.BYTES_PER_ELEMENT

The initial value of *Type*Array.BYTES_PER_ELEMENT is the size in bytes of *Type*.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

2.9.1.2.4 Properties of the TypeArray Prototype Object

The value of the [[Prototype]] internal property of the TypeArray prototype object is the standard built-in Object prototype object (15.2.4). It’s [[Class]] is "*Type*Array".

2.9.1.2.4.1 TypeArray.prototype.constructor

The initial value of *Type*Array.prototype.constructor is the standard built-in *Type*Array constructor.

2.9.1.2.4.2 TypeArray.prototype.set(Array [, offset])

Set multiple values in the TypedArray, reading from the array input., reading input values from the array. The optional offset value indicates the index in the current array where values are written. If omitted, it is assumed to be 0.

1. If this does not have class "*Type*Array", throw a TypeError.
2. Let offsetIndex be ToUInt32(offset)
3. Let *O* be the result of calling ToObject(array).

4. Let srcLength be the result of calling [[Get]] on O with property name "length".
5. Let targetLength be the result of calling [[Get]] on this with property name "length"
6. If srcLength + offset > targetLength, throw a RangeError.
7. Let temp be a new TypeArray created as if by a call to "new TypeArray(srcLength)"
8. Let k be 0
9. While k < srcLength
    1. Let v be the result of calling [[Get]] on src with property name toString(k)
    2. Call [[Put]] on temp with arguments ToString(k), v, and false
10. Let k be offset
11. While k < targetLength
    1. Let v be the result of calling [[Get]] on temp with property name ToString(k-offset)
    2. Call [[Put]] on temp with arguments ToString(k), v, and false

2.9.1.2.4.3 TypeArray.prototype.subarray(begin [, end])

Returns a new TypedArray view of the ArrayBuffer store for this TypedArray, referencing the elements at begin, inclusive, up to end, exclusive. If either begin or end is negative, it refers to an index from the end of the array, as opposed to from the beginning.

1. If this does not have class "*Type*Array", throw a TypeError.
2. Let srcLength be the result of calling [[Get]] on this with property name "length"
3. Let beginInt be ToInt32(begin)
4. If beginInt < 0, let beginInt be srcLength + beginInt
5. Let beginIndex be min(srcLength, max(0, beginInt))
6. Let endInt be ToInt32(end) if end was provided, else srcLength.
7. If endInt <0,let endInt be srcLength + endInt
8. Let endIndex be max(0,min(srcLength, endInt))
9. If endIndex < beginIndex, let endIndex be beginIndex
10. Return a new TypeArray with the following values for it’s proeprties:
    1. The length property of the newly constructed object is set to endIndex - beginIndex
    2. The byteLength property of the newly constructed object is set to length multiplied by the size in bytes of *Type*.
    3. The buffer property of the newly constructed object is set to this.buffer.
    4. The byteOffset property of the newly constructed object is set to this.offset + beginIndex.

2.9.1.2.5 Properties of TypeArray Instances

*Type*Array instances inherit properties from the *Type*Array prototype object and their [[Class]] internal property value is "*Type*Array". *Type*Array instances also have the following properties.

2.9.1.2.5.1 [[DefineOwnProperty]] (P, Desc, Throw)

*Type*Array objects use a variation of the [[DefineOwnProperty]] internal method used for other native ECMAScript objects (8.12.9).

When the [[DefineOwnProperty]] internal method of A is called with property P, Property Descriptor Desc and Boolean flag Throw, the following steps are taken:

1. Let succeeded be the result of calling the default [[DefineOwnProperty]] internal method (8.12.9) on A passing P, Desc, and Throw as arguments.
2. If succeeded is false, return false.
3. If Desc contains a Value field, let newValue be Desc.Value
4. Let convertedValue to To*Type*(newValue)
5. Let index be ToUInt32(P)
6. Call the SetValueInBuffer internal operation with arguments A.buffer.[[NativeBuffer]], A.byteOffset, index, convertedValue, and *Type*.
7. Return true.

The internal operation SetValueInBuffer takes five parameters, a native buffer nativeBuffer, an integer byteOffset, an integer index, a value of type *Type* newValue, and a Type valueType. It operates as follows:

1. Let size be the size in bytes of the type valueType.
2. Let bytes be the array of bytes from nativeBuffer between offset byteOffset+(index*size) and offset byteOffset+((index+1)*size)-1 inclusive.
3. Let newValueBytes be the result of converting newValue to an array of bytes, using the platform endianness.
4. Set each byte of bytes from the corresponding byte of newValueBytes.

2.9.1.2.5.2 [[GetOwnProperty]] (P)

*Type*Array objects use a variation of the [[GetOwnProperty]] internal method used for other native ECMAScript objects (8.12.1). This special internal method provides access to named properties corresponding to the individual index values of the *Type*Array objects.

When the [[GetOwnProperty]] internal method of A is called with property name P, the following steps are taken:

1. Let desc be the result of calling the default [[GetOwnProperty]] internal method (8.12.1) on A with argument P.
2. If desc is not undefined return desc.
3. If ToString(abs(ToInteger(P))) is not the same value as P, return undefined.

4. Let length be the result of a calling [[Get]] on A with parameter "length"

5. Let index be ToInteger(P).

6. If length ≤ index, return undefined.

7. Let isLittleEndian be true if the platform endianness is little endian, else false.

8. Let value be the result of calling the GetValueFromBuffer internal operation with arguments A.buffer.[[NativeBuffer]], A.byteOffset, index, *Type*, and littleEndian.

9. Return a Property Descriptor { [[Value]]: value, [[Enumerable]]: true, [[Writable]]: true, [[Configurable]]: false }

The internal operation GetValueFromBuffer takes three parameters, a native buffer nativeBuffer, an integer byteOffset, an integer index, a Type valueType, and a boolean isLittleEndian. It operates as follows:

1. Let size be the size in bytes of the type valueType.

2. Let bytes be the array of bytes from nativeBuffer between offset byteOffset+(index*size) and offset byteOffset+((index+1)*size)-1 inclusive.

3. Let rawValue be the result of convert the array bytes to a value of type valueType, using little endian if isLittleEndian is true, otherwise big endian.

4. If valueType is Float32 and rawValue is a Float32 representation of IEEE754 NaN, return the NaN Number value.

5. Else, if valueType is Float64 and rawValue is a Float64 representation of IEEE754 NaN, return the NaN Number value.

6. Else, return the Number value that that represents the same numeric value as rawValue

2.9.1.2.5.3 length

The value of the length property is the length of the TypeArray object, which was fixed at creation. This property has attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]:**false** }.

2.9.1.2.5.4 byteLength

The value of the byteLength property is the length of the TypeArray object, which was fixed at creation. This property has attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]:**false** }.

2.9.1.2.5.5 buffer

The value of the buffer property is the length of the TypeArray object, which was fixed at creation. This property has attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]:**false** }.

2.9.1.2.5.6 byteOffset

The value of the byteOffset property is the length of the TypeArray object, which was fixed at creation. This property has attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]:**false** }.

2.9.1.3 DataView Objects

This section describes DataView Objects.

2.9.1.3.1 The DataView Constructor called as a function

When DataView is called as a function rather than as a constructor, it creates and initialises a new DataView object. Thus the function call DataView(…) is equivalent to the object creation expression new DataView(…) with the same arguments.

2.9.1.3.2 The DataView Constructor

When DataView is called as part of a new expression, it is a constructor: it initialises the newly created object.

2.9.1.3.2.1 New DataView (buffer [, byteOffset [, byteLength]])

The [[Prototype]] internal property of the newly constructed object is set to the original DataView prototype object, the one that is the initial value of DataView.prototype (16.1.3.1). The [[Class]] internal property of the newly constructed object is set to "DataView". The [[Extensible]] internal property of the newly constructed object is set to true.

The remaining proeprties are set as follows:

1. Let O be ToObject(buffer)
2. If the [[Class]] internal property of O is not "ArrayBuffer", raise a TypeError.
3. Let byteOffset be the result of calling ToUInt32 on byteOffset, if provided, or else 0.
4. Let bufferLength be the result of calling [[Get]] on O with property name "byteLength".
5. Let byteLength be the result of calling ToUInt32 on byteLength, if provided, or else bufferLength – byteOffset.
6. If byteOffset + byteLength is greater than bufferLength, raise a RangeError exception.
7. The byteLength property of the newly constructed object is set to byteLength.
8. The buffer property of the newly constructed object is set to O.
9. The byteOffset property of the newly constructed object is set to byteOffset.

2.9.1.3.3 Properties of the DataView Constructor

The value of the [[Prototype]] internal property of the DataView constructor is the Function prototype object (15.3.4).

Besides the internal properties and the length property (whose value is 3), the DataView constructor has the following properties:

2.9.1.3.3.1 DataView.prototype

The initial value of DataView.prototype is the DataView prototype object (16.1.4).

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

2.9.1.3.4 Properties of the DataView Prototype Object

The value of the [[Prototype]] internal property of the DataView prototype object is the standard built-in Object

prototype object (15.2.4). The [[Class]] internal property of the newly constructed object is set to **"Object"**. The [[Extensible]] internal property of the newly constructed object is set to **true**.

The internal operation GetValue(byteOffset, isLittleEndian, type) used by functions on DataView instances is defined as follows:

1. Let byteOffsetInt be ToUInt32(byteOffset)
2. Let totalOffset be byteOffsetInt plus the result of calling [[Get]] on this with parameter "byteOffset"
3. Let byteLength be the result of calling [[Get]] on this with parameter "byteLength"
4. If totalOffset >= byteLength, raise a RangeError
5. Let value be the result of calling the GetValueFromBuffer internal operation (9.2.5.2) with arguments this.buffer.[[NativeBuffer]], totalOffset, 0 and type.
6. Return value

The internal operation SetValue(byteOffset, isLittleEndian, type, value) used by functions on DataView instances is defined as follows:

1. Let byteOffsetInt be ToUInt32(byteOffset)
2. Let totalOffset be byteOffsetInt plus the result of calling [[Get]] on this with parameter "byteOffset"
3. Let byteLength be the result of calling [[Get]] on this with parameter "byteLength"
4. If totalOffset >= byteLength, raise a RangeError
5. Let value be the result of calling the SetValueInBuffer internal operation (9.2.5.2) with arguments this.buffer.[[NativeBuffer]], totalOffset, 0, value and type.
6. Return value

2.9.1.3.4.1 DataView.prototype.constructor

The initial value of DataView.prototype.constructor is the standard built-in DataView constructor.

2.9.1.3.4.2 DataView.prototype.GetInt8(byteOffset)

Gets the Int8 value at offset byteOffset in the DataView.

1. Let O be ToObject(this)
2. If the [[Class]] internal property of O is not "DataView", raise a TypeError.
3. Return GetValue(byteOffset, true, Int8)

2.9.1.3.4.3 DataView.prototype.GetUInt8(byteOffset)

Gets the UInt8 value at offset byteOffset in the DataView.

1. Let O be ToObject(this)
2. If the [[Class]] internal property of O is not "DataView", raise a TypeError.
3. Return GetValue(byteOffset, true, UInt8)

2.9.1.3.4.4 DataView.prototype.GetInt16(byteOffset, littleEndian)

Gets the Int16 value at offset byteOffset in the DataView, using the provided endianness.

1. Let O be ToObject(this)
2. Let isLittleEndian be ToBoolean(littleEndian) if provided, else false
3. If the [[Class]] internal property of O is not "DataView", raise a TypeError.
4. Return GetValue(byteOffset, isLittleEndian, Int16)

2.9.1.3.4.5 DataView.prototype.GetUInt16(byteOffset, littleEndian)

Gets the Uint16 value at offset byteOffset in the DataView, using the provided endianness.

1. Let O be ToObject(this)
2. Let isLittleEndian be ToBoolean(littleEndian) if provided, else false
3. If the [[Class]] internal property of O is not "DataView", raise a TypeError.
4. Return GetValue(byteOffset, isLittleEndian, Uint16)

2.9.1.3.4.6 DataView.prototype.GetInt32(byteOffset, littleEndian)

Gets the Int32 value at offset byteOffset in the DataView, using the provided endianness.

1. Let O be ToObject(this)
2. Let isLittleEndian be ToBoolean(littleEndian) if provided, else false
3. If the [[Class]] internal property of O is not "DataView", raise a TypeError.
4. Return GetValue(byteOffset, isLittleEndian, Int32)

2.9.1.3.4.7 DataView.prototype.GetUInt32(byteOffset, littleEndian)

Gets the Uint32 value at offset byteOffset in the DataView, using the provided endianness.

1. Let O be ToObject(this)
2. Let isLittleEndian be ToBoolean(littleEndian) if provided, else false
3. If the [[Class]] internal property of O is not "DataView", raise a TypeError.
4. Return GetValue(byteOffset, isLittleEndian, Uint32)

2.9.1.3.4.8 DataView.prototype.GetFloat32(byteOffset, littleEndian)

Gets the Float32 value at offset byteOffset in the DataView, using the provided endianness.

1. Let O be ToObject(this)
2. Let isLittleEndian be ToBoolean(littleEndian) if provided, else false
3. If the [[Class]] internal property of O is not "DataView", raise a TypeError.
4. Return GetValue(byteOffset, isLittleEndian, Float32)

2.9.1.3.4.9 DataView.prototype.GetFloat64(byteOffset, littleEndian)

Gets the Float64 value at offset byteOffset in the DataView, using the provided endianness.

1. Let O be ToObject(this)
2. Let isLittleEndian be ToBoolean(littleEndian) if provided, else false
3. If the [[Class]] internal property of O is not "DataView", raise a TypeError.
4. Return GetValue(byteOffset, isLittleEndian, Float64)

2.9.1.3.4.10 DataView.prototype.SetInt8(byteOffset, value)

Sets the Int8 value at offset byteOffset in the DataView.

1. Let O be ToObject(this)
2. If the [[Class]] internal property of O is not "DataView", raise a TypeError.
3. Return GetValue(byteOffset, true, Int8, ToInt8(value))

2.9.1.3.4.11 DataView.prototype.SetUInt8(byteOffset, value)

Sets the Uint8 value at offset byteOffset in the DataView.

1. Let O be ToObject(this)
2. If the [[Class]] internal property of O is not "DataView", raise a TypeError.
3. Return GetValue(byteOffset, true, Uint8, ToUint8(value))

2.9.1.3.4.12 DataView.prototype.SetInt16(byteOffset, value, littleEndian)

Sets the Int16 value at offset byteOffset in the DataView.

1. Let O be ToObject(this)
2. Let isLittleEndian be ToBoolean(littleEndian) if provided, else false
3. If the [[Class]] internal property of O is not "DataView", raise a TypeError.
4. Return GetValue(byteOffset, isLittleEndian, Int16, ToInt16(value))

2.9.1.3.4.13 DataView.prototype.SetUInt16(byteOffset, value, littleEndian)

Sets the Uint16 value at offset byteOffset in the DataView.

1. Let O be ToObject(this)
2. Let isLittleEndian be ToBoolean(littleEndian) if provided, else false
3. If the [[Class]] internal property of O is not "DataView", raise a TypeError.
4. Return GetValue(byteOffset, isLittleEndian, Uint16, ToUint16(value))

2.9.1.3.4.14 DataView.prototype.SetInt32(byteOffset, value, littleEndian)

Sets the Int32 value at offset byteOffset in the DataView.

1. Let O be ToObject(this)
2. Let isLittleEndian be ToBoolean(littleEndian) if provided, else false
3. If the [[Class]] internal property of O is not "DataView", raise a TypeError.
4. Return GetValue(byteOffset, isLittleEndian, Int32, ToInt32(value))

2.9.1.3.4.15 DataView.prototype.SetUInt32(byteOffset, value, littleEndian)

Sets the Uint32 value at offset byteOffset in the DataView.

1. Let O be ToObject(this)
2. Let isLittleEndian be ToBoolean(littleEndian) if provided, else false
3. If the [[Class]] internal property of O is not "DataView", raise a TypeError.
4. Return GetValue(byteOffset, isLittleEndian, Uint32, ToUint32(value))

2.9.1.3.4.16 DataView.prototype.SetFloat32(byteOffset, value, littleEndian)

Sets the Float32 value at offset byteOffset in the DataView.

1. Let O be ToObject(this)
2. Let isLittleEndian be ToBoolean(littleEndian) if provided, else false
3. If the [[Class]] internal property of O is not "DataView", raise a TypeError.
4. Return GetValue(byteOffset, isLittleEndian, Float32, ToFloat32(value))

2.9.1.3.4.17 DataView.prototype.SetFloat64(byteOffset, value, littleEndian)

Sets the Float64 value at offset byteOffset in the DataView.

1. Let O be ToObject(this)
2. Let isLittleEndian be ToBoolean(littleEndian) if provided, else false
3. If the [[Class]] internal property of O is not "DataView", raise a TypeError.

4. Return GetValue(byteOffset, isLittleEndian, Float64, ToFloat64(value))

2.9.1.3.5 Properties of DataView Instances

DataView instances inherit properties from the **DataView** prototype object and their [[Class]] internal property value is "DataView". DataView instances also have the following properties.

2.9.1.3.5.1 byteLength

The value of the **byteLength** property is the length of the **DataView** object, which was fixed at creation. This property has attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]:**false** }.

2.9.1.3.5.2 buffer

The value of the **buffer** property is the length of the **DataView** object, which was fixed at creation. This property has attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]:**false** }.

2.9.1.3.5.3 byteOffset

The value of the **byteOffset** property is the length of the **DataView** object, which was fixed at creation. This property has attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]:**false** }.

2.9.2 Properties of Error Constructor

Internet Explorer 10 ECMAScript defines an additional property on Error constructor of [ECMA-262/5]. The additional property is described in the following section.

2.9.2.1 stackTraceLimit

The initial value of **stackTraceLimit** is the numeric value 10. This property has the attributes { [[Enumerable]]:true, [[Configurable]]:true, [[Writable]]:true }.

2.9.3 Properties of Error Instances

Internet Explorer ECMAScript defines additional error instances inherited from the [[Prototype]] object of [ECMA-262/5]. This error instance is described in the following section.

2.9.3.1 stack

The initial value of **stack** is undefined. This property has the attributes { [[Enumerable]]:true, [[Configurable]]:true, [[Writable]]:true }. When an error is thrown the stack property is set to contain a string value which describes the stack frames formatted as described below.

```
"<Error Type>: <Error Description>
   at FunctionName (<Fully qualified file/URL>:<line#>:<col#>)
   at FunctionName (<Fully qualified file/URL>:<line#>:<col#>)
   at FunctionName (<Fully qualified file/URL>:<line#>:<col#>)
   at FunctionName (<Fully qualified file/URL>:<line#>:<col#>)
   at FunctionName (<Fully qualified file/URL>:<line#>:<col#>)
   at FunctionName (<Fully qualified file/URL>:<line#>:<col#>)
   at FunctionName (<Fully qualified file/URL>:<line#>:<col#>)
   at FunctionName (<Fully qualified file/URL>:<line#>:<col#>)
   at FunctionName (<Fully qualified file/URL>:<line#>:<col#>)
```

```
     at FunctionName (<Fully qualified file/URL>:<line#>:<col#>)"
```

The number of stack frames shown is controlled by the **stackTraceLimit** property defined on the Error constructor.

3 Security Considerations

There are no additional security considerations.

4 Appendix A: Product Behavior

The information in this specification is applicable to the following Microsoft products or supplemental software. References to product versions include released service packs:

- Windows® Internet Explorer® 9
- Windows® Internet Explorer® 10

Exceptions, if any, are noted below. If a service pack or Quick Fix Engineering (QFE) number appears with the product version, behavior changed in that service pack or QFE. The new behavior also applies to subsequent service packs of the product unless otherwise specified. If a product edition appears with the product version, behavior is different in that product edition.

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

5 Change Tracking

This section identifies changes that were made to the [MS-ES5EX] protocol document between the February 2012 and July 2012 releases. Changes are classified as New, Major, Minor, Editorial, or No change.

The revision class **New** means that a new document is being released.

The revision class **Major** means that the technical content in the document was significantly revised. Major changes affect protocol interoperability or implementation. Examples of major changes are:

- A document revision that incorporates changes to interoperability requirements or functionality.
- An extensive rewrite, addition, or deletion of major portions of content.
- The removal of a document from the documentation set.
- Changes made for template compliance.

The revision class **Minor** means that the meaning of the technical content was clarified. Minor changes do not affect protocol interoperability or implementation. Examples of minor changes are updates to clarify ambiguity at the sentence, paragraph, or table level.

The revision class **Editorial** means that the language and formatting in the technical content was changed. Editorial changes apply to grammatical, formatting, and style issues.

The revision class **No change** means that no new technical or language changes were introduced. The technical content of the document is identical to the last released version, but minor editorial and formatting changes, as well as updates to the header and footer information, and to the revision summary, may have been made.

Major and minor changes can be described further using the following change types:

- New content added.
- Content updated.
- Content removed.
- New product behavior note added.
- Product behavior note updated.
- Product behavior note removed.
- New protocol syntax added.
- Protocol syntax updated.
- Protocol syntax removed.
- New content added due to protocol revision.
- Content updated due to protocol revision.
- Content removed due to protocol revision.
- New protocol syntax added due to protocol revision.

- Protocol syntax updated due to protocol revision.
- Protocol syntax removed due to protocol revision.
- New content added for template compliance.
- Content updated for template compliance.
- Content removed for template compliance.
- Obsolete document removed.

Editorial changes are always classified with the change type **Editorially updated.**

Some important terms used in the change type descriptions are defined as follows:

- **Protocol syntax** refers to data elements (such as packets, structures, enumerations, and methods) as well as interfaces.
- **Protocol revision** refers to changes made to a protocol that affect the bits that are sent over the wire.

The changes made to this document are listed in the following table. For more information, please contact protocol@microsoft.com.

Section	Tracking number (if applicable) and description	Major change (Y or N)	Change type
1 Introduction	Updated document to remove beta tagging.	N	Content updated.

6 Index