



Draft

*Ecma/TC39/2011/0xx*

**Standard** ECMA-262

6<sup>th</sup> Edition / Draft May 4, 2012

**Draft**

## **ECMAScript Language Specification**

Report Errors and Issues at: <https://bugs.ecmascript.org>

Product: Draft for 6th Edition

Component: choose an appropriate one

Version: May 4 2012 Draft

**Standard**

Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001

DRAFT



COPYRIGHT PROTECTED DOCUMENT

## Contents

	Page
<b>Introduction.....</b>	<b>vii</b>
<b>1 Scope.....</b>	<b>1</b>
<b>2 Conformance.....</b>	<b>1</b>
<b>3 Normative references.....</b>	<b>1</b>
<b>4 Overview .....</b>	<b>1</b>
<b>4.1 Web Scripting.....</b>	<b>2</b>
<b>4.2 Language Overview .....</b>	<b>2</b>
<b>4.2.1 Objects .....</b>	<b>3</b>
<b>4.2.2 The Strict Variant of ECMAScript .....</b>	<b>4</b>
<b>4.3 Terms and definitions.....</b>	<b>4</b>
<b>5 Notational Conventions.....</b>	<b>7</b>
<b>5.1 Syntactic and Lexical Grammars.....</b>	<b>7</b>
<b>5.1.1 Context-Free Grammars .....</b>	<b>7</b>
<b>5.1.2 The Lexical and RegExp Grammars.....</b>	<b>8</b>
<b>5.1.3 The Numeric String Grammar .....</b>	<b>8</b>
<b>5.1.4 The Syntactic Grammar.....</b>	<b>8</b>
<b>5.1.5 The JSON Grammar .....</b>	<b>9</b>
<b>5.1.6 Grammar Notation.....</b>	<b>9</b>
<b>5.2 Algorithm Conventions .....</b>	<b>12</b>
<b>5.3 Static Semantic Rules .....</b>	<b>13</b>
<b>6 Source Text .....</b>	<b>13</b>
<b>7 Lexical Conventions .....</b>	<b>14</b>
<b>7.1 Unicode Format-Control Characters .....</b>	<b>15</b>
<b>7.2 White Space .....</b>	<b>15</b>
<b>7.3 Line Terminators .....</b>	<b>16</b>
<b>7.4 Comments .....</b>	<b>17</b>
<b>7.5 Tokens .....</b>	<b>17</b>
<b>7.6 Identifier Names and Identifiers .....</b>	<b>18</b>
<b>7.6.1 Reserved Words .....</b>	<b>19</b>
<b>7.7 Punctuators .....</b>	<b>20</b>
<b>7.8 Literals .....</b>	<b>20</b>
<b>7.8.1 Null Literals .....</b>	<b>20</b>
<b>7.8.2 Boolean Literals .....</b>	<b>21</b>
<b>7.8.3 Numeric Literals .....</b>	<b>21</b>
<b>7.8.4 String Literals .....</b>	<b>24</b>
<b>7.8.5 Regular Expression Literals .....</b>	<b>26</b>
<b>7.9 Automatic Semicolon Insertion .....</b>	<b>27</b>
<b>7.9.1 Rules of Automatic Semicolon Insertion .....</b>	<b>27</b>
<b>7.9.2 Examples of Automatic Semicolon Insertion .....</b>	<b>28</b>
<b>8 Types .....</b>	<b>29</b>
<b>8.1 The Undefined Type .....</b>	<b>30</b>
<b>8.2 The Null Type .....</b>	<b>30</b>
<b>8.3 The Boolean Type .....</b>	<b>30</b>
<b>8.4 The String Type .....</b>	<b>30</b>
<b>8.5 The Number Type .....</b>	<b>30</b>
<b>8.6 The Object Type .....</b>	<b>31</b>
<b>8.6.1 Property Attributes .....</b>	<b>31</b>
<b>8.6.2 Object Internal Properties and Methods .....</b>	<b>32</b>

8.7	The List and Record Specification Type .....	36
8.8	The Completion Record Specification Type .....	36
8.9	The Reference Specification Type .....	37
8.9.1	GetValue (V) .....	38
8.9.2	PutValue (V, W) .....	39
8.9.3	GetThisValue (V) .....	39
8.10	The Property Descriptor and Property Identifier Specification Types .....	40
8.10.1	IsAccessorDescriptor ( Desc ) .....	40
8.10.2	IsDataDescriptor ( Desc ) .....	40
8.10.3	IsGenericDescriptor ( Desc ) .....	40
8.10.4	FromPropertyDescriptor ( Desc ) .....	41
8.10.5	ToPropertyDescriptor ( Obj ) .....	41
8.11	The Lexical Environment and Environment Record Specification Types .....	42
8.12	Algorithms for Object Internal Methods .....	42
8.12.1	[[GetOwnProperty]] (P) .....	42
8.12.2	[[GetProperty]] (P) .....	42
8.12.3	[[Get]] (P [, accessorThisValue] ) .....	42
8.12.4	[[CanPut]] (P) .....	43
8.12.5	[[Put]] ( P, V, Throw[, accessorThisValue] ) .....	43
8.12.6	[[HasProperty]] (P) .....	43
8.12.7	[[Delete]] (P, Throw) .....	44
8.12.8	[[DefaultValue]] (hint) .....	44
8.12.9	[[DefineOwnProperty]] (P, Desc, Throw) .....	45
8.12.10	[[Enumerate]] (includePrototype, onlyEnumerable) .....	46
8.12.11	[[Iterate]] ( ) .....	46
9	Abstract Operations .....	46
9.1	Type Conversion and Testing .....	47
9.1.1	ToPrimitive .....	47
9.1.2	ToBoolean .....	47
9.1.3	ToNumber .....	48
9.1.4	ToInteger .....	50
9.1.5	ToInt32: (Signed 32 Bit Integer) .....	51
9.1.6	ToUint32: (Unsigned 32 Bit Integer) .....	51
9.1.7	ToUint16: (Unsigned 16 Bit Integer) .....	51
9.1.8	ToString .....	52
9.1.9	ToObject .....	53
9.1.10	ToPropertyKey .....	53
9.2	Testing and Comparison Operations .....	53
9.2.1	CheckObjectCoercible .....	53
9.2.2	IsCallable .....	54
9.2.3	The SameValue Algorithm .....	54
9.3	Property Access Operations .....	54
9.3.1	Invoke .....	54
10	Executable Code and Execution Contexts .....	55
10.1	Types of Executable Code .....	55
10.1.1	Strict Mode Code .....	55
10.1.2	Code Realms .....	56
10.2	Lexical Environments .....	56
10.2.1	Environment Records .....	56
10.2.2	Lexical Environment Operations .....	62
10.3	Code Realms .....	63
10.4	Execution Contexts .....	64
10.4.1	Identifier Resolution .....	65
10.4.3	GetThisEnvironment .....	65
10.4.3	This Resolution .....	65
10.5	Declaration Binding Instantiation .....	65
10.5.1	Top-Level Declaration Instantiation .....	65
10.5.2	Module Declaration Instantiation .....	66

10.5.3 Function Declaration Instantiation.....	66
10.5.4 Block Declaration Instantiation.....	68
10.6 Arguments Object .....	68
11 Expressions.....	71
11.1 Primary Expressions .....	71
11.1.1 The this Keyword .....	71
11.1.2 Identifier Reference .....	71
11.1.3 Literal Reference .....	71
11.1.4 Array Initialiser.....	72
11.1.5 Object Initialiser.....	75
11.1.6 Function Expressions.....	78
11.1.7 Generator Comprehensions .....	78
11.1.8 The Grouping Operator .....	78
11.2 Left-Hand-Side Expressions.....	79
Static Semantics .....	79
11.2.1 Property Accessors .....	80
11.2.2 The new Operator .....	80
11.2.3 Function Calls .....	81
11.2.4 The super Keyword.....	81
11.2.5 Argument Lists .....	83
11.3 Postfix Expressions.....	84
11.3.1 Postfix Increment Operator .....	84
11.3.2 Postfix Decrement Operator .....	84
11.4 Unary Operators .....	85
11.4.1 The delete Operator.....	85
11.4.2 The void Operator .....	86
11.4.3 The typeof Operator.....	86
11.4.4 Prefix Increment Operator .....	86
11.4.5 Prefix Decrement Operator .....	87
11.4.6 Unary + Operator.....	87
11.4.7 Unary - Operator .....	87
11.4.8 Bitwise NOT Operator ( ~ ) .....	87
11.4.9 Logical NOT Operator ( ! ) .....	88
11.5 Multiplicative Operators .....	88
11.5.1 Applying the * Operator .....	88
11.5.2 Applying the / Operator .....	89
11.5.3 Applying the % Operator .....	89
11.6 Additive Operators .....	90
11.6.1 The Addition operator ( + ) .....	90
11.6.2 The Subtraction Operator ( - ) .....	90
11.6.3 Applying the Additive Operators to Numbers .....	91
11.7 Bitwise Shift Operators .....	91
11.7.1 The Left Shift Operator ( << ) .....	91
11.7.2 The Signed Right Shift Operator ( >> ) .....	92
11.7.3 The Unsigned Right Shift Operator ( >>> ) .....	92
11.8 Relational Operators .....	92
11.8.1 Runtime Semantics .....	93
11.9 Equality Operators .....	95
11.9.1 Runtime Semantics .....	96
11.10 Binary Bitwise Operators.....	98
11.11 Binary Logical Operators .....	99
11.12 Conditional Operator ( ? : ) .....	100
11.13 Assignment Operators .....	100
Static Semantics .....	101
Runtime Semantics.....	101
11.13.1 Destructuring Assignment .....	102
11.14 Comma Operator ( , ) .....	105

<b>12</b>	<b>Statements and Declarations</b>	106
	<b>Static Semantics</b>	106
	<b>Runtime Semantics</b>	107
<b>12.1</b>	<b>Block</b>	108
<b>12.2</b>	<b>Declarations and the Variable Statement</b>	110
<b>12.2.1</b>	<b>Let and Const Declarations</b>	110
<b>12.2.2</b>	<b>Variable Statement</b>	113
<b>12.2.4</b>	<b>Destructuring Binding Patterns</b>	116
<b>12.3</b>	<b>Empty Statement</b>	122
<b>12.4</b>	<b>Expression Statement</b>	122
<b>12.5</b>	<b>The if Statement</b>	122
<b>12.6</b>	<b>Iteration Statements</b>	123
<b>12.6.1</b>	<b>The do-while Statement</b>	124
<b>12.6.2</b>	<b>The while Statement</b>	124
<b>12.6.3</b>	<b>The for Statement</b>	124
<b>12.6.4</b>	<b>The for-in and for-of Statements</b>	126
<b>12.7</b>	<b>The continue Statement</b>	129
<b>12.8</b>	<b>The break Statement</b>	130
<b>12.9</b>	<b>The return Statement</b>	130
<b>12.10</b>	<b>The with Statement</b>	131
<b>12.11</b>	<b>The switch Statement</b>	132
<b>12.12</b>	<b>Labelled Statements</b>	136
<b>12.13</b>	<b>The throw Statement</b>	137
<b>12.14</b>	<b>The try Statement</b>	137
<b>12.15</b>	<b>The debugger statement</b>	140
<b>13</b>	<b>Functions and Generators</b>	140
<b>13.1</b>	<b>Function Definitions</b>	140
<b>13.2</b>	<b>Arrow Function Definitions</b>	146
<b>13.3</b>	<b>Method Definitions</b>	150
<b>13.4</b>	<b>Generator Definitions</b>	152
<b>13.5</b>	<b>Creating Function Objects and Constructors</b>	153
<b>14</b>	<b>Program</b>	155
<b>14.1</b>	<b>Directive Prologues and the Use Strict Directive</b>	155
<b>15</b>	<b>Standard Built-in ECMAScript Objects</b>	156
<b>15.1</b>	<b>The Global Object</b>	157
<b>15.1.1</b>	<b>Value Properties of the Global Object</b>	157
<b>15.1.2</b>	<b>Function Properties of the Global Object</b>	157
<b>15.1.3</b>	<b>URI Handling Function Properties</b>	160
<b>15.1.4</b>	<b>Constructor Properties of the Global Object</b>	164
<b>15.1.5</b>	<b>Other Properties of the Global Object</b>	165
<b>15.2</b>	<b>Object Objects</b>	165
<b>15.2.1</b>	<b>The Object Constructor Called as a Function</b>	166
<b>15.2.2</b>	<b>The Object Constructor</b>	166
<b>15.2.3</b>	<b>Properties of the Object Constructor</b>	166
<b>15.2.4</b>	<b>Properties of the Object Prototype Object</b>	169
<b>15.2.5</b>	<b>Properties of Object Instances</b>	171
<b>15.3</b>	<b>Function Objects</b>	171
<b>15.3.1</b>	<b>The Function Constructor Called as a Function</b>	171
<b>15.3.2</b>	<b>The Function Constructor</b>	172
<b>15.3.3</b>	<b>Properties of the Function Constructor</b>	173
<b>15.3.4</b>	<b>Properties of the Function Prototype Object</b>	173
<b>15.3.5</b>	<b>Properties of Function Instances</b>	175
<b>15.4</b>	<b>Array Objects</b>	177
<b>15.4.1</b>	<b>The Array Constructor Called as a Function</b>	177
<b>15.4.2</b>	<b>The Array Constructor</b>	177
<b>15.4.3</b>	<b>Properties of the Array Constructor</b>	178
<b>15.4.4</b>	<b>Properties of the Array Prototype Object</b>	180

15.4.5 Properties of Array Instances .....	198
15.5 String Objects .....	200
15.5.1 The String Constructor Called as a Function .....	200
15.5.2 The String Constructor .....	200
15.5.3 Properties of the String Constructor .....	200
15.5.4 Properties of the String Prototype Object .....	201
15.5.5 Properties of String Instances .....	212
15.6 Boolean Objects .....	213
15.6.1 The Boolean Constructor Called as a Function .....	213
15.6.2 The Boolean Constructor .....	213
15.6.3 Properties of the Boolean Constructor .....	214
15.6.4 Properties of the Boolean Prototype Object .....	214
15.6.5 Properties of Boolean Instances .....	214
15.7 Number Objects .....	215
15.7.1 The Number Constructor Called as a Function .....	215
15.7.2 The Number Constructor .....	215
15.7.3 Properties of the Number Constructor .....	215
15.7.4 Properties of the Number Prototype Object .....	217
15.7.5 Properties of Number Instances .....	221
15.8 The Math Object .....	221
15.8.1 Value Properties of the Math Object .....	221
15.8.2 Function Properties of the Math Object .....	222
15.9 Date Objects .....	230
15.9.1 Overview of Date Objects and Definitions of Abstract Operators .....	230
15.9.2 The Date Constructor Called as a Function .....	235
15.9.3 The Date Constructor .....	235
15.9.4 Properties of the Date Constructor .....	236
15.9.5 Properties of the Date Prototype Object .....	238
15.9.6 Properties of Date Instances .....	245
15.10 RegExp (Regular Expression) Objects .....	245
15.10.1 Patterns .....	245
15.10.2 Pattern Semantics .....	247
15.10.3 The RegExp Constructor Called as a Function .....	259
15.10.4 The RegExp Constructor .....	259
15.10.5 Properties of the RegExp Constructor .....	260
15.10.6 Properties of the RegExp Prototype Object .....	260
15.10.7 Properties of RegExp Instances .....	262
15.11 Error Objects .....	263
15.11.1 The Error Constructor Called as a Function .....	263
15.11.2 The Error Constructor .....	263
15.11.3 Properties of the Error Constructor .....	263
15.11.4 Properties of the Error Prototype Object .....	264
15.11.5 Properties of Error Instances .....	264
15.11.6 Native Error Types Used in This Standard .....	264
15.11.7 <i>NativeError</i> Object Structure .....	265
15.12 The JSON Object .....	267
15.12.1 The JSON Grammar .....	267
15.12.2 <i>parse</i> ( <i>text</i> [ , <i>reviver</i> ] ) .....	269
15.12.3 <i>stringify</i> ( <i>value</i> [ , <i>replacer</i> [ , <i>space</i> ] ] ) .....	270
16 Errors .....	274
Annex A (informative) Grammar Summary .....	276
A.1 Lexical Grammar .....	276
A.2 Number Conversions .....	282
A.3 Expressions .....	283
A.4 Statements .....	287
A.5 Functions and Programs .....	289
A.6 Universal Resource Identifier Character Classes .....	290
A.7 Regular Expressions .....	290

A.8 JSON.....	292
A.8.1 JSON Lexical Grammar.....	292
A.8.2 JSON Syntactic Grammar.....	293
Annex B (normative) Additional ECMAScript Features for Web Browsers.....	296
B.1 Additional Syntax.....	296
B.1.1 Numeric Literals .....	296
B.1.2 String Literals .....	296
B.2 Additional Properties.....	297
B.2.1 escape (string) .....	297
B.2.2 unescape (string) .....	298
B.2.3 String.prototype.substr (start, length).....	298
B.2.4 Date.prototype.getFullYear ().....	299
B.2.5 Date.prototype.setYear (year).....	299
B.2.6 Date.prototype.toGMTString ( ) .....	299
B.3 Other Additional Features .....	299
B.3.1 The __proto__ pseudo property.....	299
Annex C (informative) The Strict Mode of ECMAScript .....	302
Annex D (informative) Corrections and Clarifications Possible Compatibility Impact .....	304
In the 6 <sup>th</sup> Edition .....	304
In the 5 <sup>th</sup> Edition .....	304
Annex E (informative) Additions and Changes that Introduce Incompatibilities with Prior Editions .....	307
In the 6 <sup>th</sup> Edition .....	307
In the 5 <sup>th</sup> Edition .....	307
Annex F (informative) Technically Significant Corrections and Clarifications in the 5.1 Edition .....	311

DRAFT

## Introduction

This Ecma Standard is based on several originating technologies, the most well known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at Netscape and first appeared in that company's Navigator 2.0 browser. It has appeared in all subsequent browsers from Netscape and in all browsers from Microsoft starting with Internet Explorer 3.0.

The development of this Standard started in November 1996. The first edition of this Ecma Standard was adopted by the Ecma General Assembly of June 1997.

That Ecma Standard was submitted to ISO/IEC JTC 1 for adoption under the fast-track procedure, and approved as international standard ISO/IEC 16262, in April 1998. The Ecma General Assembly of June 1998 approved the second edition of ECMA-262 to keep it fully aligned with ISO/IEC 16262. Changes between the first and the second edition are editorial in nature.

The third edition of the Standard introduced powerful regular expressions, better string handling, new control statements, try/catch exception handling, tighter definition of errors, formatting for numeric output and minor changes in anticipation of forthcoming internationalisation facilities and future language growth. The third edition of the ECMAScript standard was adopted by the Ecma General Assembly of December 1999 and published as ISO/IEC 16262:2002 in June 2002.

Since publication of the third edition, ECMAScript has achieved massive adoption in conjunction with the World Wide Web where it has become the programming language that is supported by essentially all web browsers. Significant work was done to develop a fourth edition of ECMAScript. Although that work was not completed and not published<sup>1</sup> as the fourth edition of ECMAScript, it informs continuing evolution of the language. The fifth edition of ECMAScript (published as ECMA-262 5<sup>th</sup> edition) codifies de facto interpretations of the language specification that have become common among browser implementations and adds support for new features that have emerged since the publication of the third edition. Such features include accessor properties, reflective creation and inspection of objects, program control of property attributes, additional array manipulation functions, support for the JSON object encoding format, and a strict mode that provides enhanced error checking and program security.

The edition 5.1 of the ECMAScript Standard has been fully aligned with the third edition of the international standard ISO/IEC 16262:2011.

This present sixth edition of the Standard.....

ECMAScript is a vibrant language and the evolution of the language is not complete. Significant technical enhancement will continue with future editions of this specification.

This Ecma Standard has been adopted by the General Assembly of <month> <year>.

---

<sup>1</sup> Note: Please note that for ECMAScript Edition 4 the Ecma standard number "ECMA-262 Edition 4" was reserved but not used in the Ecma publication process. Therefore "ECMA-262 Edition 4" as an Ecma International publication does not exist.

"DISCLAIMER

*This draft document may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to Ecma International, except as needed for the purpose of developing any document or deliverable produced by Ecma International.*

*This disclaimer is valid only prior to final version of this document. After approval all rights on the standard are reserved by Ecma International.*

*The limited permissions are granted through the standardization phase and will not be revoked by Ecma International or its successors or assigns during this time.*

*This document and the information contained herein is provided on an "AS IS" basis and ECMA INTERNATIONAL DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE."*

DRAFT



# ECMAScript Language Specification

## 1 Scope

This Standard defines the ECMAScript scripting language.

## 2 Conformance

A conforming implementation of ECMAScript must provide and support all the types, values, objects, properties, functions, and program syntax and semantics described in this specification.

A conforming implementation of this Standard shall interpret characters in conformance with the Unicode Standard, Version 3.0 or later and ISO/IEC 10646-1 with either UCS-2 or UTF-16 as the adopted encoding form, implementation level 3. If the adopted ISO/IEC 10646-1 subset is not otherwise specified, it is presumed to be the BMP subset, collection 300. If the adopted encoding form is not otherwise specified, it is presumed to be the UTF-16 encoding form.

A conforming implementation of ECMAScript is permitted to provide additional types, values, objects, properties, and functions beyond those described in this specification. In particular, a conforming implementation of ECMAScript is permitted to provide properties not described in this specification, and values for those properties, for objects that are described in this specification.

A conforming implementation of ECMAScript is permitted to support program and regular expression syntax not described in this specification. In particular, a conforming implementation of ECMAScript is permitted to support program syntax that makes use of the "future reserved words" listed in 7.6.1.2 of this specification.

## 3 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 9899:1996, *Programming Languages – C, including amendment 1 and technical corrigenda 1 and 2*

ISO/IEC 10646-1:1993, *Information Technology – Universal Multiple-Octet Coded Character Set (UCS) plus its amendments and corrigenda*

## 4 Overview

This section contains a non-normative overview of the ECMAScript language.

ECMAScript is an object-oriented programming language for performing computations and manipulating computational objects within a host environment. ECMAScript as defined here is not intended to be computationally self-sufficient; indeed, there are no provisions in this specification for input of external data or output of computed results. Instead, it is expected that the computational environment of an ECMAScript program will provide not only the objects and other facilities described in this specification but also certain environment-specific *host objects*, whose description and behaviour are beyond the scope of this specification except to indicate that they may provide certain properties that can be accessed and certain functions that can be called from an ECMAScript program.

A **scripting language** is a programming language that is used to manipulate, customise, and automate the facilities of an existing system. In such systems, useful functionality is already available through a user interface, and the scripting language is a mechanism for exposing that functionality to program control. In this way, the existing system is said to provide a host environment of objects and facilities, which completes the

capabilities of the scripting language. A scripting language is intended for use by both professional and non-professional programmers.

ECMAScript was originally designed to be a **Web scripting language**, providing a mechanism to enliven Web pages in browsers and to perform server computation as part of a Web-based client-server architecture. ECMAScript can provide core scripting capabilities for a variety of host environments, and therefore the core scripting language is specified in this document apart from any particular host environment.

Some of the facilities of ECMAScript are similar to those used in other programming languages; in particular Java™, Self, and Scheme as described in:

Gosling, James, Bill Joy and Guy Steele. The Java™ Language Specification. Addison Wesley Publishing Co., 1996.

Ungar, David, and Smith, Randall B. Self: The Power of Simplicity. OOPSLA '87 Conference Proceedings, pp. 227–241, Orlando, FL, October 1987.

IEEE Standard for the Scheme Programming Language. IEEE Std 1178-1990.

#### 4.1 Web Scripting

A web browser provides an ECMAScript host environment for client-side computation including, for instance, objects that represent windows, menus, pop-ups, dialog boxes, text areas, anchors, frames, history, cookies, and input/output. Further, the host environment provides a means to attach scripting code to events such as change of focus, page and image loading, unloading, error and abort, selection, form submission, and mouse actions. Scripting code appears within the HTML and the displayed page is a combination of user interface elements and fixed and computed text and images. The scripting code is reactive to user interaction and there is no need for a main program.

A web server provides a different host environment for server-side computation including objects representing requests, clients, and files; and mechanisms to lock and share data. By using browser-side and server-side scripting together, it is possible to distribute computation between the client and server while providing a customised user interface for a Web-based application.

Each Web browser and server that supports ECMAScript supplies its own host environment, completing the ECMAScript execution environment.

#### 4.2 Language Overview

The following is an informal overview of ECMAScript—not all parts of the language are described. This overview is not part of the standard proper.

ECMAScript is object-based: basic language and host facilities are provided by objects, and an ECMAScript program is a cluster of communicating objects. An ECMAScript **object** is a collection of **properties** each with zero or more **attributes** that determine how each property can be used—for example, when the **Writable** attribute for a property is set to **false**, any attempt by executed ECMAScript code to change the value of the property fails. Properties are containers that hold other objects, **primitive values**, or **functions**. A primitive value is a member of one of the following built-in types: **Undefined**, **Null**, **Boolean**, **Number**, and **String**; an object is a member of the remaining built-in type **Object**; and a function is a callable object. A function that is associated with an object via a property is a **method**.

ECMAScript defines a collection of **built-in objects** that round out the definition of ECMAScript entities. These built-in objects include the global object, the **Object** object, the **Function** object, the **Array** object, the **String** object, the **Boolean** object, the **Number** object, the **Math** object, the **Date** object, the **RegExp** object, the **JSON** object, and the Error objects **Error**, **EvalError**, **RangeError**, **ReferenceError**, **SyntaxError**, **TypeError** and **URIError**.

ECMAScript also defines a set of built-in **operators**. ECMAScript operators include various unary operations, multiplicative operators, additive operators, bitwise shift operators, relational operators, equality operators, binary bitwise operators, binary logical operators, assignment operators, and the comma operator.

ECMAScript syntax intentionally resembles Java syntax. ECMAScript syntax is relaxed to enable it to serve as an easy-to-use scripting language. For example, a variable is not required to have its type declared nor are types associated with properties, and defined functions are not required to have their declarations appear textually before calls to them.

#### 4.2.1 Objects

ECMAScript does not use classes such as those in C++, Smalltalk, or Java. Instead objects may be created in various ways including via a literal notation or via **constructors** which create objects and then execute code that initialises all or part of them by assigning initial values to their properties. Each constructor is a function that has a property named “**prototype**” that is used to implement **prototype-based inheritance** and **shared properties**. Objects are created by using constructors in **new** expressions; for example, **new Date(2009, 11)** creates a new Date object. Invoking a constructor without using **new** has consequences that depend on the constructor. For example, **Date()** produces a string representation of the current date and time rather than an object.

Every object created by a constructor has an implicit reference (called the object’s *prototype*) to the value of its constructor’s “**prototype**” property. Furthermore, a prototype may have a non-null implicit reference to its prototype, and so on; this is called the *prototype chain*. When a reference is made to a property in an object, that reference is to the property of that name in the first object in the prototype chain that contains a property of that name. In other words, first the object mentioned directly is examined for such a property; if that object contains the named property, that is the property to which the reference refers; if that object does not contain the named property, the prototype for that object is examined next; and so on.

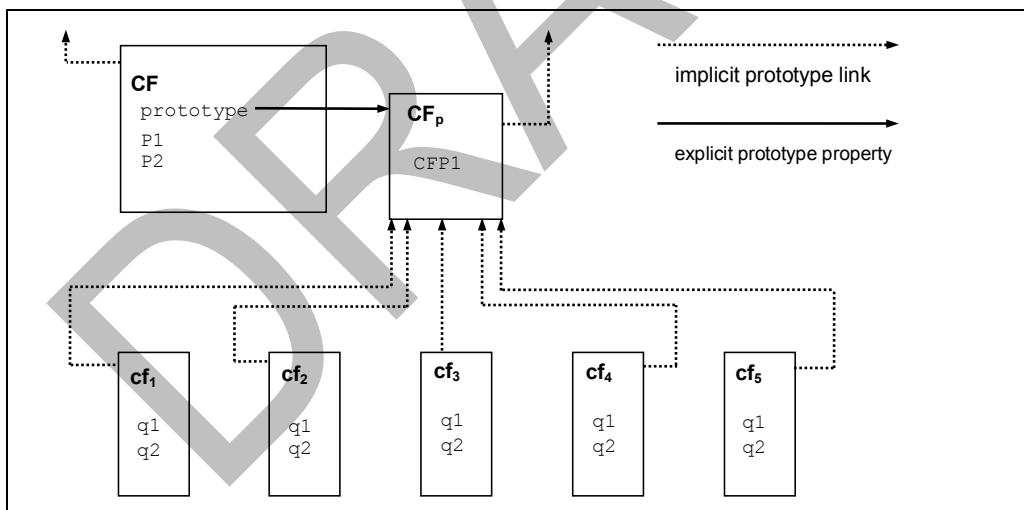


Figure 1 — Object/Prototype Relationships

In a class-based object-oriented language, in general, state is carried by instances, methods are carried by classes, and inheritance is only of structure and behaviour. In ECMAScript, the state and methods are carried by objects, and structure, behaviour, and state are all inherited.

All objects that do not directly contain a particular property that their prototype contains share that property and its value. Figure 1 illustrates this:

**CF** is a constructor (and also an object). Five objects have been created by using `new` expressions: **cf<sub>1</sub>**, **cf<sub>2</sub>**, **cf<sub>3</sub>**, **cf<sub>4</sub>**, and **cf<sub>5</sub>**. Each of these objects contains properties named **q<sub>1</sub>** and **q<sub>2</sub>**. The dashed lines represent the implicit prototype relationship; so, for example, **cf<sub>3</sub>**'s prototype is **CF<sub>p</sub>**. The constructor, **CF**, has two properties itself, named **P<sub>1</sub>** and **P<sub>2</sub>**, which are not visible to **CF<sub>p</sub>**, **cf<sub>1</sub>**, **cf<sub>2</sub>**, **cf<sub>3</sub>**, **cf<sub>4</sub>**, or **cf<sub>5</sub>**. The property named **CFP<sub>1</sub>** in **CF<sub>p</sub>** is shared by **cf<sub>1</sub>**, **cf<sub>2</sub>**, **cf<sub>3</sub>**, **cf<sub>4</sub>**, and **cf<sub>5</sub>** (but not by **CF**), as are any properties found in **CF<sub>p</sub>**'s implicit prototype chain that are not named **q<sub>1</sub>**, **q<sub>2</sub>**, or **CFP<sub>1</sub>**. Notice that there is no implicit prototype link between **CF** and **CF<sub>p</sub>**.

Unlike class-based object languages, properties can be added to objects dynamically by assigning values to them. That is, constructors are not required to name or assign values to all or any of the constructed object's properties. In the above diagram, one could add a new shared property for **cf<sub>1</sub>**, **cf<sub>2</sub>**, **cf<sub>3</sub>**, **cf<sub>4</sub>**, and **cf<sub>5</sub>** by assigning a new value to the property in **CF<sub>p</sub>**.

#### 4.2.2 The Strict Variant of ECMAScript

The ECMAScript Language recognises the possibility that some users of the language may wish to restrict their usage of some features available in the language. They might do so in the interests of security, to avoid what they consider to be error-prone features, to get enhanced error checking, or for other reasons of their choosing. In support of this possibility, ECMAScript defines a strict variant of the language. The strict variant of the language excludes some specific syntactic and semantic features of the regular ECMAScript language and modifies the detailed semantics of some features. The strict variant also specifies additional error conditions that must be reported by throwing error exceptions in situations that are not specified as errors by the non-strict form of the language.

The strict variant of ECMAScript is commonly referred to as the *strict mode* of the language. Strict mode selection and use of the strict mode syntax and semantics of ECMAScript is explicitly made at the level of individual ECMAScript code units. Because strict mode is selected at the level of a syntactic code unit, strict mode only imposes restrictions that have local effect within such a code unit. Strict mode does not restrict or modify any aspect of the ECMAScript semantics that must operate consistently across multiple code units. A complete ECMAScript program may be composed for both strict mode and non-strict mode ECMAScript code units. In this case, strict mode only applies when actually executing code that is defined within a strict mode code unit.

In order to conform to this specification, an ECMAScript implementation must implement both the full unrestricted ECMAScript language and the strict mode variant of the ECMAScript language as defined by this specification. In addition, an implementation must support the combination of unrestricted and strict mode code units into a single composite program.

### 4.3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

#### 4.3.1 type

set of data values as defined in Clause 8 of this specification

#### 4.3.2 primitive value

member of one of the types Undefined, Null, Boolean, Number, or String as defined in Clause 8

NOTE A primitive value is a datum that is represented directly at the lowest level of the language implementation.

#### 4.3.3 object

member of the type Object

NOTE An object is a collection of properties and has a single prototype object. The prototype may be the null value.

#### 4.3.4

##### **constructor**

function object that creates and initialises objects

NOTE The value of a constructor's "prototype" property is a prototype object that is used to implement inheritance and shared properties.

#### 4.3.5

##### **prototype**

object that provides shared properties for other objects

NOTE When a constructor creates an object, that object implicitly references the constructor's "prototype" property for the purpose of resolving property references. The constructor's "prototype" property can be referenced by the program expression `constructor.prototype`, and properties added to an object's prototype are shared, through inheritance, by all objects sharing the prototype. Alternatively, a new object may be created with an explicitly specified prototype by using the `Object.create` built-in function.

#### 4.3.6

##### **native object**

object in an ECMAScript implementation whose semantics are fully defined by this specification rather than by the host environment

NOTE Standard native objects are defined in this specification. Some native objects are built-in; others may be constructed during the course of execution of an ECMAScript program.

#### 4.3.7

##### **built-in object**

object supplied by an ECMAScript implementation, independent of the host environment, that is present at the start of the execution of an ECMAScript program

NOTE Standard built-in objects are defined in this specification, and an ECMAScript implementation may specify and define others. Every built-in object is a native object. A *built-in constructor* is a built-in object that is also a constructor.

#### 4.3.8

##### **host object**

object supplied by the host environment to complete the execution environment of ECMAScript

NOTE Any object that is not native is a host object.

#### 4.3.9

##### **undefined value**

primitive value used when a variable has not been assigned a value

#### 4.3.10

##### **Undefined type**

type whose sole value is the undefined value

#### 4.3.11

##### **null value**

primitive value that represents the intentional absence of any object value

#### 4.3.12

##### **Null type**

type whose sole value is the null value

#### 4.3.13

##### **Boolean value**

member of the Boolean type

NOTE There are only two Boolean values, `true` and `false`.

#### 4.3.14

##### **Boolean type**

type consisting of the primitive values **true** and **false**

#### 4.3.15

##### **Boolean object**

member of the Object type that is an instance of the standard built-in **Boolean** constructor

**NOTE** A Boolean object is created by using the **Boolean** constructor in a **new** expression, supplying a Boolean value as an argument. The resulting object has an internal property whose value is the Boolean value. A Boolean object can be coerced to a Boolean value.

#### 4.3.16

##### **String value**

primitive value that is a finite ordered sequence of zero or more 16-bit unsigned integer

**NOTE** A String value is a member of the String type. Each integer value in the sequence usually represents a single 16-bit unit of UTF-16 text. However, ECMAScript does not place any restrictions or requirements on the values except that they must be 16-bit unsigned integers.

#### 4.3.17

##### **String type**

set of all possible String values

#### 4.3.18

##### **String object**

member of the Object type that is an instance of the standard built-in **String** constructor

**NOTE** A String object is created by using the **String** constructor in a **new** expression, supplying a String value as an argument. The resulting object has an internal property whose value is the String value. A String object can be coerced to a String value by calling the **String** constructor as a function (15.5.1).

#### 4.3.19

##### **Number value**

primitive value corresponding to a double-precision 64-bit binary format IEEE 754 value

**NOTE** A Number value is a member of the Number type and is a direct representation of a number.

#### 4.3.20

##### **Number type**

set of all possible Number values including the special “Not-a-Number” (NaN) values, positive infinity, and negative infinity

#### 4.3.21

##### **Number object**

member of the Object type that is an instance of the standard built-in **Number** constructor

**NOTE** A Number object is created by using the **Number** constructor in a **new** expression, supplying a Number value as an argument. The resulting object has an internal property whose value is the Number value. A Number object can be coerced to a Number value by calling the **Number** constructor as a function (15.7.1).

#### 4.3.22

##### **Infinity**

number value that is the positive infinite Number value

#### 4.3.23

##### **NaN**

number value that is a IEEE 754 “Not-a-Number” value

**4.3.24****function**

member of the Object type that is an instance of the standard built-in `Function` constructor and that may be invoked as a subroutine

**NOTE** In addition to its named properties, a function contains executable code and state that determine how it behaves when invoked. A function's code may or may not be written in ECMAScript.

**4.3.25****built-in function**

built-in object that is a function

**NOTE** Examples of built-in functions include `parseInt` and `Math.exp`. An implementation may provide implementation-dependent built-in functions that are not described in this specification.

**4.3.26****property**

association between a name and a value that is a part of an object

**NOTE** Depending upon the form of the property the value may be represented either directly as a data value (a primitive value, an object, or a function object) or indirectly by a pair of accessor functions.

**4.3.27****method**

function that is the value of a property

**NOTE** When a function is called as a method of an object, the object is passed to the function as its `this` value.

**4.3.28****built-in method**

method that is a built-in function

**NOTE** Standard built-in methods are defined in this specification, and an ECMAScript implementation may specify and provide other additional built-in methods.

**4.3.29****attribute**

internal value that defines some characteristic of a property

**4.3.30****own property**

property that is directly contained by its object

**4.3.31****inherited property**

property of an object that is not an own property but is a property (either own or inherited) of the object's prototype

## 5 Notational Conventions

### 5.1 Syntactic and Lexical Grammars

#### 5.1.1 Context-Free Grammars

A *context-free grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of zero or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified alphabet.

Starting from a sentence consisting of a single distinguished nonterminal, called the *goal symbol*, a given context-free grammar specifies a *language*, namely, the (perhaps infinite) set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

### 5.1.2 The Lexical and RegExp Grammars

A *lexical grammar* for ECMAScript is given in clause 7. This grammar has as its terminal symbols characters (Unicode code units) that conform to the rules for *SourceCharacter* defined in Clause 6. It defines a set of productions, starting from the goal symbol *InputElementDiv* or *InputElementRegExp*, that describe how sequences of such characters are translated into a sequence of input elements.

Input elements other than white space and comments form the terminal symbols for the syntactic grammar for ECMAScript and are called ECMAScript *tokens*. These tokens are the reserved words, identifiers, literals, and punctuators of the ECMAScript language. Moreover, line terminators, although not considered to be tokens, also become part of the stream of input elements and guide the process of automatic semicolon insertion (7.9). Simple white space and single-line comments are discarded and do not appear in the stream of input elements for the syntactic grammar. A *MultiLineComment* (that is, a comment of the form “/\*...\*/” regardless of whether it spans more than one line) is likewise simply discarded if it contains no line terminator; but if a *MultiLineComment* contains one or more line terminators, then it is replaced by a single line terminator, which becomes part of the stream of input elements for the syntactic grammar.

A *RegExp grammar* for ECMAScript is given in 15.10. This grammar also has as its terminal symbols the characters as defined by *SourceCharacter*. It defines a set of productions, starting from the goal symbol *Pattern*, that describe how sequences of characters are translated into regular expression patterns.

Productions of the lexical and RegExp grammars are distinguished by having two colons “::” as separating punctuation. The lexical and RegExp grammars share some productions.

### 5.1.3 The Numeric String Grammar

Another grammar is used for translating Strings into numeric values. This grammar is similar to the part of the lexical grammar having to do with numeric literals and has as its terminal symbols *SourceCharacter*. This grammar appears in 9.3.1.

Productions of the numeric string grammar are distinguished by having three colons “:::” as punctuation.

### 5.1.4 The Syntactic Grammar

The *syntactic grammar* for ECMAScript is given in clauses 11, 12, 13 and 14. This grammar has ECMAScript tokens defined by the lexical grammar as its terminal symbols (5.1.2). It defines a set of productions, starting from the goal symbol *Program*, that describe how sequences of tokens can form syntactically correct ECMAScript programs.

When a stream of characters is to be parsed as an ECMAScript program, it is first converted to a stream of input elements by repeated application of the lexical grammar; this stream of input elements is then parsed by a single application of the syntactic grammar. The program is syntactically in error if the tokens in the stream of input elements cannot be parsed as a single instance of the goal nonterminal *Program*, with no tokens left over.

Productions of the syntactic grammar are distinguished by having just one colon “:” as punctuation.

The syntactic grammar as presented in clauses 11, 12, 13 and 14 is actually not a complete account of which token sequences are accepted as correct ECMAScript programs. Certain additional token sequences are also accepted, namely, those that would be described by the grammar if only semicolons were added to the sequence in certain places (such as before line terminator characters). Furthermore, certain token sequences that are described by the grammar are not considered acceptable if a terminator character appears in certain “awkward” places.

In certain cases in order to avoid ambiguities the syntactic grammar uses productions that permit token sequences that are not valid ECMAScript programs. In such cases a more restrictive *supplemental grammar* is provided that further restricts the acceptable token sequences. In such situations, when explicitly specific, the input elements corresponding to such a production is parsed again using a goal symbol of a supplemental grammar. The program is syntactically in error if the tokens in the stream of input elements cannot be parsed as a single instance of the supplemental goal symbol, with no tokens left over.

### 5.1.5 The JSON Grammar

The JSON grammar is used to translate a String describing a set of ECMAScript objects into actual objects. The JSON grammar is given in 15.12.1.

The JSON grammar consists of the JSON lexical grammar and the JSON syntactic grammar. The JSON lexical grammar is used to translate character sequences into tokens and is similar to parts of the ECMAScript lexical grammar. The JSON syntactic grammar describes how sequences of tokens from the JSON lexical grammar can form syntactically correct JSON object descriptions.

Productions of the JSON lexical grammar are distinguished by having two colons “::” as separating punctuation. The JSON lexical grammar uses some productions from the ECMAScript lexical grammar. The JSON syntactic grammar is similar to parts of the ECMAScript syntactic grammar. Productions of the JSON syntactic grammar are distinguished by using one colon “:” as separating punctuation.

### 5.1.6 Grammar Notation

Terminal symbols of the lexical, RegExp, and numeric string grammars, and some of the terminal symbols of the other grammars, are shown in **fixed width** font, both in the productions of the grammars and throughout this specification whenever the text directly refers to such a terminal symbol. These are to appear in a program exactly as written. All terminal symbol characters specified in this way are to be understood as the appropriate Unicode character from the ASCII range, as opposed to any similar-looking characters from other Unicode ranges.

Nonterminal symbols are shown in *italic* type. The definition of a nonterminal (*also called a “production”*) is introduced by the name of the nonterminal being defined followed by one or more colons. (The number of colons indicates to which grammar the production belongs.) One or more alternative right-hand sides for the nonterminal then follow on succeeding lines. For example, the syntactic definition:

*WhileStatement* :  
  **while** ( *Expression* ) *Statement*

states that the nonterminal *WhileStatement* represents the token **while**, followed by a left parenthesis token, followed by an *Expression*, followed by a right parenthesis token, followed by a *Statement*. The occurrences of *Expression* and *Statement* are themselves nonterminals. As another example, the syntactic definition:

*ArgumentList* :  
  *AssignmentExpression*  
  *ArgumentList* , *AssignmentExpression*

states that an *ArgumentList* may represent either a single *AssignmentExpression* or an *ArgumentList*, followed by a comma, followed by an *AssignmentExpression*. This definition of *ArgumentList* is recursive, that is, it is defined in terms of itself. The result is that an *ArgumentList* may contain any positive number of arguments, separated by commas, where each argument expression is an *AssignmentExpression*. Such recursive definitions of nonterminals are common.

The subscripted suffix “<sub>opt</sub>”, which may appear after a terminal or nonterminal, indicates an optional symbol. The alternative containing the optional symbol actually specifies two right-hand sides, one that omits the optional element and one that includes it. This means that:

*VariableDeclaration* :  
  *Identifier Initialiser*<sub>opt</sub>

is a convenient abbreviation for:

```
VariableDeclaration :
  Identifier
  Identifier Initialiser
```

and that:

```
IterationStatement :
  for ( ExpressionNoInopt ; Expressionopt ; Expressionopt ) Statement
```

is a convenient abbreviation for:

```
IterationStatement :
  for ( ; Expressionopt ; Expressionopt ) Statement
  for ( ExpressionNoIn ; Expressionopt ; Expressionopt ) Statement
```

which in turn is an abbreviation for:

```
IterationStatement :
  for ( ; ; Expressionopt ) Statement
  for ( ; Expression ; Expressionopt ) Statement
  for ( ExpressionNoIn ; ; Expressionopt ) Statement
  for ( ExpressionNoIn ; Expression ; Expressionopt ) Statement
```

which in turn is an abbreviation for:

```
IterationStatement :
  for ( ; ; ) Statement
  for ( ; Expression ) Statement
  for ( ; Expression ; ) Statement
  for ( ; Expression ; Expression ) Statement
  for ( ExpressionNoIn ; ; ) Statement
  for ( ExpressionNoIn ; ; Expression ) Statement
  for ( ExpressionNoIn ; Expression ; ) Statement
  for ( ExpressionNoIn ; Expression ; Expression ) Statement
```

so the nonterminal *IterationStatement* actually has eight alternative right-hand sides.

When the words “**one of**” follow the colon(s) in a grammar definition, they signify that each of the terminal symbols on the following line or lines is an alternative definition. For example, the lexical grammar for ECMAScript contains the production:

```
NonZeroDigit :: one of
  1 2 3 4 5 6 7 8 9
```

which is merely a convenient abbreviation for:

```
NonZeroDigit :::
  1
  2
  3
  4
  5
  6
  7
  8
  9
```

If the phrase “[empty]” appears as the right-hand side of a production, it indicates that the production's right-hand side contains no terminals or nonterminals.

If the phrase “[lookahead  $\notin$  set]” appears in the right-hand side of a production, it indicates that the production may not be used if the immediately following input token is a member of the given *set*. The *set* can be written as a list of terminals enclosed in curly braces. For convenience, the set can also be written as a nonterminal, in which case it represents the set of all terminals to which that nonterminal could expand. For example, given the definitions

```
DecimalDigit :: one of
  0 1 2 3 4 5 6 7 8 9

DecimalDigits :: 
  DecimalDigit
  DecimalDigits DecimalDigit
```

the definition

```
LookaheadExample :: 
  n [lookahead  $\notin$  {1, 3, 5, 7, 9}] DecimalDigits
  DecimalDigit [lookahead  $\notin$  DecimalDigit]
```

matches either the letter *n* followed by one or more decimal digits the first of which is even, or a decimal digit not followed by another decimal digit.

If the phrase “[no LineTerminator here]” appears in the right-hand side of a production of the syntactic grammar, it indicates that the production is a *restricted production*: it may not be used if a *LineTerminator* occurs in the input stream at the indicated position. For example, the production:

```
ThrowStatement :: 
  throw [no LineTerminator here] Expression ;
```

indicates that the production may not be used if a *LineTerminator* occurs in the program between the *throw* token and the *Expression*.

Unless the presence of a *LineTerminator* is forbidden by a restricted production, any number of occurrences of *LineTerminator* may appear between any two consecutive tokens in the stream of input elements without affecting the syntactic acceptability of the program.

When an alternative in a production of the lexical grammar or the numeric string grammar appears to be a multi-character token, it represents the sequence of characters that would make up such a token.

The right-hand side of a production may specify that certain expansions are not permitted by using the phrase “**but not**” and then indicating the expansions to be excluded. For example, the production:

```
Identifier :: 
  IdentifierName but not ReservedWord
```

means that the nonterminal *Identifier* may be replaced by any sequence of characters that could replace *IdentifierName* provided that the same sequence of characters could not replace *ReservedWord*.

Finally, a few nonterminal symbols are described by a descriptive phrase in sans-serif type in cases where it would be impractical to list all the alternatives:

```
SourceCharacter :: 
  any Unicode code unit
```

## 5.2 Algorithm Conventions

The specification often uses a numbered list to specify steps in an algorithm. These algorithms are used to precisely specify the required semantics of ECMAScript language constructs. The algorithms are not intended to imply the use of any specific implementation technique. In practice, there may be more efficient algorithms available to implement a given feature.

In order to facilitate their use in multiple parts of this specification, some algorithms, called *abstract operations*, are named and written in parameterised functional form so that they may be referenced by name from within other algorithms.

Algorithms may be associated with productions of one of the ECMAScript grammars. A production that has multiple alternative definitions will typically have a distinct algorithm for each alternative. When an algorithm is associated with a grammar production, it may reference the terminal and non-terminal symbols of the production alternative as if they were parameters of the algorithm. When used in this manner, non-terminal symbols refer to the actual alternative definition that is matched when parsing the program source code.

When an algorithm is to produce a value as a result, the directive “`return x`” is used to indicate that the result of the algorithm is the value of  $x$  and that the algorithm should terminate. The notation  $\text{Result}(n)$  is used as shorthand for “the result of step  $n$ ”.

For clarity of expression, algorithm steps may be subdivided into sequential substeps. Substeps are indented and may themselves be further divided into indented substeps. Outline numbering conventions are used to identify substeps with the first level of substeps labelled with lower case alphabetic characters and the second level of substeps labelled with lower case roman numerals. If more than three levels are required these rules repeat with the fourth level using numeric labels. For example:

1. Top-level step
  - a. Substep.
  - b. Substep
    - i. Subsubstep.
    - ii. Subsubstep.
      1. Subsubsubstep
      - a. Subsubsubsubstep

A step or substep may be written as an “if” predicate that conditions its substeps. In this case, the substeps are only applied if the predicate is true. If a step or substep begins with the word “else”, it is a predicate that is the negation of the preceding “if” predicate step at the same level.

A step may specify the iterative application of its substeps.

A step may assert an invariant condition of its algorithm. Such assertions are used to make explicit algorithmic invariants that would otherwise be implicit. Such assertions add no additional semantic requirements and hence need not be checked by an implementation. They are used simply to clarify algorithms.

Mathematical operations such as addition, subtraction, negation, multiplication, division, and the mathematical functions defined later in this clause should always be understood as computing exact mathematical results on mathematical real numbers, which do not include infinities and do not include a negative zero that is distinguished from positive zero. Algorithms in this standard that model floating-point arithmetic include explicit steps, where necessary, to handle infinities and signed zero and to perform rounding. If a mathematical operation or function is applied to a floating-point number, it should be understood as being applied to the exact mathematical value represented by that floating-point number; such a floating-point number must be finite, and if it is  $+0$  or  $-0$  then the corresponding mathematical value is simply  $0$ .

The mathematical function  $\text{abs}(x)$  yields the absolute value of  $x$ , which is  $-x$  if  $x$  is negative (less than zero) and otherwise is  $x$  itself.

The mathematical function  $\text{sign}(x)$  yields 1 if  $x$  is positive and -1 if  $x$  is negative. The sign function is not used in this standard for cases when  $x$  is zero.

The notation “ $x$  modulo  $y$ ” ( $y$  must be finite and nonzero) computes a value  $k$  of the same sign as  $y$  (or zero) such that  $\text{abs}(k) < \text{abs}(y)$  and  $x - k = q \times y$  for some integer  $q$ .

The mathematical function  $\text{floor}(x)$  yields the largest integer (closest to positive infinity) that is not larger than  $x$ .

NOTE  $\text{floor}(x) = x - (x \text{ modulo } 1)$ .

If an algorithm is defined to “throw an exception”, execution of the algorithm is terminated and no result is returned. The calling algorithms are also terminated, until an algorithm step is reached that explicitly deals with the exception, using terminology such as “If an exception was thrown...”. Once such an algorithm step has been encountered the exception is no longer considered to have occurred.

### 5.3 Static Semantic Rules

Context-free grammars are not sufficiently powerful to express all the rules that define whether a stream of input elements make up a valid ECMAScript program that may be evaluated. In some situations additional rules are needed that may be expressed using either ECMAScript algorithm conventions or prose requirements. Such rules are always associated with a production of a grammar and are called the static semantics of the production.

Static Semantic Rules have names and typically are defined using an algorithm. Such algorithms may be parameterized. Named Static Semantic Rules are associated with grammar productions and a production that has multiple alternative definitions will typically have for each alternative a distinct algorithm for each applicable named static semantic rule.

Unless otherwise specified every grammar production alternative in this specification implicitly has a definition for a static semantic rule name `Contains` which takes an argument named `symbol` whose value is a terminal or non-terminal of the grammar that includes the associated production. The default definition of `Contains` is:

1. For each terminal and non-terminal grammar symbol `sym`, in the definition of this production do
  - a. If `sym` is the same grammar symbol as `symbol`, return `true`.
  - b. If `sym` is a non-terminal, let `contained` be the result of `Contains` for `sym` with argument `symbol`.
  - c. If `contained` is `true`, return `true`.
2. Return `false`.

The above definition is explicitly over-ridden for specific productions.

A special kind of static semantic rule is an Early Error Rule. Early error rules define early error conditions (see clause 16) that are associate with specific grammar productions. Evaluation of most early error rules are not explicitly invoked within the algorithms of this specification. A conforming implementation must, prior to the first evaluation of a Program, validate all of the early error rules of the productions used to parse that Program. If any of the early error rules are violated the Program is invalid and can not be evaluated.

**Deleted:** An

**Deleted:** static semantic

**Deleted:** used to

**Deleted:** a

**Deleted:** prior to the first evaluation of that Program

**Comment [AW1]:** Perhaps this should be somewhere else. Currently we don't have a section that enumerates all the steps in loading and evaluating a program.

**Deleted:** static semantic

**Deleted:** Static semantic rule violations are early errors (see clause 16) and reported in the same manner as syntax errors.

## 6 Source Text

ECMAScript source text is represented as a sequence of characters in the Unicode character encoding, version 3.0 or later. The text is expected to have been normalised to Unicode Normalization Form C (canonical composition), as described in Unicode Technical Report #15. Conforming ECMAScript implementations are not required to perform any normalisation of text, or behave as though they were performing normalisation of text, themselves. ECMAScript source text is assumed to be a sequence of 16-bit code units for the purposes of this specification. Such a source text may include sequences of 16-bit code units that are not valid UTF-16 character encodings. If an actual source text is encoded in a form other than 16-bit code units it must be processed as if it was first converted to UTF-16.

### Syntax

```
SourceCharacter ::=
  any Unicode code unit
```

Throughout the rest of this document, the phrase “code unit” and the word “character” will be used to refer to a 16-bit unsigned value used to represent a single 16-bit unit of text. The phrase “Unicode character” will be used to refer to the abstract linguistic or typographical unit represented by a single Unicode scalar value (which may be longer than 16 bits and thus may be represented by more than one code unit). The phrase “code point” refers to such a Unicode scalar value. “Unicode character” only refers to entities represented by single Unicode scalar values: the components of a combining character sequence are still individual “Unicode characters,” even though a user might think of the whole sequence as a single character.

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

**NOTE** Although this document sometimes refers to a “transformation” between a “character” within a “string” and the 16-bit unsigned integer that is the code unit of that character, there is actually no transformation because a “character” within a “string” is actually represented using that 16-bit unsigned value.

ECMAScript differs from the Java programming language in the behaviour of Unicode escape sequences. In a Java program, if the Unicode escape sequence \u000A, for example, occurs within a single-line comment, it is interpreted as a line terminator (Unicode character 000A is line feed) and therefore the next character is not part of the comment. Similarly, if the Unicode escape sequence \u000A occurs within a string literal in a Java program, it is likewise interpreted as a line terminator, which is not allowed within a string literal—one must write \n instead of \u000A to cause a line feed to be part of the string value of a string literal. In an ECMAScript program, a Unicode escape sequence occurring within a comment is never interpreted and therefore cannot contribute to termination of the comment. Similarly, a Unicode escape sequence occurring within a string literal in an ECMAScript program always contributes a character to the String value of the literal and is never interpreted as a line terminator or as a quote mark that might terminate the string literal.

## 7 Lexical Conventions

The source text of an ECMAScript program is first converted into a sequence of input elements, which are tokens, line terminators, comments, or white space. The source text is scanned from left to right, repeatedly taking the longest possible sequence of characters as the next input element.

There are two goal symbols for the lexical grammar. The *InputElementDiv* symbol is used in those syntactic grammar contexts where a leading division (/) or division-assignment (/=) operator is permitted. The *InputElementRegExp* symbol is used in other syntactic grammar contexts.

**NOTE** There are no syntactic grammar contexts where both a leading division or division-assignment, and a leading *RegularExpressionLiteral* are permitted. This is not affected by semicolon insertion (see 7.9); in examples such as the following:

```
a = b
/hi/g.exec(c).map(d);
```

where the first non-whitespace, non-comment character after a *LineTerminator* is slash (/) and the syntactic context allows division or division-assignment, no semicolon is inserted at the *LineTerminator*. That is, the above example is interpreted in the same way as:

```
a = b / hi / g.exec(c).map(d);
```

### Syntax

```
InputElementDiv ::=
WhiteSpace
LineTerminator
Comment
Token
DivPunctuator
```

*InputElementRegExp ::*  
*WhiteSpace*  
*LineTerminator*  
*Comment*  
*Token*  
*RegularExpressionLiteral*

## 7.1 Unicode Format-Control Characters

The Unicode format-control characters (i.e., the characters in category “Cf” in the Unicode Character Database such as LEFT-TO-RIGHT MARK or RIGHT-TO-LEFT MARK) are control codes used to control the formatting of a range of text in the absence of higher-level protocols for this (such as mark-up languages).

It is useful to allow format-control characters in source text to facilitate editing and display. All format control characters may be used within comments, and within string literals and regular expression literals.

<ZWNJ> and <ZWJ> are format-control characters that are used to make necessary distinctions when forming words or phrases in certain languages. In ECMAScript source text, <ZWNJ> and <ZWJ> may also be used in an identifier after the first character.

<BOM> is a format-control character used primarily at the start of a text to mark it as Unicode and to allow detection of the text's encoding and byte order. <BOM> characters intended for this purpose can sometimes also appear after the start of a text, for example as a result of concatenating files. <BOM> characters are treated as white space characters (see 7.2).

The special treatment of certain format-control characters outside of comments, string literals, and regular expression literals is summarised in Table 1.

Table 1 — Format-Control Character Usage

Code Unit Value	Name	Formal Name	Usage
\u200C	Zero width non-joiner	<ZWNJ>	<i>IdentifierPart</i>
\u200D	Zero width joiner	<ZWJ>	<i>IdentifierPart</i>
\uFEFF	Byte Order Mark	<BOM>	<i>Whitespace</i>

## 7.2 White Space

White space characters are used to improve source text readability and to separate tokens (indivisible lexical units) from each other, but are otherwise insignificant. White space characters may occur between any two tokens and at the start or end of input. White space characters may also occur within a *StringLiteral* or a *RegularExpressionLiteral* (where they are considered significant characters forming part of the literal value) or within a *Comment*, but cannot appear within any other kind of token.

The ECMAScript white space characters are listed in Table 2.

Table 2 — Whitespace Characters

Code Unit Value	Name	Formal Name
\u0009	Tab	<TAB>
\u000B	Vertical Tab	<VT>
\u000C	Form Feed	<FF>
\u0020	Space	<SP>
\u00A0	No-break space	<NBSP>
\uFEFF	Byte Order Mark	<BOM>
Other category “Zs”	Any other Unicode “space separator”	<USP>

Formatted: Not Small caps, All caps

Formatted: Not Small caps, All caps

ECMAScript implementations must recognise all of the white space characters defined in Unicode 3.0. Later editions of the Unicode Standard may define other white space characters. ECMAScript implementations may recognise white space characters from later editions of the Unicode Standard.

## Syntax

*WhiteSpace* ::

- <TAB>
- <VT>
- <FF>
- <SP>
- <NBSP>
- <BOM>
- <USP>

### 7.3 Line Terminators

Like white space characters, line terminator characters are used to improve source text readability and to separate tokens (indivisible lexical units) from each other. However, unlike white space characters, line terminators have some influence over the behaviour of the syntactic grammar. In general, line terminators may occur between any two tokens, but there are a few places where they are forbidden by the syntactic grammar. Line terminators also affect the process of automatic semicolon insertion (7.9). A line terminator cannot occur within any token except a *StringLiteral*. Line terminators may only occur within a *StringLiteral* token as part of a *LineContinuation*.

A line terminator can occur within a *MultiLineComment* (7.4) but cannot occur within a *SingleLineComment*.

Line terminators are included in the set of white space characters that are matched by the \s class in regular expressions.

The ECMAScript line terminator characters are listed in Table 3.

**Table 3 — Line Terminator Characters**

<b>Code Unit Value</b>	<b>Name</b>	<b>Formal Name</b>
\u000A	Line Feed	<LF>
\u000D	Carriage Return	<CR>
\u2028	Line separator	<LS>
\u2029	Paragraph separator	<PS>

Only the characters in Table 3 are treated as line terminators. Other new line or line breaking characters are treated as white space but not as line terminators. The character sequence <CR><LF> is commonly used as a line terminator. It should be considered a single character for the purpose of reporting line numbers.

## Syntax

*LineTerminator* ::

- <LF>
- <CR>
- <LS>
- <PS>

*LineTerminatorSequence* ::

- <LF>
- <CR> [lookahead  $\notin$  <LF>]
- <LS>
- <PS>
- <CR><LF>

## 7.4 Comments

Comments can be either single or multi-line. Multi-line comments cannot nest.

Because a single-line comment can contain any character except a *LineTerminator* character, and because of the general rule that a token is always as long as possible, a single-line comment always consists of all characters from the `//` marker to the end of the line. However, the *LineTerminator* at the end of the line is not considered to be part of the single-line comment; it is recognised separately by the lexical grammar and becomes part of the stream of input elements for the syntactic grammar. This point is very important, because it implies that the presence or absence of single-line comments does not affect the process of automatic semicolon insertion (see 7.9).

Comments behave like white space and are discarded except that, if a *MultiLineComment* contains a line terminator character, then the entire comment is considered to be a *LineTerminator* for purposes of parsing by the syntactic grammar.

### Syntax

```

Comment :: 
    MultiLineComment
    SingleLineComment

MultiLineComment :: 
    /* MultiLineCommentCharsopt */

MultiLineCommentChars :: 
    MultiLineNotAsteriskChar MultiLineCommentCharsopt
    * PostAsteriskCommentCharsopt

PostAsteriskCommentChars :: 
    MultiLineNotForwardSlashOrAsteriskChar MultiLineCommentCharsopt
    * PostAsteriskCommentCharsopt

MultiLineNotAsteriskChar :: 
    SourceCharacter but not *

MultiLineNotForwardSlashOrAsteriskChar :: 
    SourceCharacter but not one of / or *

SingleLineComment :: 
    // SingleLineCommentCharsopt

SingleLineCommentChars :: 
    SingleLineCommentChar SingleLineCommentCharsopt

SingleLineCommentChar :: 
    SourceCharacter but not LineTerminator

```

## 7.5 Tokens

### Syntax

```

Token :: 
    IdentifierName
    Punctuator
    NumericLiteral
    StringLiteral

```

**NOTE** The *DivPunctuator* and *RegularExpressionLiteral* productions define tokens, but are not included in the *Token* production.

## 7.6 Identifier Names and Identifiers

*Identifier*, *IdentifierName*, and *ReservedWord* are tokens that are interpreted according to the grammar given in the “Identifiers” section of chapter 5 of the Unicode standard, with some small modifications. An *Identifier* is an *IdentifierName* that is not a *ReservedWord* (see 7.6.1). The Unicode identifier grammar is based on both normative and informative character categories specified by the Unicode Standard. The characters in the specified categories in version 3.0 of the Unicode standard must be treated as in those categories by all conforming ECMAScript implementations.

This standard specifies specific character additions: The dollar sign (\$) and the underscore (\_) are permitted anywhere in an *IdentifierName*.

Unicode escape sequences are also permitted in an *IdentifierName*, where they contribute a single character to the *IdentifierName*, as computed by the CV of the *UnicodeEscapeSequence* (see 7.8.4). The \ preceding the *UnicodeEscapeSequence* does not contribute a character to the *IdentifierName*. A *UnicodeEscapeSequence* cannot be used to put a character into an *IdentifierName* that would otherwise be illegal. In other words, if a \ *UnicodeEscapeSequence* sequence were replaced by its *UnicodeEscapeSequence*'s CV, the result must still be a valid *IdentifierName* that has the exact same sequence of characters as the original *IdentifierName*. All interpretations of *IdentifierName* within this specification are based upon their actual characters regardless of whether or not an escape sequence was used to contribute any particular characters.

Two *IdentifierName* that are canonically equivalent according to the Unicode standard are *not* equal unless they are represented by the exact same sequence of code units (in other words, conforming ECMAScript implementations are only required to do bitwise comparison on *IdentifierName* values). The intent is that the incoming source text has been converted to normalised form C before it reaches the compiler.

ECMAScript implementations may recognise identifier characters defined in later editions of the Unicode Standard. If portability is a concern, programmers should only employ identifier characters defined in Unicode 3.0.

### Syntax

*Identifier* ::  
  *IdentifierName* **but not** *ReservedWord*

*IdentifierName* ::  
  *IdentifierStart*  
  *IdentifierName* *IdentifierPart*

*IdentifierStart* ::  
  *UnicodeLetter*  
  \$  
  \ *UnicodeEscapeSequence*

*IdentifierPart* ::  
  *IdentifierStart*  
  *UnicodeCombiningMark*  
  *UnicodeDigit*  
  *UnicodeConnectorPunctuation*  
  <ZWNJ>  
  <ZWJ>

*UnicodeLetter* ::  
  any character in the Unicode categories “Uppercase letter (Lu)”, “Lowercase letter (Ll)”, “Titlecase letter (Lt)”, “Modifier letter (Lm)”, “Other letter (Lo)”, or “Letter number (Ni)”.

*UnicodeCombiningMark* ::  
  any character in the Unicode categories “Non-spacing mark (Mn)” or “Combining spacing mark (Mc)”

**Formatted:** Font: Times New Roman, Italic  
**Deleted:** Identifier Names  
**Formatted:** Font: Times New Roman, Italic  
**Formatted:** Font: (Default) Arial  
**Formatted:** Font: Times New Roman, Italic

**Deleted:** identifiers

*UnicodeDigit* ::  
any character in the Unicode category “Decimal number (Nd)”

*UnicodeConnectorPunctuation* ::  
any character in the Unicode category “Connector punctuation (Pc)”

The definitions of the nonterminal *UnicodeEscapeSequence* is given in 7.8.4

#### **Static Semantics: String Value**

*Identifier* :: *IdentifierName* but not *ReservedWord*

1. Return the String Value of *IdentifierName*.

*IdentifierName* ::

*IdentifierStart*

*IdentifierName IdentifierPart*

1. Return the String value consisting of the sequence of characters corresponding to *IdentifierName*.

#### **7.6.1 Reserved Words**

A reserved word is an *IdentifierName* that cannot be used as an *Identifier*.

##### **Syntax**

*ReservedWord* ::  
*Keyword*  
*FutureReservedWord*  
*NullLiteral*  
*BooleanLiteral*

##### **7.6.1.1 Keywords**

The following tokens are ECMAScript keywords and may not be used as *Identifiers* in ECMAScript programs.

##### **Syntax**

*Keyword* :: one of

break	do	instanceof	typeof
case	else	new	var
catch	finally	return	void
continue	for	switch	while
debugger	function	this	with
default	if	throw	
delete	in	try	

**Deleted:** Semantics¶

The String value of the production

**Deleted:** is determined as follows:

**Deleted:** v

**Deleted:** consisting

**Deleted:** The String value of *IdentifierName* is determined as follows:

##### **7.6.1.2 Future Reserved Words**

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

##### **Syntax**

*FutureReservedWord* :: one of

class	enum	extends	super
const	export	import	

**Formatted Table**

**Formatted Table**

The following tokens are also considered to be *FutureReservedWords* when they occur within strict mode code (see 10.1.1). The occurrence of any of these tokens within strict mode code in any context where the occurrence of a *FutureReservedWord* would produce an error must also produce an equivalent error:

<code>implements</code>	<code>let</code>	<code>private</code>	<code>public</code>	<code>yield</code>
<code>interface</code>	<code>package</code>	<code>protected</code>	<code>static</code>	

► Formatted Table

## 7.7 Punctuators

### Syntax

*Punctuator* :: one of

{	}	(	)	[	]
.	;	,	<	>	<=
<code>&gt;=</code>	<code>==</code>	<code>!=</code>	<code>====</code>	<code>!==</code>	
<code>+</code>	<code>-</code>	<code>*</code>	<code>%</code>	<code>++</code>	<code>--</code>
<code>&lt;&lt;</code>	<code>&gt;&gt;</code>	<code>&gt;&gt;&gt;</code>	<code>&amp;</code>	<code> </code>	<code>^</code>
!	<code>~</code>	<code>&amp;&amp;</code>	<code>  </code>	<code>?</code>	<code>:</code>
=	<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>%=</code>	<code>&lt;&lt;=</code>
<code>&gt;&gt;=</code>	<code>&gt;&gt;&gt;=</code>	<code>&amp;=</code>	<code> =</code>	<code>^=</code>	<code>&lt; =</code>

*DivPunctuator* :: one of

/	/=
---	----

► Formatted Table

## 7.8 Literals

### Syntax

*Literal* ::

*NullLiteral*  
*ValueLiteral*

*ValueLiteral* ::

*BooleanLiteral*  
*NumericLiteral*  
*StringLiteral*  
*RegularExpressionLiteral*

► Deleted:

### 7.8.1 Null Literals

#### Syntax

*NullLiteral* ::  
  `null`

#### Semantics

The value of the null literal `null` is the sole value of the Null type, namely `null`.

## 7.8.2 Boolean Literals

### Syntax

*BooleanLiteral* ::  
**true**  
**false**

### Semantics

The value of the Boolean literal **true** is a value of the Boolean type, namely **true**.

The value of the Boolean literal **false** is a value of the Boolean type, namely **false**.

## 7.8.3 Numeric Literals

### Syntax

*NumericLiteral* ::  
*DecimalLiteral*  
*BinaryIntegerLiteral*  
*OctalIntegerLiteral*  
*HexIntegerLiteral*

*DecimalLiteral* ::  
*DecimalIntegerLiteral* . *DecimalDigits*<sub>opt</sub> *ExponentPart*<sub>opt</sub>  
. *DecimalDigits* *ExponentPart*<sub>opt</sub>  
*DecimalIntegerLiteral* *ExponentPart*<sub>opt</sub>

*DecimalIntegerLiteral* ::  
0  
*NonZeroDigit* *DecimalDigits*<sub>opt</sub>

*DecimalDigits* ::  
*DecimalDigit*  
*DecimalDigits* *DecimalDigit*

*DecimalDigit* :: **one of**  
0 1 2 3 4 5 6 7 8 9

*NonZeroDigit* :: **one of**  
1 2 3 4 5 6 7 8 9

*ExponentPart* ::  
*ExponentIndicator* *SignedInteger*

*ExponentIndicator* :: **one of**  
e E

*SignedInteger* ::  
*DecimalDigits*  
+ *DecimalDigits*  
- *DecimalDigits*

*BinaryIntegerLiteral* ::  
0b BinaryDigit  
0B BinaryDigit  
*BinaryIntegerLiteral* BinaryDigit

*BinaryDigit* :: **one of**  
0 1

**Comment [AWB72]:** From March 29 meeting notes: Hex floating point literals:  
Waldemar: Other languages include these things. They're rarely used but when you want one, you really want one. Use cases are similar to that of hex literals.  
Will explore adding them.  
MarkM: 0x3.p1 currently evaluates to undefined. This would be a breaking change.  
Waldemar: Not clear anyone would notice. How did other languages deal with this?

**Comment [AWB73]:** The various Digit productions could be refactored to have less redundancy

*OctalIntegerLiteral ::*  
*0o OctalDigit*  
*0o OctalDigit*  
*OctalIntegerLiteral OctalDigit*

*OctalDigit :: one of*  
*0 1 2 3 4 5 6 7*

*HexIntegerLiteral ::*  
*0x HexDigit*  
*0X HexDigit*  
*HexIntegerLiteral HexDigit*

*HexDigit :: one of*  
*0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F*

The source character immediately following a *NumericLiteral* must not be an *IdentifierStart* or *DecimalDigit*.

NOTE For example:  
3in

is an error and not the two input elements 3 and in.

## Semantics

A numeric literal stands for a value of the *Number* type. This value is determined in two steps: first, a mathematical value (MV) is derived from the literal; second, this mathematical value is rounded as described below.

- The MV of *NumericLiteral :: DecimalLiteral* is the MV of *DecimalLiteral*.
- The MV of *NumericLiteral :: HexIntegerLiteral* is the MV of *HexIntegerLiteral*.
- The MV of *DecimalLiteral :: DecimalIntegerLiteral .* is the MV of *DecimalIntegerLiteral*.
- The MV of *DecimalLiteral :: DecimalIntegerLiteral . DecimalDigits* is the MV of *DecimalIntegerLiteral* plus (the MV of *DecimalDigits* times  $10^{-n}$ ), where *n* is the number of characters in *DecimalDigits*.
- The MV of *DecimalLiteral :: DecimalIntegerLiteral . ExponentPart* is the MV of *DecimalIntegerLiteral* times  $10^e$ , where *e* is the MV of *ExponentPart*.
- The MV of *DecimalLiteral :: DecimalIntegerLiteral . DecimalDigits ExponentPart* is (the MV of *DecimalIntegerLiteral* plus (the MV of *DecimalDigits* times  $10^{-n}$ )) times  $10^e$ , where *n* is the number of characters in *DecimalDigits* and *e* is the MV of *ExponentPart*.
- The MV of *DecimalLiteral :: . DecimalDigits* is the MV of *DecimalDigits* times  $10^{-n}$ , where *n* is the number of characters in *DecimalDigits*.
- The MV of *DecimalLiteral :: . DecimalDigits ExponentPart* is the MV of *DecimalDigits* times  $10^{e-n}$ , where *n* is the number of characters in *DecimalDigits* and *e* is the MV of *ExponentPart*.
- The MV of *DecimalLiteral :: . DecimalIntegerLiteral* is the MV of *DecimalIntegerLiteral*.
- The MV of *DecimalLiteral :: . DecimalIntegerLiteral ExponentPart* is the MV of *DecimalIntegerLiteral* times  $10^e$ , where *e* is the MV of *ExponentPart*.
- The MV of *DecimalIntegerLiteral :: 0* is 0.
- The MV of *DecimalIntegerLiteral :: NonZeroDigit* is the MV of *NonZeroDigit*.
- The MV of *DecimalIntegerLiteral :: NonZeroDigit DecimalDigits* is (the MV of *NonZeroDigit* times  $10^n$ ) plus the MV of *DecimalDigits*, where *n* is the number of characters in *DecimalDigits*.
- The MV of *DecimalDigits :: DecimalDigit* is the MV of *DecimalDigit*.
- The MV of *DecimalDigits :: DecimalDigits DecimalDigit* is (the MV of *DecimalDigits* times 10) plus the MV of *DecimalDigit*.
- The MV of *ExponentPart :: ExponentIndicator SignedInteger* is the MV of *SignedInteger*.
- The MV of *SignedInteger :: DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger :: + DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger :: - DecimalDigits* is the negative of the MV of *DecimalDigits*.

- The MV of *DecimalDigit* :: 0 or of *HexDigit* :: 0 is 0.
- The MV of *DecimalDigit* :: 1 or of *NonZeroDigit* :: 1 or of *HexDigit* :: 1 is 1.
- The MV of *DecimalDigit* :: 2 or of *NonZeroDigit* :: 2 or of *HexDigit* :: 2 is 2.
- The MV of *DecimalDigit* :: 3 or of *NonZeroDigit* :: 3 or of *HexDigit* :: 3 is 3.
- The MV of *DecimalDigit* :: 4 or of *NonZeroDigit* :: 4 or of *HexDigit* :: 4 is 4.
- The MV of *DecimalDigit* :: 5 or of *NonZeroDigit* :: 5 or of *HexDigit* :: 5 is 5.
- The MV of *DecimalDigit* :: 6 or of *NonZeroDigit* :: 6 or of *HexDigit* :: 6 is 6.
- The MV of *DecimalDigit* :: 7 or of *NonZeroDigit* :: 7 or of *HexDigit* :: 7 is 7.
- The MV of *DecimalDigit* :: 8 or of *NonZeroDigit* :: 8 or of *HexDigit* :: 8 is 8.
- The MV of *DecimalDigit* :: 9 or of *NonZeroDigit* :: 9 or of *HexDigit* :: 9 is 9.
- The MV of *NumericLiteral* :: *BinaryIntegerLiteral* is the MV of *BinaryIntegerLiteral*.
- The MV of *BinaryDigit* :: 0 is 0.
- The MV of *BinaryDigit* :: 1 is 1.
- The MV of *BinaryIntegerLiteral* :: 0o *BinaryDigit* is the MV of *BinaryDigit*.
- The MV of *BinaryIntegerLiteral* :: 00 *BinaryDigit* is the MV of *BinaryDigit*.
- The MV of *BinaryIntegerLiteral* :: *BinaryIntegerLiteral* *BinaryDigit* is (the MV of *BinaryIntegerLiteral* times 2) plus the MV of *BinaryDigit*.
- The MV of *NumericLiteral* :: *OctalIntegerLiteral* is the MV of *OctalIntegerLiteral*.
- The MV of *OctalDigit* :: 0 is 0.
- The MV of *OctalDigit* :: 1 is 1.
- The MV of *OctalDigit* :: 2 is 2.
- The MV of *OctalDigit* :: 3 is 3.
- The MV of *OctalDigit* :: 4 is 4.
- The MV of *OctalDigit* :: 5 is 5.
- The MV of *OctalDigit* :: 6 is 6.
- The MV of *OctalDigit* :: 7 is 7.
- The MV of *OctalIntegerLiteral* :: 0o *OctalDigit* is the MV of *OctalDigit*.
- The MV of *OctalIntegerLiteral* :: 00 *OctalDigit* is the MV of *OctalDigit*.
- The MV of *OctalIntegerLiteral* :: *OctalIntegerLiteral* *OctalDigit* is (the MV of *OctalIntegerLiteral* times 8) plus the MV of *OctalDigit*.
- The MV of *HexDigit* :: a or of *HexDigit* :: A is 10.
- The MV of *HexDigit* :: b or of *HexDigit* :: B is 11.
- The MV of *HexDigit* :: c or of *HexDigit* :: C is 12.
- The MV of *HexDigit* :: d or of *HexDigit* :: D is 13.
- The MV of *HexDigit* :: e or of *HexDigit* :: E is 14.
- The MV of *HexDigit* :: f or of *HexDigit* :: F is 15.
- The MV of *HexIntegerLiteral* :: 0x *HexDigit* is the MV of *HexDigit*.
- The MV of *HexIntegerLiteral* :: 0X *HexDigit* is the MV of *HexDigit*.
- The MV of *HexIntegerLiteral* :: *HexIntegerLiteral* *HexDigit* is (the MV of *HexIntegerLiteral* times 16) plus the MV of *HexDigit*.

Once the exact MV for a numeric literal has been determined, it is then rounded to a value of the Number type. If the MV is 0, then the rounded value is +0; otherwise, the rounded value must be the Number value for the MV (as specified in 8.5), unless the literal is a *DecimalLiteral* and the literal has more than 20 significant digits, in which case the Number value may be either the Number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit or the Number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit and then incrementing the literal at the 20th significant digit position. A digit is *significant* if it is not part of an *ExponentPart* and

- it is not 0; or
- there is a nonzero digit to its left and there is a nonzero digit, not in the *ExponentPart*, to its right.

A conforming implementation, when processing strict mode code (see 10.1.1), must not extend the syntax of *NumericLiteral* to include *OctalIntegerLiteral* as described in B.1.1.

#### 7.8.4 String Literals

A string literal is zero or more characters enclosed in single or double quotes. Each character may be represented by an escape sequence. All characters may appear literally in a string literal except for the closing quote character, backslash, carriage return, line separator, paragraph separator, and line feed. Any character may appear in the form of an escape sequence.

##### Syntax

```

StringLiteral :::
  " DoubleStringCharactersopt "
  ' SingleStringCharactersopt '

DoubleStringCharacters :::
  DoubleStringCharacter DoubleStringCharactersopt

SingleStringCharacters :::
  SingleStringCharacter SingleStringCharactersopt

DoubleStringCharacter :::
  SourceCharacter but not one of " or \ or LineTerminator
  \ EscapeSequence
  LineContinuation

SingleStringCharacter :::
  SourceCharacter but not one of ' or \ or LineTerminator
  \ EscapeSequence
  LineContinuation

LineContinuation :::
  \ LineTerminatorSequence

EscapeSequence :::
  CharacterEscapeSequence
  0 [lookahead  $\notin$  DecimalDigit]
  HexEscapeSequence
  UnicodeEscapeSequence

CharacterEscapeSequence :::
  SingleEscapeCharacter
  NonEscapeCharacter

SingleEscapeCharacter :: one of
  ' " \ b f n r t v

NonEscapeCharacter :::
  SourceCharacter but not one of EscapeCharacter or LineTerminator

EscapeCharacter :::
  SingleEscapeCharacter
  DecimalDigit
  x
  u

HexEscapeSequence :::
  x HexDigit HexDigit

```

*UnicodeEscapeSequence* ::=  
  u HexDigit HexDigit HexDigit HexDigit

The definition of the nonterminal *HexDigit* is given in 7.8.3. *SourceCharacter* is defined in clause 6.

### Semantics

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

- The SV of *StringLiteral* :: "" is the empty character sequence.
- The SV of *StringLiteral* :: ' ' is the empty character sequence.
- The SV of *StringLiteral* :: " *DoubleStringCharacters* " is the SV of *DoubleStringCharacters*.
- The SV of *StringLiteral* :: ' *SingleStringCharacters* ' is the SV of *SingleStringCharacters*.
- The SV of *DoubleStringCharacters* :: *DoubleStringCharacter* is a sequence of one character, the CV of *DoubleStringCharacter*.
- The SV of *DoubleStringCharacters* :: *DoubleStringCharacter DoubleStringCharacters* is a sequence of the CV of *DoubleStringCharacter* followed by all the characters in the SV of *DoubleStringCharacters* in order.
- The SV of *SingleStringCharacters* :: *SingleStringCharacter* is a sequence of one character, the CV of *SingleStringCharacter*.
- The SV of *SingleStringCharacters* :: *SingleStringCharacter SingleStringCharacters* is a sequence of the CV of *SingleStringCharacter* followed by all the characters in the SV of *SingleStringCharacters* in order.
- The SV of *LineContinuation* :: \ *LineTerminatorSequence* is the empty character sequence.
- The CV of *DoubleStringCharacter* :: *SourceCharacter but not one of " or \ or LineTerminator* is the *SourceCharacter* character itself.
- The CV of *DoubleStringCharacter* :: \ *EscapeSequence* is the CV of the *EscapeSequence*.
- The CV of *DoubleStringCharacter* :: *LineContinuation* is the empty character sequence.
- The CV of *SingleStringCharacter* :: *SourceCharacter but not one of ' or \ or LineTerminator* is the *SourceCharacter* character itself.
- The CV of *SingleStringCharacter* :: \ *EscapeSequence* is the CV of the *EscapeSequence*.
- The CV of *SingleStringCharacter* :: *LineContinuation* is the empty character sequence.
- The CV of *EscapeSequence* :: *CharacterEscapeSequence* is the CV of the *CharacterEscapeSequence*.
- The CV of *EscapeSequence* :: 0 [lookahead ≠ *DecimalDigit*] is a <NUL> character (Unicode value 0000).
- The CV of *EscapeSequence* :: *HexEscapeSequence* is the CV of the *HexEscapeSequence*.
- The CV of *EscapeSequence* :: *UnicodeEscapeSequence* is the CV of the *UnicodeEscapeSequence*.
- The CV of *CharacterEscapeSequence* :: *SingleEscapeCharacter* is the character whose code unit value is determined by the *SingleEscapeCharacter* according to Table 4:

Table 4 — String Single Character Escape Sequences

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

Formatted Table

- The CV of *CharacterEscapeSequence* :: *NonEscapeCharacter* is the CV of the *NonEscapeCharacter*.
- The CV of *NonEscapeCharacter* :: *SourceCharacter* **but not one of** *EscapeCharacter* **or** *LineTerminator* is the *SourceCharacter* character itself.
- The CV of *HexEscapeSequence* :: **x** *HexDigit* *HexDigit* is the character whose code unit value is (16 times the MV of the first *HexDigit*) plus the MV of the second *HexDigit*.
- The CV of *UnicodeEscapeSequence* :: **u** *HexDigit* *HexDigit* *HexDigit* *HexDigit* is the character whose code unit value is (4096 times the MV of the first *HexDigit*) plus (256 times the MV of the second *HexDigit*) plus (16 times the MV of the third *HexDigit*) plus the MV of the fourth *HexDigit*.

A conforming implementation, when processing strict mode code (see 10.1.1), may not extend the syntax of *EscapeSequence* to include *OctalEscapeSequence* as described in B.1.2.

**NOTE** A line terminator character cannot appear in a string literal, except as part of a *LineContinuation* to produce the empty character sequence. The correct way to cause a line terminator character to be part of the String value of a string literal is to use an escape sequence such as `\n` or `\u000A`.

### 7.8.5 Regular Expression Literals

A regular expression literal is an input element that is converted to a *RegExp* object (see 15.10) each time the literal is evaluated. Two regular expression literals in a program evaluate to regular expression objects that never compare as === to each other even if the two literals' contents are identical. A *RegExp* object may also be created at runtime by `new RegExp` (see 15.10.4) or calling the `RegExp` constructor as a function (15.10.3).

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

#### Syntax

*RegularExpressionLiteral* ::  
  / *RegularExpressionBody* / *RegularExpressionFlags*

*RegularExpressionBody* ::  
  *RegularExpressionFirstChar* *RegularExpressionChars*

*RegularExpressionChars* ::  
  [empty]  
  *RegularExpressionChars* *RegularExpressionChar*

*RegularExpressionFirstChar* ::  
  *RegularExpressionNonTerminator* **but not one of** \* or \ or / or [  
  *RegularExpressionBackslashSequence*  
  *RegularExpressionClass*

*RegularExpressionChar* ::  
  *RegularExpressionNonTerminator* **but not one of** \ or / or [  
  *RegularExpressionBackslashSequence*  
  *RegularExpressionClass*

*RegularExpressionBackslashSequence* ::  
  \ *RegularExpressionNonTerminator*

*RegularExpressionNonTerminator* ::  
  *SourceCharacter* **but not** *LineTerminator*

*RegularExpressionClass* ::  
  [ *RegularExpressionClassChars* ]

*RegularExpressionClassChars* ::

[empty]  
*RegularExpressionClassChars* *RegularExpressionClassChar*

*RegularExpressionClassChar* ::

*RegularExpressionNonTerminator* **but not one of ] or \**  
*RegularExpressionBackslashSequence*

*RegularExpressionFlags* ::

[empty]  
*RegularExpressionFlags* *IdentifierPart*

**NOTE** Regular expression literals may not be empty; instead of representing an empty regular expression literal, the characters // start a single-line comment. To specify an empty regular expression, use: /(?:)/.

## Semantics

A regular expression literal evaluates to a value of the Object type that is an instance of the standard built-in constructor RegExp. This value is determined in two steps: first, the characters comprising the regular expression's *RegularExpressionBody* and *RegularExpressionFlags* production expansions are collected uninterpreted into two Strings Pattern and Flags, respectively. Then each time the literal is evaluated, a new object is created as if by the expression `new RegExp(Pattern, Flags)` where RegExp is the standard built-in constructor with that name. The newly constructed object becomes the value of the *RegularExpressionLiteral*. If the call to `new RegExp` would generate an error as specified in 15.10.4.1, the error must be treated as an early error (Clause 16).

## 7.9 Automatic Semicolon Insertion

Certain ECMAScript statements (empty statement, variable statement, expression statement, `do-while` statement, `continue` statement, `break` statement, `return` statement, and `throw` statement) must be terminated with semicolons. Such semicolons may always appear explicitly in the source text. For convenience, however, such semicolons may be omitted from the source text in certain situations. These situations are described by saying that semicolons are automatically inserted into the source code token stream in those situations.

### 7.9.1 Rules of Automatic Semicolon Insertion

There are three basic rules of semicolon insertion:

1. When, as the program is parsed from left to right, a token (called the *offending token*) is encountered that is not allowed by any production of the grammar, then a semicolon is automatically inserted before the offending token if one or more of the following conditions is true:
  - The offending token is separated from the previous token by at least one *LineTerminator*.
  - The offending token is }.
2. When, as the program is parsed from left to right, the end of the input stream of tokens is encountered and the parser is unable to parse the input token stream as a single complete ECMAScript *Program*, then a semicolon is automatically inserted at the end of the input stream.
3. When, as the program is parsed from left to right, a token is encountered that is allowed by some production of the grammar, but the production is a *restricted production* and the token would be the first token for a terminal or nonterminal immediately following the annotation “[no *LineTerminator* here]” within the restricted production (and therefore such a token is called a restricted token), and the restricted token is separated from the previous token by at least one *LineTerminator*, then a semicolon is automatically inserted before the restricted token.

However, there is an additional overriding condition on the preceding rules: a semicolon is never inserted automatically if the semicolon would then be parsed as an empty statement or if that semicolon would become one of the two semicolons in the header of a `for` statement (see 12.6.3).

NOTE The following are the only restricted productions in the grammar:

*PostfixExpression* :  
*LeftHandSideExpression* [no *LineTerminator* here] **++**  
*LeftHandSideExpression* [no *LineTerminator* here] **--**

*ContinueStatement* :  
**continue** [no *LineTerminator* here] *Identifier* ;

*BreakStatement* :  
**break** [no *LineTerminator* here] *Identifier* ;

*ReturnStatement* :  
**return** [no *LineTerminator* here] *Expression* ;

*ThrowStatement* :  
**throw** [no *LineTerminator* here] *Expression* ;

The practical effect of these restricted productions is as follows:

When a **++** or **--** token is encountered where the parser would treat it as a postfix operator, and at least one *LineTerminator* occurred between the preceding token and the **++** or **--** token, then a semicolon is automatically inserted before the **++** or **--** token.

When a **continue**, **break**, **return**, or **throw** token is encountered and a *LineTerminator* is encountered before the next token, a semicolon is automatically inserted after the **continue**, **break**, **return**, or **throw** token.

The resulting practical advice to ECMAScript programmers is:

A postfix **++** or **--** operator should appear on the same line as its operand.

An *Expression* in a **return** or **throw** statement should start on the same line as the **return** or **throw** token.

An *Identifier* in a **break** or **continue** statement should be on the same line as the **break** or **continue** token.

## 7.9.2 Examples of Automatic Semicolon Insertion

The source

```
{ 1 2 } 3
```

is not a valid sentence in the ECMAScript grammar, even with the automatic semicolon insertion rules. In contrast, the source

```
{ 1  
2 } 3
```

is also not a valid ECMAScript sentence, but is transformed by automatic semicolon insertion into the following:

```
{ 1  
;2 ;} 3;
```

which is a valid ECMAScript sentence.

The source

```
for (a; b  
)
```

is not a valid ECMAScript sentence and is not altered by automatic semicolon insertion because the semicolon is needed for the header of a **for** statement. Automatic semicolon insertion never inserts one of the two semicolons in the header of a **for** statement.

The source

```
return  
a + b
```

is transformed by automatic semicolon insertion into the following:

```
return;  
a + b;
```

**NOTE** The expression `a + b` is not treated as a value to be returned by the `return` statement, because a *LineTerminator* separates it from the token `return`.

The source

```
a = b  
++c
```

is transformed by automatic semicolon insertion into the following:

```
a = b;  
++c;
```

**NOTE** The token `++` is not treated as a postfix operator applying to the variable `b`, because a *LineTerminator* occurs between `b` and `++`.

The source

```
if (a > b)  
else c = d
```

is not a valid ECMAScript sentence and is not altered by automatic semicolon insertion before the `else` token, even though no production of the grammar applies at that point, because an automatically inserted semicolon would then be parsed as an empty statement.

The source

```
a = b + c  
(d + e).print()
```

is *not* transformed by automatic semicolon insertion, because the parenthesised expression that begins the second line can be interpreted as an argument list for a function call:

```
a = b + c(d + e).print()
```

In the circumstance that an assignment statement must begin with a left parenthesis, it is a good idea for the programmer to provide an explicit semicolon at the end of the preceding statement rather than to rely on automatic semicolon insertion.

## 8 Types

Algorithms within this specification manipulate values each of which has an associated type. The possible value types are exactly those defined in this clause. Types are further subclassified into ECMAScript language types and specification types.

An ECMAScript language type corresponds to values that are directly manipulated by an ECMAScript programmer using the ECMAScript language. The ECMAScript language types are Undefined, Null, Boolean, String, Number, and Object.

A specification type corresponds to meta-values that are used within algorithms to describe the semantics of ECMAScript language constructs and ECMAScript language types. The specification types are Reference, List, Completion, Property Descriptor, Property Identifier, Lexical Environment, and Environment Record. Specification type values are specification artefacts that do not necessarily correspond to any specific entity within an ECMAScript implementation. Specification type values may be used to describe intermediate results of ECMAScript expression evaluation but such values cannot be stored as properties of objects or values of ECMAScript language variables.

Within this specification, the notation “Type(*x*)” is used as shorthand for “the type of *x*” where “type” refers to the ECMAScript language and specification types defined in this clause.

## 8.1 The Undefined Type

The Undefined type has exactly one value, called **undefined**. Any variable that has not been assigned a value has the value **undefined**.

## 8.2 The Null Type

The Null type has exactly one value, called **null**.

## 8.3 The Boolean Type

The Boolean type represents a logical entity having two values, called **true** and **false**.

## 8.4 The String Type

The String type is the set of all finite ordered sequences of zero or more 16-bit unsigned integer values (“elements”). The String type is generally used to represent textual data in a running ECMAScript program, in which case each element in the String is treated as a code unit value (see Clause 6). Each element is regarded as occupying a position within the sequence. These positions are indexed with nonnegative integers. The first element (if any) is at position 0, the next element (if any) at position 1, and so on. The length of a String is the number of elements (i.e., 16-bit values) within it. The empty String has length zero and therefore contains no elements.

When a String contains actual textual data, each element is considered to be a single UTF-16 code unit. Whether or not this is the actual storage format of a String, the characters within a String are numbered by their initial code unit element position as though they were represented using UTF-16. All operations on Strings (except as otherwise stated) treat them as sequences of undifferentiated 16-bit unsigned integers; they do not ensure the resulting String is in normalised form, nor do they ensure language-sensitive results.

**NOTE** The rationale behind this design was to keep the implementation of Strings as simple and high-performing as possible. The intent is that textual data coming into the execution environment from outside (e.g., user input, text read from a file or received over the network, etc.) be converted to Unicode Normalised Form C before the running program sees it. Usually this would occur at the same time incoming text is converted from its original character encoding to Unicode (and would impose no additional overhead). Since it is recommended that ECMAScript source code be in Normalised Form C, string literals are guaranteed to be normalised (if source text is guaranteed to be normalised), as long as they do not contain any Unicode escape sequences.

## 8.5 The Number Type

The Number type has exactly  $18437736874454810627$  (that is,  $2^{64}-2^{53}+3$ ) values, representing the double-precision 64-bit format IEEE 754 values as specified in the IEEE Standard for Binary Floating-Point Arithmetic, except that the  $9007199254740990$  (that is,  $2^{53}-2$ ) distinct “Not-a-Number” values of the IEEE Standard are represented in ECMAScript as a single special **NaN** value. (Note that the **NaN** value is produced by the program expression **NaN**.) In some implementations, external code might be able to detect a difference between various Not-a-Number values, but such behaviour is implementation-dependent; to ECMAScript code, all **NaN** values are indistinguishable from each other.

There are two other special values, called **positive Infinity** and **negative Infinity**. For brevity, these values are also referred to for expository purposes by the symbols  $+\infty$  and  $-\infty$ , respectively. (Note that these two infinite Number values are produced by the program expressions **+Infinity** (or simply **Infinity**) and **-Infinity**.)

The other  $18437736874454810624$  (that is,  $2^{64}-2^{53}$ ) values are called the finite numbers. Half of these are positive numbers and half are negative numbers; for every finite positive Number value there is a corresponding negative value having the same magnitude.

Note that there is both a **positive zero** and a **negative zero**. For brevity, these values are also referred to for expository purposes by the symbols  $+0$  and  $-0$ , respectively. (Note that these two different zero Number values are produced by the program expressions **+0** (or simply **0**) and **-0**.)

The 18437736874454810622 (that is,  $2^{64}-2^{53}-2$ ) finite nonzero values are of two kinds:

18428729675200069632 (that is,  $2^{64}-2^{54}$ ) of them are normalised, having the form

$$s \times m \times 2^e$$

where  $s$  is  $+1$  or  $-1$ ,  $m$  is a positive integer less than  $2^{53}$  but not less than  $2^{52}$ , and  $e$  is an integer ranging from  $-1074$  to  $971$ , inclusive.

The remaining 9007199254740990 (that is,  $2^{53}-2$ ) values are denormalised, having the form

$$s \times m \times 2^e$$

where  $s$  is  $+1$  or  $-1$ ,  $m$  is a positive integer less than  $2^{52}$ , and  $e$  is  $-1074$ .

Note that all the positive and negative integers whose magnitude is no greater than  $2^{53}$  are representable in the Number type (indeed, the integer  $0$  has two representations,  $+0$  and  $-0$ ).

A finite number has an *odd significand* if it is nonzero and the integer  $m$  used to express it (in one of the two forms shown above) is odd. Otherwise, it has an *even significand*.

In this specification, the phrase “the Number value for  $x$ ” where  $x$  represents an exact nonzero real mathematical quantity (which might even be an irrational number such as  $\pi$ ) means a Number value chosen in the following manner. Consider the set of all finite values of the Number type, with  $-0$  removed and with two additional values added to it that are not representable in the Number type, namely  $2^{1024}$  (which is  $+1 \times 2^{53} \times 2^{971}$ ) and  $-2^{1024}$  (which is  $-1 \times 2^{53} \times 2^{971}$ ). Choose the member of this set that is closest in value to  $x$ . If two values of the set are equally close, then the one with an even significand is chosen; for this purpose, the two extra values  $2^{1024}$  and  $-2^{1024}$  are considered to have even significands. Finally, if  $2^{1024}$  was chosen, replace it with  $+\infty$ ; if  $-2^{1024}$  was chosen, replace it with  $-\infty$ ; if  $+0$  was chosen, replace it with  $-0$  if and only if  $x$  is less than zero; any other chosen value is used unchanged. The result is the Number value for  $x$ . (This procedure corresponds exactly to the behaviour of the IEEE 754 “round to nearest” mode.)

Some ECMAScript operators deal only with integers in the range  $-2^{31}$  through  $2^{31}-1$ , inclusive, or in the range  $0$  through  $2^{32}-1$ , inclusive. These operators accept any value of the Number type but first convert each such value to one of  $2^{32}$  integer values. See the descriptions of the ToInt32 and ToUInt32 operators in 9.5 and 9.6, respectively.

## 8.6 The Object Type

An Object is a collection of properties. Each property is either a named data property, a named accessor property, or an internal property:

- A *named data property* associates a name with an ECMAScript language value and a set of Boolean attributes.
- A *named accessor property* associates a name with one or two accessor functions, and a set of Boolean attributes. The accessor functions are used to store or retrieve an ECMAScript language value that is associated with the property.
- An *internal property* has no name and is not directly accessible via ECMAScript language operators. Internal properties exist purely for specification purposes.

There are two kinds of access for named (non-internal) properties: *get* and *put*, corresponding to retrieval and assignment, respectively.

### 8.6.1 Property Attributes

Attributes are used in this specification to define and explain the state of named properties. A named data property associates a name with the attributes listed in Table 5

**Table 5 — Attributes of a Named Data Property**

<b>Attribute Name</b>	<b>Value Domain</b>	<b>Description</b>
<code>[[Value]]</code>	Any ECMAScript language type	The value retrieved by reading the property.
<code>[[Writable]]</code>	Boolean	If <b>false</b> , attempts by ECMAScript code to change the property's <code>[[Value]]</code> attribute using <code>[[Put]]</code> will not succeed.
<code>[[Enumerable]]</code>	Boolean	If <b>true</b> , the property will be enumerated by a for-in enumeration (see 12.6.4). Otherwise, the property is said to be non-enumerable.
<code>[[Configurable]]</code>	Boolean	If <b>false</b> , attempts to delete the property, change the property to be an accessor property, or change its attributes (other than <code>[[Value]]</code> ) will fail.

A named accessor property associates a name with the attributes listed in Table 6.

**Table 6 — Attributes of a Named Accessor Property**

<b>Attribute Name</b>	<b>Value Domain</b>	<b>Description</b>
<code>[[Get]]</code>	Object or Undefined	If the value is an Object it must be a function Object. The function's <code>[[Call]]</code> internal method (8.6.2) is called with an empty arguments list to return the property value each time a get access of the property is performed.
<code>[[Set]]</code>	Object or Undefined	If the value is an Object it must be a function Object. The function's <code>[[Call]]</code> internal method (8.6.2) is called with an arguments list containing the assigned value as its sole argument each time a set access of the property is performed. The effect of a property's <code>[[Set]]</code> internal method may, but is not required to, have an effect on the value returned by subsequent calls to the property's <code>[[Get]]</code> internal method.
<code>[[Enumerable]]</code>	Boolean	If <b>true</b> , the property is to be enumerated by a for-in enumeration (see 12.6.4). Otherwise, the property is said to be non-enumerable.
<code>[[Configurable]]</code>	Boolean	If <b>false</b> , attempts to delete the property, change the property to be a data property, or change its attributes will fail.

If the value of an attribute is not explicitly specified by this specification for a named property, the default value defined in Table 7 is used.

**Table 7 — Default Attribute Values**

<b>Attribute Name</b>	<b>Default Value</b>
<code>[[Value]]</code>	<code>undefined</code>
<code>[[Get]]</code>	<code>undefined</code>
<code>[[Set]]</code>	<code>undefined</code>
<code>[[Writable]]</code>	<code>false</code>
<code>[[Enumerable]]</code>	<code>false</code>
<code>[[Configurable]]</code>	<code>false</code>

## 8.6.2 Object Internal Properties and Methods

This specification uses various internal properties to define the semantics of object values. These internal properties are not part of the ECMAScript language. They are defined by this specification purely for expository purposes. An implementation of ECMAScript must behave as if it produced and operated upon internal properties in the manner described here. The names of internal properties are enclosed in double

square brackets [[ ]]. When an algorithm uses an internal property of an object and the object does not implement the indicated internal property, a **TypeError** exception is thrown.

The Table 8 summarises the internal properties used by this specification that are applicable to all ECMAScript objects. The Table 9 summarises the internal properties used by this specification that are only applicable to some ECMAScript objects. The descriptions in these tables indicate their behaviour for native ECMAScript objects, unless stated otherwise in this document for particular kinds of native ECMAScript objects. Host objects may support these internal properties with any implementation-dependent behaviour as long as it is consistent with the specific host object restrictions stated in this document.

The “Value Type Domain” columns of the following tables define the types of values associated with internal properties. The type names refer to the types defined in Clause 8 augmented by the following additional names. “any” means the value may be any ECMAScript language type. “primitive” means Undefined, Null, Boolean, String, or Number. “SpecOp” means the internal property is an internal method, an implementation provided procedure defined by an abstract operation specification. “SpecOp” is followed by a list of descriptive parameter names. If a parameter name is the same as a type name then the name describes the type of the parameter. If a “SpecOp” returns a value, its parameter list is followed by the symbol “→” and the type of the returned value.

**Table 8 — Internal Properties Common to All Objects**

<b>Internal Property</b>	<b>Value Type Domain</b>	<b>Description</b>
[[Prototype]]	Object or Null	The prototype of this object.
[[Extensible]]	Boolean	If <b>true</b> , own properties may be added to the object.
[[Get]]	SpecOp(propertyName) → any	Returns the value of the named property.
[[GetOwnProperty]]	SpecOp(propertyName) → Undefined or Property Descriptor	Returns the Property Descriptor of the named own property of this object, or <b>undefined</b> if absent.
[[GetProperty]]	SpecOp(propertyName) → Undefined or Property Descriptor	Returns the fully populated Property Descriptor of the named property of this object, or <b>undefined</b> if absent.
[[Put]]	SpecOp(propertyName, any, Boolean)	Sets the specified named property to the value of the second parameter. The flag controls failure handling.
[[CanPut]]	SpecOp(propertyName) → Boolean	Returns a Boolean value indicating whether a [[Put]] operation with <i>PropertyName</i> can be performed.
[[HasProperty]]	SpecOp(propertyName) → Boolean	Returns a Boolean value indicating whether the object already has a property with the given name.
[[Delete]]	SpecOp(propertyName, Boolean) → Boolean	Removes the specified named own property from the object. The flag controls failure handling.
[[DefaultValue]]	SpecOp(Hint) → primitive	Hint is a String. Returns a default value for the object.
[[DefineOwnProperty]]	SpecOp(propertyName,PropertyDescriptor, Boolean) → Boolean	Creates or alters the named own property to have the state described by a Property Descriptor. The flag controls failure handling.
[[Enumerate]]	SpecOp() → Object	Returns an object that can enumerate the string values of the keys of the enumerable properties of the object. The returned object is an Iterator object.
[[Iterate]]	SpecOp() → Object	Returns an object that can enumate the logical component values of the object. The returned object is an Iterator object.

Formatted Table

Deleted: [[Class]]

Every object (including host objects) must implement all of the internal properties listed in Table 8. However, the `[[DefaultValue]]` internal method may, for some objects, simply throw a **TypeError** exception.

All objects have an internal property called `[[Prototype]]`. The value of this property is either `null` or an object and is used for implementing inheritance. Whether or not a native object can have a host object as its `[[Prototype]]` depends on the implementation. Every `[[Prototype]]` chain must have finite length (that is, starting from any object, recursively accessing the `[[Prototype]]` internal property must eventually lead to a `null` value). Named data properties of the `[[Prototype]]` object are inherited (are visible as properties of the child object) for the purposes of get access, but not for put access. Named accessor properties are inherited for both get access and put access.

Every ECMAScript object has a Boolean-valued `[[Extensible]]` internal property that controls whether or not named properties may be added to the object. If the value of the `[[Extensible]]` internal property is `false` then additional named properties may not be added to the object. In addition, if `[[Extensible]]` is `false` the value of `[[Prototype]]` internal properties of the object may not be modified. Once the value of an `[[Extensible]]` internal property has been set to `false` it may not be subsequently changed to `true`.

**NOTE** This specification defines no ECMAScript language operators or built-in functions that permit a program to modify an object's `[[Prototype]]` internal properties or to change the value of `[[Extensible]]` from `false` to `true`. Implementation specific extensions that modify `[[Prototype]]` or `[[Extensible]]` must not violate the invariants defined in the preceding paragraph.

Unless otherwise specified, the common internal methods of native ECMAScript objects behave as described in 8.12. Array objects have a slightly different implementation of the `[[DefineOwnProperty]]` internal method (see 15.4.5.1) and String objects have a slightly different implementation of the `[[GetOwnProperty]]` internal method (see 15.5.5.2). Arguments objects (10.6) have different implementations of `[[Get]]`, `[[GetOwnProperty]]`, `[[DefineOwnProperty]]`, and `[[Delete]]`. Function objects (15.3) have a different implementation of `[[Get]]`.

Host objects may implement these internal methods in any manner unless specified otherwise; for example, one possibility is that `[[Get]]` and `[[Put]]` for a particular host object indeed fetch and store property values but `[[HasProperty]]` always generates `false`. However, if any specified manipulation of a host object's internal properties is not supported by an implementation, that manipulation must throw a **TypeError** exception when attempted.

The `[[GetOwnProperty]]` internal method of a host object must conform to the following invariants for each property of the host object:

- If a property is described as a data property and it may return different values over time, then either or both of the `[[Writable]]` and `[[Configurable]]` attributes must be `true` even if no mechanism to change the value is exposed via the other internal methods.
- If a property is described as a data property and its `[[Writable]]` and `[[Configurable]]` are both `false`, then the `SameValue` (according to 9.12) must be returned for the `[[Value]]` attribute of the property on all calls to `[[GetOwnProperty]]`.
- If the attributes other than `[[Writable]]` may change over time or if the property might disappear, then the `[[Configurable]]` attribute must be `true`.
- If the `[[Writable]]` attribute may change from `false` to `true`, then the `[[Configurable]]` attribute must be `true`.
- If the value of the host object's `[[Extensible]]` internal property has been observed by ECMAScript code to be `false`, then if a call to `[[GetOwnProperty]]` describes a property as non-existent all subsequent calls must also describe that property as non-existent.

The `[[DefineOwnProperty]]` internal method of a host object must not permit the addition of a new property to a host object if the `[[Extensible]]` internal property of that host object has been observed by ECMAScript code to be `false`.

If the `[[Extensible]]` internal property of that host object has been observed by ECMAScript code to be `false` then it must not subsequently become `true`.

**Deleted:** the `[[Class]]` and

**Deleted:** `[[Class]]` or

**Deleted:** `[[Class]]`.

**Deleted:** The value of the `[[Class]]` internal property is defined by this specification for every kind of built-in object. The value of the `[[Class]]` internal property of a host object may be any String value except one of `"Arguments"`, `"Array"`, `"Boolean"`, `"Date"`, `"Error"`, `"Function"`, `"JSON"`, `"Math"`, `"Number"`, `"Object"`, `"RegExp"`, and `"String"`. The value of a `[[Class]]` internal property is used internally to distinguish different kinds of objects. Note that this specification does not provide any means for a program to access that value except through `Object.prototype.toString` (see 15.2.4.2).¶

Table 9 — Internal Properties Only Defined for Some Objects

<b>Internal Property</b>	<b>Value Type Domain</b>	<b>Description</b>
[[NativeBrand]]	Members of the NativeBrand enumeration.	A tag value used by this specification to categorize various kinds of native ECMAScript objects defined in this specification. Host objects do not have this internal property.
[[PrimitiveValue]]	primitive	Internal state information associated with this object. Of the standard built-in ECMAScript objects, only Boolean, Date, Number, and String objects implement [[PrimitiveValue]].
[[Construct]]	SpecOp(a List of <i>any</i> ) → Object	Creates an object. Invoked via the <code>new</code> operator. The arguments to the SpecOp are the arguments passed to the <code>new</code> operator. Objects that implement this internal method are called <i>constructors</i> .
[[Call]]	SpecOp( <i>any</i> , a List of <i>any</i> ) → <i>any</i> or Reference	Executes code associated with the object. Invoked via a function call expression. The arguments to the SpecOp are this object and a list containing the arguments passed to the function call expression. Objects that implement this internal method are <i>callable</i> . Only callable objects that are host objects may return Reference values.
[[HasInstance]]	SpecOp( <i>any</i> ) → Boolean	Returns a Boolean value indicating whether the argument is likely an Object that was constructed by this object. Of the standard built-in ECMAScript objects, only Function objects implement [[HasInstance]].
[[Scope]]	Lexical Environment	A lexical environment that <i>is</i> the environment in which a Function object is executed. Of the standard built-in ECMAScript objects, only Function objects implement [[Scope]].
[[FormalParameters]]	<a href="#">Parse Tree</a>	<a href="#">A parse tree for ECMAScript code parsed with FormalParameterList as the goal symbol.</a> Of the standard built-in ECMAScript objects, only Function objects implement [[FormalParameters]].
[[Code]]	<a href="#">Parse Tree</a>	<a href="#">A parse tree for ECMAScript code parsed with FunctionBody as the goal symbol.</a> Of the standard built-in ECMAScript objects, only Function objects implement [[Code]].
[[Strict]]	Boolean	<a href="#">True if a Function object is a strict mode function. Of the standard built-in ECMAScript objects, only Function objects implement [[Strict]].</a>
[[TargetFunction]]	Object	The target function of a function object created using the standard built-in <code>Function.prototype.bind</code> method. Only ECMAScript objects created using <code>Function.prototype.bind</code> have a [[TargetFunction]] internal property.
[[BoundThis]]	<i>any</i>	The pre-bound <code>this</code> value of a function Object created using the standard built-in <code>Function.prototype.bind</code> method. Only ECMAScript objects created using <code>Function.prototype.bind</code> have a [[BoundThis]] internal property.
[[BoundArguments]]	List of <i>any</i>	The pre-bound argument values of a function Object created <i>by</i> the standard built-in <code>Function.prototype.bind</code> method. Only objects created <i>by</i> <code>Function.prototype.bind</code> have a [[BoundArguments]] internal property.
[[Match]]	SpecOp( <i>String</i> , <i>index</i> ) → <i>MatchResult</i>	Tests for a regular expression match and returns a <i>MatchResult</i> value (see 15.10.2.1). Of the standard built-in ECMAScript objects, only <code>RegExp</code> objects implement [[Match]].
[[ParameterMap]]	Object	Provides a mapping between the properties of an arguments object (see 10.6) and the formal parameters of the associated function. Only objects that are arguments objects have a [[ParameterMap]] internal property.

Formatted Table

Deleted: defines

Deleted: List of Strings

Deleted: A possibly empty List containing the identifier Strings of a Function's

Deleted: FormalParameterList

Deleted: ECMAScript code

Deleted: The ECMAScript code of a function

Deleted: using

Deleted: ECMAScript

Deleted: using

Deleted: ECMAScript

The `[[NativeBrand]]` internal property is used to identify native ECMAScript objects as objects that conform to specific parts of this specification. The value of a `[[NativeBrand]]` property is a single member of this set of enumerated values: `NativeFunction`, `NativeArray`, `StringWrapper`, `BooleanWrapper`, `NumberWrapper`, `NativeMath`, `NativeDate`, `NativeRegExp`, `NativeError`, `NativeJSON`, `NativeArguments`, `NativePrivateName`. The actual value of the `[[NativeBrand]]` internal property is only used to identify specific kinds of native ECMAScript objects. Host objects do not have this internal property.

**Table 10 — Values of the `[[NativeBrand]]` Internal Property**

Value	Category	Description
<code>NativeFunction</code>	Function objects	
<code>NativeArray</code>	Array objects	
<code>StringWrapper</code>	String objects	
<code>BooleanWrapper</code>	Boolean objects	
<code>NumberWrapper</code>	Number objects	
<code>NativeMath</code>	The Math object	
<code>NativeDate</code>	Date objects	
<code>NativeRegExp</code>	RegExp objects	
<code>NativeError</code>	Error objects	
<code>NativeJSON</code>	The JSON object	
<code>NativeArguments</code>	Arguments objects	
<code>NativePrivateName</code>	Private Name objects	

## 8.7 The List and Record Specification Type

The List type is used to explain the evaluation of argument lists (see 11.2.4) in `new` expressions, in function calls, and in other algorithms where a simple list of values is needed. Values of the List type are simply ordered sequences of values. These sequences may be of any length.

The Record type is used to describe data aggregations within the algorithms of this specification. A Record type value consists of one or more named fields. The value of each field is either the ECMAScript type value or a an abstract value represented by a name associated with the Record type. Field names are always enclosed in double brackets. For example, `[[value]]`

For notational convenience within this specification, an object literal-like syntax can be used to express a Record value. For example, `{[[field1]]: 42, [[field2]]: false, [[field3]]: empty}` defines a Record value that has fields three fields each of which is initialized to a specific value. Field name order is not significant. Any fields that are not explicitly listed are considered to be absent.

In specification text and algorithms, dot notation may be used to refer to a specific field of a Record value. For example, if D is the record shown in the previous paragraph then `D.[[field2]]` is shorthand for “the field of D named `[[Field2]]`”.

Schema for commonly used Record field combination may be named and that name may be used as a prefix to a literal Record value to identify the specific kind of aggregations that is being described. For example: Property Descriptor `{[[Value]]: 42, [[Writable]]: false, [[Configurable]]: true}`

## 8.8 The Completion Record Specification Type

The Completion type is a Record used to explain the runtime propagation of values and control flow such as the behaviour of statements (`break`, `continue`, `return` and `throw`) that perform nonlocal transfers of control.

Values of the Completion type are Record values whose fields are defined as by Table xxx.

**Deleted:** Table 10 summarises internal properties used by this specification to categorize various kinds of native ECMAScript objects defined in this specification. If an object has one of these properties it is a member of the corresponding object category and has the characteristics specified for that category of objects.

**Deleted:** these

**Deleted:** ies

**Deleted:** are not

**Deleted:** used

**Deleted:** may

**Deleted:** any of these

**Deleted:** ies

**Deleted:** Internal Properties

**Deleted:** that Categorize Native Object

**Deleted:** s

**Deleted:** Internal Property

**Deleted:** Formatted Table

**Deleted:** `[[IsFunction]]`

**Deleted:** `]]`

**Deleted:** `[[IsArray]]`

**Deleted:** `[[IsString]]`

**Deleted:** `[[IsBoolean]]`

**Deleted:** `[[IsNumber]]`

**Deleted:** `[[IsMath]]`

**Deleted:** c

**Deleted:** `[[IsDate]]`

**Deleted:** ]

**Deleted:** ]

**Deleted:** `[[IsRegExp]]`

**Deleted:** `[[IsError]]`

**Deleted:** `[[IsJSON]]`

**Deleted:** `[[IsArguments]]`

**Deleted:** []

**Deleted:** Formatted Table

**Deleted:** le

**Table xxx — Completion Record Fields**

<u>Field Name</u>	<u>Value</u>	<u>Meaning</u>
<u>[[type]]</u>	<u>One of <code>normal</code>, <code>break</code>, <code>continue</code>, <code>return</code>, or <code>throw</code></u>	<u>The type of completion that occurred.</u>
<u>[[value]]</u>	<u>any ECMAScript language value or <code>empty</code></u>	<u>The value that was produced.</u>
<u>[[target]]</u>	<u>any ECMAScript identifier or <code>empty</code></u>	<u>The target label for directed control transfers.</u>

**Formatted Table**

The term “abrupt completion” refers to any completion with a [[type]] value other than `normal`.

The algorithms of this specification often implicitly return Completion Records whose [[type]] is `normal`. Unless it is otherwise obvious from the context, an algorithm statement that returns a value that is not a Completion Record such as:

1. Return the String `"Infinity"`.

mean the same things as:

1. Return Completion {[[type]]: `normal`, [[value]]: String `"Infinity"`, [[target]]: `empty`}.

Similarly, any reference to a Completion Record value that is in a context that does not explicitly require a complete Completion Record value is equivalent to an explicit reference to the [[value]] field of the Completion Record value unless the Completion Record is an abrupt completion.

The abstraction operation `NormalCompletion` with a single argument is a short hand that is defined as follows:

1. Return Completion {[[type]]: `normal`, [[value]]: *argument*, [[target]]: `empty`}.

Algorithms steps that say to throw an exception, such as

1. Throw a `TypeError` exception.

mean the same things as:

1. Return Completion {[[type]]: `throw`, [[value]]: a newly created `TypeError` object, [[target]]: `empty`}.

Algorithms steps that say

1. `ReturnIfAbrupt(argument)`.

mean the same things as:

1. If *argument* is an abrupt completion, then return *argument*.  
 2. Else if *argument* is a Completion Record, then let *argument* be *argument*.[[value]].

## 8.9 The Reference Specification Type

**NOTE** The Reference type is used to explain the behaviour of such operators as `delete`, `typeof`, and the assignment operators. For example, the left-hand operand of an assignment is expected to produce a reference. The behaviour of assignment could, instead, be explained entirely in terms of a case analysis on the syntactic form of the left-hand operand of an assignment operator, but for one difficulty: function calls are permitted to return references. This possibility is admitted purely for the sake of host objects. No built-in ECMAScript function defined by this specification returns a reference and there is no provision for a user-defined function to return a reference. (Another reason not to use a syntactic case analysis is that it would be lengthy and awkward, affecting many parts of the specification.)

A **Reference** is a resolved name binding. A Reference consists of three components, the *base* value, the referenced name and the Boolean valued *strict reference* flag. The *base* value is either `undefined`, an Object, a

**Deleted:** The abstraction operation `NormalValue` with a single argument is a short hand that is defined as follows:  
 ↘ If *argument* is not a Completion Record, return *argument*.  
 ↘ If *argument* is an abrupt completion, return *argument*.  
 ↘ Return *argument*.[[value]].

**Formatted:** Font: Times New Roman

**Formatted:** Outline numbered + Level: 1 + Numbering Style: 1, 2, 3, ... + Start at: 1 + Alignment: Left + Aligned at: 0" + Tab after: 0.25" + Indent at: 0.25"

**Formatted:** Font: Italic

**Formatted:** Alg 3, Space After: 11 pt, Outline numbered + Level: 1 + Numbering Style: 1, 2, 3, ... + Start at: 1 + Alignment: Left + Aligned at: 0" + Tab after: 0.25" + Indent at: 0.25"

**Deleted:**

**Formatted:** Add space between paragraphs the same style

**Deleted:** <#>.

**Deleted:** 7

**Formatted:** Font: 9 pt

**Formatted:** Font: Times New Roman, Italic

Boolean, a String, a Number, or an environment record (10.2.1). A *base* value of **undefined** indicates that the Reference could not be resolved to a binding. The referenced name is a String.

A Super Reference is a Reference that is used to represents a name binding that was expressed using the `super` keyword. A Super Reference has an additional `thisValue` component and its `base` value will never be an environment record.

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

- GetBase(*V*). Returns the base value component of the reference *V*.
- GetReferencedName(*V*). Returns the referenced name component of the reference *V*.
- IsStrictReference(*V*). Returns the strict reference component of the reference *V*.
- HasPrimitiveBase(*V*). Returns **true** if the base value is a Boolean, String, or Number.
- IsPropertyReference(*V*). Returns **true** if either the base value is an object or HasPrimitiveBase(*V*) is **true**; otherwise returns **false**.
- IsUnresolvableReference(*V*). Returns **true** if the base value is **undefined** and **false** otherwise.
- IsSuperReference(*V*). Returns **true** if this reference has a `thisValue` component.
- GetThisValue(*V*). Returns the value of the `thisValue` component of the reference *V*. May only be used when IsSuperReference(*V*) is **true**.

The following abstract operations are used in this specification to operate on references:

### 8.9.1 GetValue (*V*)

1. ReturnIfAbrupt(*V*).
2. If Type(*V*) is not Reference, return *V*.
3. Let *base* be the result of calling GetBase(*V*).
4. If IsUnresolvableReference(*V*), throw a **ReferenceError** exception.
5. If IsPropertyReference(*V*), then
  - a. If HasPrimitiveBase(*V*) is **false**, then let *get* be the [[Get]] internal method of *base*, otherwise let *get* be the special [[Get]] internal method defined below.
  - b. If IsSuperReference(*V*) is **false**, then
    - i. Return the result of calling *get* as an internal method of *base* passing GetReferencedName(*V*) for the argument.
  - c. Else,
    - i. Return the result of calling *get* as an internal method of *base* passing GetReferencedName(*V*) and ThisValue(*V*) as the arguments.
6. Else, *base* must be an environment record.
  - a. Return the result of calling the GetBindingValue (see 10.2.1) concrete method of *base* passing GetReferencedName(*V*) and IsStrictReference(*V*) as arguments.

The following [[Get]](*P* [*I*, *accessorThisValue*]) internal method is used by GetValue when *V* is a property reference with a primitive base value. *base* is the value the internal method is called upon, with property name *P* as its argument. The following steps are taken:

1. Let *O* be ToObject(*base*).
2. Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.
3. If *desc* is **undefined**, return **undefined**.
4. If IsDataDescriptor(*desc*) is **true**, return *desc*.[[Value]].
5. Otherwise, IsAccessorDescriptor(*desc*) must be **true** so, let *getter* be *desc*.[[Get]] (see 8.10).
6. If *getter* is **undefined**, return **undefined**.
7. If the *accessorThisValue* argument is not present, let *accessorThisValue* be *base*
8. Return the result of calling the [[Call]] internal method of *getter* providing *accessorThisValue*, as the **this** value and providing no arguments.

NOTE The object that may be created in step 1 is not accessible outside of the above method. An implementation might choose to avoid the actual creation of the object. The only situation where such an actual property access that uses this internal method can have visible effect is when it invokes an accessor function.

**Formatted:** Font: Times New Roman, Italic

**Deleted:** reference

**Formatted:** Font: Times New Roman, Italic

**Formatted:** Font: (Default) Courier New, Bold

**Formatted:** Font: Times New Roman, Italic

**Formatted:** Font: Times New Roman, Italic

**Formatted:** Don't add space between paragraphs of the same style

**Formatted:** Don't add space between paragraphs of the same style

**Deleted:** 7

**Deleted:** <#>Let *V* be NormalValue(*I*).¶

**Deleted:** the

**Deleted:** using

**Deleted:** as its **this** value, and

**Deleted:** It is called using

**Deleted:** as

**Deleted:** its **this** value and

**Deleted:** *base*

### 8.9.2 PutValue (V, W)

1. ReturnIfAbrupt(V).
2. ReturnIfAbrupt(W).
3. If Type(V) is not Reference, throw a **ReferenceError** exception.
4. Let *base* be the result of calling GetBase(V).
5. If IsUnresolvableReference(V), then
  - a. If IsStrictReference(V) is true, then
    - i. Throw **ReferenceError** exception.
  - b. Return the result of calling the [[Put]] internal method of the global object, passing GetReferencedName(V) for the property name, W for the value, and false for the Throw flag.
6. Else if IsPropertyReference(V), then
  - a. If HasPrimitiveBase(V) is false, then let *put* be the [[Put]] internal method of *base*, otherwise let *put* be the special [[Put]] internal method defined below.
  - b. If IsSuperReference(V) is false, then
    - i. Return the result of calling put as an internal method of base passing GetReferencedName(V) for the property name, W for the value, and IsStrictReference(V) for the Throw flag.
    - c. Else,
      - i. Return the result of calling put as an internal method of base passing GetReferencedName(V) for the property name, W for the value, IsStrictReference(V) for the Throw flag, and ThisValue(V) as the accessorThisValue argument.
7. Else *base* must be a reference whose base is an environment record. So,
  - a. Return the result of calling the SetMutableBinding (10.2.1) concrete method of base, passing GetReferencedName(V), W, and IsStrictReference(V) as arguments.
8. Return undefined.

The following [[Put]] internal method is used by PutValue when V is a property reference with a primitive base value. *base* is the value the internal method is called upon with property *name* P, value *W*, and Boolean flag *Throw* as required arguments and optional argument *accessorThisValue*. The following steps are taken:

1. Let *O* be ToObject(*base*).
2. If the result of calling the [[CanPut]] internal method of *O* with argument *P* is false, then
  - a. If *Throw* is true, then throw a **TypeError** exception.
  - b. Else return undefined.
3. Let *ownDesc* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument *P*.
4. If IsDataDescriptor(*ownDesc*) is true, then
  - a. If *Throw* is true, then throw a **TypeError** exception.
  - b. Else return undefined.
5. Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with argument *P*. This may be either an own or inherited accessor property descriptor or an inherited data property descriptor.
6. If IsAccessorDescriptor(*desc*) is true, then
  - a. Let *setter* be *desc*.[[Set]] (see 8.10) which cannot be undefined.
  - b. If accessorThisValue is not present, then let accessorThisValue be base.
  - c. Return the result of calling the [[Call]] internal method of *setter* providing *accessorThisValue* as the this value and an argument list containing only *W*.
7. Else, this is a request to create an own property on the transient object *O*
  - a. If *Throw* is true, then throw a **TypeError** exception.
8. Return undefined.

**NOTE** The object that may be created in step 1 is not accessible outside of the above method. An implementation might choose to avoid the actual creation of that transient object. The only situations where such an actual property assignment that uses this internal method can have visible effect are when it either invokes an accessor function or is in violation of a *Throw* predicated error check. When *Throw* is true any property assignment that would create a new property on the transient object throws an error.

### 8.9.3 GetThisValue (V)

1. ReturnIfAbrupt(V).
2. If Type(V) is not Reference, return *V*.
3. If IsUnresolvableReference(V), throw a **ReferenceError** exception.

**Deleted:** 7

**Deleted:** <#>Let *V* be NormalValue(*V*).  
<#>Let *W* be NormalValue(*W*).¶

**Deleted:** Call

**Formatted**  
**Deleted:** Call  
**Deleted:** the  
**Deleted:** as its this value, and  
**Deleted:** using

**Deleted:** Call

**Deleted:** It is called using  
**Deleted:** as its this value and

**Deleted:** Call  
**Deleted:** base

4. If `IsSuperReference(V)`, then
  - a. Return the value of the `thisValue` component of the reference `V`.
5. Return `GetBase(V)`.

## 8.10 The Property Descriptor and Property Identifier Specification Types

The Property Descriptor type is used to explain the manipulation and reification of named property attributes. Values of the Property Descriptor type are records composed of named fields where each field's name is an attribute name and its value is a corresponding attribute value as specified in 8.6.1. In addition, any field may be present or absent.

Property Descriptor values may be further classified as data property descriptors and accessor property descriptors based upon the existence or use of certain fields. A data property descriptor is one that includes any fields named either `[[Value]]` or `[[Writable]]`. An accessor property descriptor is one that includes any fields named either `[[Get]]` or `[[Set]]`. Any property descriptor may have fields named `[[Enumerable]]` and `[[Configurable]]`. A Property Descriptor value may not be both a data property descriptor and an accessor property descriptor; however, it may be neither. A generic property descriptor is a Property Descriptor value that is neither a data property descriptor nor an accessor property descriptor. A fully populated property descriptor is one that is either an accessor property descriptor or a data property descriptor and that has all of the fields that correspond to the property attributes defined in either 8.6.1 Table 5 or Table 6.

For notational convenience within this specification, an object literal-like syntax can be used to define a property descriptor value. For example, Property Descriptor `{[[Value]]: 42, [[Writable]]: false, [[Configurable]]: true}` defines a data property descriptor. Field name order is not significant. Any fields that are not explicitly listed are considered to be absent.

In specification text and algorithms, dot notation may be used to refer to a specific field of a Property Descriptor. For example, if `D` is a property descriptor then `D.[[Value]]` is shorthand for “the field of `D` named `[[Value]]`”.

The Property Identifier type is used to associate a property name with a Property Descriptor. Values of the Property Identifier type are pairs of the form `(name, descriptor)`, where `name` is a String and `descriptor` is a Property Descriptor value.

The following abstract operations are used in this specification to operate upon Property Descriptor values:

### 8.10.1 IsAccessorDescriptor ( Desc )

When the abstract operation `IsAccessorDescriptor` is called with property descriptor `Desc`, the following steps are taken:

2. If `Desc` is **undefined**, then return **false**.
3. If both `Desc.[[Get]]` and `Desc.[[Set]]` are absent, then return **false**.
4. Return **true**.

### 8.10.2 IsDataDescriptor ( Desc )

When the abstract operation `IsDataDescriptor` is called with property descriptor `Desc`, the following steps are taken:

1. If `Desc` is **undefined**, then return **false**.
2. If both `Desc.[[Value]]` and `Desc.[[Writable]]` are absent, then return **false**.
3. Return **true**.

### 8.10.3 IsGenericDescriptor ( Desc )

When the abstract operation `IsGenericDescriptor` is called with property descriptor `Desc`, the following steps are taken:

**Comment [AWB4]:** The ES5 use of Property Identifier in specifying object literals has been removed. If no usage is added they should be removed from the spec.

**Deleted: 8.8 The List Specification Type¶**

The List type is used to explain the evaluation of argument lists (see 11.2.4) in `new` expressions, in function calls, and in other algorithms where a simple list of values is needed. Values of the List type are simply ordered sequences of values. These sequences may be of any length.¶

**8.9 . The Completion Specification Type¶**

The Completion type is used to explain the behaviour of statements (`break`, `continue`, `return` and `throw`) to perform nonlocal transfers of control. Values of the Completion type are triples of the form `(type, value, target)`, where `type` is one of `normal`, `break`, `continue`, `return`, or `throw`, `value` is any ECMAScript language value or `empty`, and `target` is any ECMAScript identifier or `empty`. If `cv` is a completion value then `cv.type`, `cv.value` and `cv.target` may be used to directly refer to its constituent values.¶

The term “abrupt completion” refers to any completion with a type other than `normal`.¶

**Formatted:** Outline numbered + Level: 1 + Numbering Style: 1, 2, 3, ... + Start at: 1 + Alignment: Left + Aligned at: 0" + Tab after: 0.25" + Indent at: 0.25"

1. If *Desc* is **undefined**, then return **false**.
2. If *IsAccessorDescriptor(Desc)* and *IsDataDescriptor(Desc)* are both **false**, then return **true**.
3. Return **false**.

#### 8.10.4 FromPropertyDescriptor ( *Desc* )

When the abstract operation *FromPropertyDescriptor* is called with property descriptor *Desc*, the following steps are taken:

The following algorithm assumes that *Desc* is a fully populated Property Descriptor, such as that returned from *[[GetOwnProperty]]* (see 8.12.1).

1. If *Desc* is **undefined**, then return **undefined**.
2. Let *obj* be the result of [the abstraction operation ObjectCreate \(15.2\)](#)
3. If *IsDescriptor(Desc)* is **true**, then
  - a. Call the *[[DefineOwnProperty]]* internal method of *obj* with arguments "**value**", Property Descriptor *{[[Value]]: Desc.[[Value]], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}*, and **false**.
  - b. Call the *[[DefineOwnProperty]]* internal method of *obj* with arguments "**writable**", Property Descriptor *{[[Value]]: Desc.[[Writable]], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}*, and **false**.
4. Else, *IsAccessorDescriptor(Desc)* must be **true**, so
  - a. Call the *[[DefineOwnProperty]]* internal method of *obj* with arguments "**get**", Property Descriptor *{[[Value]]: Desc.[[Get]], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}*, and **false**.
  - b. Call the *[[DefineOwnProperty]]* internal method of *obj* with arguments "**set**", Property Descriptor *{[[Value]]: Desc.[[Set]], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}*, and **false**.
5. Call the *[[DefineOwnProperty]]* internal method of *obj* with arguments "**enumerable**", Property Descriptor *{[[Value]]: Desc.[[Enumerable]], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}*, and **false**.
6. Call the *[[DefineOwnProperty]]* internal method of *obj* with arguments "**configurable**", Property Descriptor *{[[Value]]: Desc.[[Configurable]], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}*, and **false**.
7. Return *obj*.

**Deleted:** creating a new object as if by the expression **new Object()** where **Object** is the standard built-in constructor with that name

#### 8.10.5 ToPropertyDescriptor ( *Obj* )

When the abstract operation *ToPropertyDescriptor* is called with object *Obj*, the following steps are taken:

1. [ReturnIfAbrupt\(\*Obj\*\)](#)
2. If *Type(Obj)* is not **Object** throw a **TypeError** exception.
3. Let *desc* be the result of creating a new Property Descriptor that initially has no fields.
4. If the result of calling the *[[HasProperty]]* internal method of *Obj* with argument "**enumerable**" is **true**, then
  - a. Let *enum* be the result of calling the *[[Get]]* internal method of *Obj* with "**enumerable**".
  - b. [ReturnIfAbrupt\(\*enum\*\)](#).
  - c. Set the *[[Enumerable]]* field of *desc* to *ToBoolean(enum)*.
5. If the result of calling the *[[HasProperty]]* internal method of *Obj* with argument "**configurable**" is **true**, then
  - a. Let *conf* be the result of calling the *[[Get]]* internal method of *Obj* with argument "**configurable**".
  - b. [ReturnIfAbrupt\(\*conf\*\)](#).
  - c. Set the *[[Configurable]]* field of *desc* to *ToBoolean(conf)*.
6. If the result of calling the *[[HasProperty]]* internal method of *Obj* with argument "**value**" is **true**, then
  - a. Let *value* be the result of calling the *[[Get]]* internal method of *Obj* with argument "**value**".
  - b. [ReturnIfAbrupt\(\*value\*\)](#).
  - c. Set the *[[Value]]* field of *desc* to *value*.
7. If the result of calling the *[[HasProperty]]* internal method of *Obj* with argument "**writable**" is **true**, then
  - a. Let *writable* be the result of calling the *[[Get]]* internal method of *Obj* with argument "**writable**".
  - b. [ReturnIfAbrupt\(\*writable\*\)](#).
  - c. Set the *[[Writable]]* field of *desc* to *ToBoolean(writable)*.
8. If the result of calling the *[[HasProperty]]* internal method of *Obj* with argument "**get**" is **true**, then
  - a. Let *getter* be the result of calling the *[[Get]]* internal method of *Obj* with argument "**get**".
  - b. [ReturnIfAbrupt\(\*getter\*\)](#).
  - c. If *IsCallable(getter)* is **false** and *getter* is not **undefined**, then throw a **TypeError** exception.

**Deleted:** g

- d. Set the [[Get]] field of *desc* to *getter*.
- 9. If the result of calling the [[HasProperty]] internal method of *Obj* with argument "set" is **true**, then
  - a. Let *setter* be the result of calling the [[Get]] internal method of *Obj* with argument "set".
  - b. ReturnIfAbrupt(setter).**
  - c. If IsCallable(*setter*) is **false** and *setter* is not **undefined**, then throw a **TypeError** exception.
  - d. Set the [[Set]] field of *desc* to *setter*.
- 10. If either *desc*.[[Get]] or *desc*.[[Set]] are present, then
  - a. If either *desc*.[[Value]] or *desc*.[[Writable]] are present, then throw a **TypeError** exception.
- 11. Return *desc*.

## 8.11 The Lexical Environment and Environment Record Specification Types

The Lexical Environment and Environment Record types are used to explain the behaviour of name resolution in nested functions and blocks. These types and the operations upon them are defined in Clause 10.

## 8.12 Algorithms for Object Internal Methods

In the following algorithm descriptions, assume *O* is a native ECMAScript object, *P* is a String, *Desc* is a Property Description record, and *Throw* is a Boolean flag.

### 8.12.1 [[GetOwnProperty]] (*P*)

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

1. **If *O* doesn't have an own property with name *P*, return **undefined**.**
2. Let *D* be a newly created Property Descriptor with no fields.
3. Let *X* be *O*'s own property named *P*.
4. If *X* is a data property, then
  - a. Set *D*.[[Value]] to the value of *X*'s [[Value]] attribute.
  - b. Set *D*.[[Writable]] to the value of *X*'s [[Writable]] attribute
5. Else *X* is an accessor property, so
  - a. Set *D*.[[Get]] to the value of *X*'s [[Get]] attribute.
  - b. Set *D*.[[Set]] to the value of *X*'s [[Set]] attribute.
6. Set *D*.[[Enumerable]] to the value of *X*'s [[Enumerable]] attribute.
7. Set *D*.[[Configurable]] to the value of *X*'s [[Configurable]] attribute.
8. Return *D*.

**Deleted:** <#>ReturnIfAbrupt(*O*).¶

However, if *O* is a String object it has a more elaborate [[GetOwnProperty]] internal method defined in 15.5.5.2.

### 8.12.2 [[GetProperty]] (*P*)

When the [[GetProperty]] internal method of *O* is called with property name *P*, the following steps are taken:

1. **Let *prop* be the result of calling the [[GetOwnProperty]] internal method of *O* with property name *P*.**
2. If *prop* is not **undefined**, return *prop*.
3. Let *proto* be the value of the [[Prototype]] internal property of *O*.
4. If *proto* is **null**, return **undefined**.
5. Return the result of calling the [[GetProperty]] internal method of *proto* with argument *P*.

**Deleted:** <#>ReturnIfAbrupt(*O*).¶

### 8.12.3 [[Get]] (*P* [*accessorThisValue*])

When the [[Get]] internal method of *O* is called with property name *P*, the following steps are taken:

1. **Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.**
2. If *desc* is **undefined**, return **undefined**.
3. If IsDataDescriptor(*desc*) is **true**, return *desc*.[[Value]].
4. Otherwise, IsAccessorDescriptor(*desc*) must be true so, let *getter* be *desc*.[[Get]].
5. If *getter* is **undefined**, return **undefined**.
- 6. If *accessorThisValue* is not present, then let *accessorThisValue* be *O*.**

**Deleted:** <#>ReturnIfAbrupt(*O*).¶

7. Return the result calling the `[[Call]]` internal method of `getter` providing `accessorThisValue` as the `this` value and providing no arguments.

**Deleted:** `O`

#### 8.12.4 `[[CanPut]](P)`

When the `[[CanPut]]` internal method of `O` is called with property name `P`, the following steps are taken:

1. Let `desc` be the result of calling the `[[GetOwnProperty]]` internal method of `O` with argument `P`.
2. If `desc` is not `undefined`, then
  - a. If `IsAccessorDescriptor(desc)` is `true`, then
    - i. If `desc.[[Set]]` is `undefined`, then return `false`.
    - ii. Else return `true`.
  - b. Else, `desc` must be a DataDescriptor so return the value of `desc.[[Writable]]`.
3. Let `proto` be the `[[Prototype]]` internal property of `O`.
4. If `proto` is `null`, then return the value of the `[[Extensible]]` internal property of `O`.
5. Let `inherited` be the result of calling the `[[GetProperty]]` internal method of `proto` with property name `P`.
6. If `inherited` is `undefined`, return the value of the `[[Extensible]]` internal property of `O`.
7. If `IsAccessorDescriptor(inherited)` is `true`, then
  - a. If `inherited.[[Set]]` is `undefined`, then return `false`.
  - b. Else return `true`.
8. Else, `inherited` must be a DataDescriptor
  - a. If the `[[Extensible]]` internal property of `O` is `false`, return `false`.
  - b. Else return the value of `inherited.[[Writable]]`.

**Deleted:** `<#>ReturnIfAbrupt(O).¶`

Host objects may define additional constraints upon `[[Put]]` operations. If possible, host objects should not allow `[[Put]]` operations in situations where this definition of `[[CanPut]]` returns false.

#### 8.12.5 `[[Put]](P, V, Throw, accessorThisValue)`

When the `[[Put]]` internal method of `O` is called with property `P`, value `V`, and Boolean flag `Throw`, the following steps are taken:

1. If the result of calling the `[[CanPut]]` internal method of `O` with argument `P` is `false`, then
  - a. If `Throw` is `true`, then throw a `TypeError` exception.
  - b. Else return `undefined`.
2. Let `ownDesc` be the result of calling the `[[GetOwnProperty]]` internal method of `O` with argument `P`.
3. If `IsDataDescriptor(ownDesc)` is `true`, then
  - a. Let `valueDesc` be the Property Descriptor `{[[Value]]: V}`.
  - b. Return the result of calling the `[[DefineOwnProperty]]` internal method of `O` passing `P, valueDesc`, and `Throw` as arguments.
4. Let `desc` be the result of calling the `[[GetProperty]]` internal method of `O` with argument `P`. This may be either an own or inherited accessor property descriptor or an inherited data property descriptor.
5. If `IsAccessorDescriptor(desc)` is `true`, then
  - a. Let `setter` be `desc.[[Set]]` which cannot be `undefined`.
  - b. If `accessorThisValue` is not present, then let `accessorThisValue` be `O`.
  - c. Return the result of calling the `[[Call]]` internal method of `setter` providing `thisValue` as the `this` value and providing `V` as the sole argument.
6. Else, create a named data property named `P` on object `O` as follows
  - a. Let `newDesc` be the Property Descriptor `{[[Value]]: V, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`.
  - b. Return the result of calling the `[[DefineOwnProperty]]` internal method of `O` passing `P, newDesc`, and `Throw` as arguments.

**Deleted:** `<#>ReturnIfAbrupt(O).¶`

**Deleted:** Call

**Deleted:** <#>Return.¶

**Deleted:** Call

**Deleted:** `O`

**Deleted:** Call

**Deleted:** <#>Return.¶

#### 8.12.6 `[[HasProperty]](P)`

When the `[[HasProperty]]` internal method of `O` is called with property name `P`, the following steps are taken:

1. Let `desc` be the result of calling the `[[GetProperty]]` internal method of `O` with property name `P`.
2. If `desc` is `undefined`, then return `false`.
3. Else return `true`.

**Deleted:** `<#>ReturnIfAbrupt(O).¶`

### 8.12.7 [[Delete]] (P, Throw)

When the [[Delete]] internal method of  $O$  is called with property name  $P$  and the Boolean flag  $Throw$ , the following steps are taken:

1. Let  $desc$  be the result of calling the [[GetOwnProperty]] internal method of  $O$  with property name  $P$ .
2. If  $desc$  is **undefined**, then return **true**.
3. If  $desc.[[Configurable]]$  is **true**, then
  - a. Remove the own property with name  $P$  from  $O$ .
  - b. Return **true**.
4. Else if  $Throw$ , then throw a **TypeError** exception.
5. Return **false**.

**Deleted:** <#>ReturnIfAbrupt( $O$ ).¶

### 8.12.8 [[DefaultValue]] (hint)

When the [[DefaultValue]] internal method of  $O$  is called with hint String, the following steps are taken:

1. Let  $toString$  be the result of calling the [[Get]] internal method of object  $O$  with argument "**toString**".
2. [ReturnIfAbrupt\( \$toString\$ \)](#).
3. If IsCallable( $toString$ ) is **true** then,
  - a. Let  $str$  be the result of calling the [[Call]] internal method of  $toString$ , with  $O$  as the **this** value and an empty argument list.
  - b. [ReturnIfAbrupt\( \$str\$ \)](#).
  - c. If  $str$  is a primitive value, return  $str$ .
4. Let  $valueOf$  be the result of calling the [[Get]] internal method of object  $O$  with argument "**valueOf**".
5. [ReturnIfAbrupt\( \$valueOf\$ \)](#).
6. If IsCallable( $valueOf$ ) is **true** then,
  - a. Let  $val$  be the result of calling the [[Call]] internal method of  $valueOf$ , with  $O$  as the **this** value and an empty argument list.
  - b. [ReturnIfAbrupt\( \$val\$ \)](#).
  - c. If  $val$  is a primitive value, return  $val$ .
7. Throw a **TypeError** exception.

**Deleted:** <#>ReturnIfAbrupt( $O$ ).¶

When the [[DefaultValue]] internal method of  $O$  is called with hint Number, the following steps are taken:

1. Let  $valueOf$  be the result of calling the [[Get]] internal method of object  $O$  with argument "**valueOf**".
2. [ReturnIfAbrupt\( \$valueOf\$ \)](#).
3. If IsCallable( $valueOf$ ) is **true** then,
  - a. Let  $val$  be the result of calling the [[Call]] internal method of  $valueOf$ , with  $O$  as the **this** value and an empty argument list.
  - b. [ReturnIfAbrupt\( \$val\$ \)](#).
  - c. If  $val$  is a primitive value, return  $val$ .
4. Let  $toString$  be the result of calling the [[Get]] internal method of object  $O$  with argument "**toString**".
5. [ReturnIfAbrupt\( \$toString\$ \)](#).
6. If IsCallable( $toString$ ) is **true** then,
  - a. Let  $str$  be the result of calling the [[Call]] internal method of  $toString$ , with  $O$  as the **this** value and an empty argument list.
  - b. [ReturnIfAbrupt\( \$str\$ \)](#).
  - c. If  $str$  is a primitive value, return  $str$ .
7. Throw a **TypeError** exception.

**Deleted:** <#>ReturnIfAbrupt( $O$ ).¶

When the [[DefaultValue]] internal method of  $O$  is called with no hint, then it behaves as if the hint were Number, unless  $O$  is a Date object (see 15.9.6), in which case it behaves as if the hint were String.

The above specification of [[DefaultValue]] for native objects can return only primitive values. If a host object implements its own [[DefaultValue]] internal method, it must ensure that its [[DefaultValue]] internal method can return only primitive values.

### 8.12.9 [[DefineOwnProperty]] (P, Desc, Throw)

In the following algorithm, the term “Reject” means “If *Throw* is **true**, then throw a **TypeError** exception, otherwise return **false**”. The algorithm contains steps that test various fields of the Property Descriptor *Desc* for specific values. The fields that are tested in this manner need not actually exist in *Desc*. If a field is absent then its value is considered to be **false**.

When the [[DefineOwnProperty]] internal method of *O* is called with property name *P*, property descriptor *Desc*, and Boolean flag *Throw*, the following steps are taken:

1. Let *current* be the result of calling the [[GetOwnProperty]] internal method of *O* with property name *P*.
2. Let *extensible* be the value of the [[Extensible]] internal property of *O*.
3. If *current* is **undefined** and *extensible* is **false**, then Reject.
4. If *current* is **undefined** and *extensible* is **true**, then
  - a. If IsGenericDescriptor(*Desc*) or IsDataDescriptor(*Desc*) is **true**, then
    - i. Create an own data property named *P* of object *O* whose [[Value]], [[Writable]], [[Enumerable]] and [[Configurable]] attribute values are described by *Desc*. If the value of an attribute field of *Desc* is absent, the attribute of the newly created property is set to its default value.
  - b. Else, *Desc* must be an accessor Property Descriptor so,
    - i. Create an own accessor property named *P* of object *O* whose [[Get]], [[Set]], [[Enumerable]] and [[Configurable]] attribute values are described by *Desc*. If the value of an attribute field of *Desc* is absent, the attribute of the newly created property is set to its default value.
  - c. Return **true**.
5. Return **true**, if every field in *Desc* is absent.
6. Return **true**, if every field in *Desc* also occurs in *current* and the value of every field in *Desc* is the same value as the corresponding field in *current* when compared using the SameValue algorithm (9.12).
7. If the [[Configurable]] field of *current* is **false** then
  - a. Reject, if the [[Configurable]] field of *Desc* is **true**.
  - b. Reject, if the [[Enumerable]] field of *Desc* is present and the [[Enumerable]] fields of *current* and *Desc* are the Boolean negation of each other.
8. If IsGenericDescriptor(*Desc*) is **true**, then no further validation is required.
9. Else, if IsDataDescriptor(*current*) and IsDataDescriptor(*Desc*) have different results, then
  - a. Reject, if the [[Configurable]] field of *current* is **false**.
  - b. If IsDataDescriptor(*current*) is **true**, then
    - i. Convert the property named *P* of object *O* from a data property to an accessor property. Preserve the existing values of the converted property’s [[Configurable]] and [[Enumerable]] attributes and set the rest of the property’s attributes to their default values.
  - c. Else,
    - i. Convert the property named *P* of object *O* from an accessor property to a data property. Preserve the existing values of the converted property’s [[Configurable]] and [[Enumerable]] attributes and set the rest of the property’s attributes to their default values.
10. Else, if IsDataDescriptor(*current*) and IsDataDescriptor(*Desc*) are both **true**, then
  - a. If the [[Configurable]] field of *current* is **false**, then
    - i. Reject, if the [[Writable]] field of *current* is **false** and the [[Writable]] field of *Desc* is **true**.
    - ii. If the [[Writable]] field of *current* is **false**, then
      1. Reject, if the [[Value]] field of *Desc* is present and SameValue(*Desc*.[[Value]], *current*.[[Value]]) is **false**.
  - b. else, the [[Configurable]] field of *current* is **true**, so any change is acceptable.
11. Else, IsAccessorDescriptor(*current*) and IsAccessorDescriptor(*Desc*) are both **true** so,
  - a. If the [[Configurable]] field of *current* is **false**, then
    - i. Reject, if the [[Set]] field of *Desc* is present and SameValue(*Desc*.[[Set]], *current*.[[Set]]) is **false**.
    - ii. Reject, if the [[Get]] field of *Desc* is present and SameValue(*Desc*.[[Get]], *current*.[[Get]]) is **false**.
12. For each attribute field of *Desc* that is present, set the correspondingly named attribute of the property named *P* of object *O* to the value of the field.
13. Return **true**.

**Deleted:** <#>ReturnIfAbrupt(*O*).¶

However, if  $O$  has an `[[NativeBrand]]` internal property whose value is `NativeArray`,  $O$  also has a more elaborate `[[DefineOwnProperty]]` internal method defined in 15.4.5.1.

**NOTE** Step 10.b allows any field of  $\text{Desc}$  to be different from the corresponding field of  $\text{current}$  if  $\text{current}[[\text{Configurable}]]$  field is `true`. This even permits changing the `[[Value]]` of a property whose `[[Writable]]` attribute is `false`. This is allowed because a `true` `[[Configurable]]` attribute would permit an equivalent sequence of calls where `[[Writable]]` is first set to `true`, a new `[[Value]]` is set, and then `[[Writable]]` is set to `false`.

**Deleted:** isArray

**Deleted:** is an Array object,

**Deleted:** it

#### 8.12.10 `[[Enumerate]] (includePrototype, onlyEnumerable)`

When the `[[Enumerate]]` internal method of  $O$  is called with Boolean arguments `includePrototype` and `onlyEnumerable`, the following steps are taken:

1. Return an Iterator object (reference `xxxx`) whose next method iterates over all the keys of enumerable property keys of  $O$ . If `includePrototype` is `false`, then only own properties of  $O$  are included. If `onlyEnumerable` is `false`, then all properties that do not have private name keys are included. The mechanics and order of enumerating the properties is not specified but must conform to the rules specified below.

**Comment [AWB65]:** This is an experiment to see if we can use this internal method to define `Object.key`, `Object.getOwnPropertyNames`, and perhaps some other things

**Comment [AWB66]:** TODO

Enumerated properties do not include properties whose property key is a private name. Properties of the object being enumerated may be deleted during enumeration. If a property that has not yet been visited during enumeration is deleted, then it will not be visited. If new properties are added to the object being enumerated during enumeration, the newly added properties are not guaranteed to be visited in the active enumeration. A property name must not be visited more than once in any enumeration.

Enumerating the properties of an object includes enumerating properties of its prototype, and the prototype of the prototype, and so on, recursively; but a property of a prototype is not enumerated if it is "shadowed" because some previous object in the prototype chain has a property with the same name. The values of `[[Enumerable]]` attributes are not considered when determining if a property of a prototype object is shadowed by a previous object on the prototype chain.

The following is an informative algorithm that conforms to these rules

1. Let  $obj$  be  $O$ .
2. Let  $proto$  be the value of the `[[Prototype]]` internal property of  $O$ .
3. If `includePrototype` is `false` or  $proto$  is the value `null`, then
  - a. Let  $propList$  be a new empty List.
4. Else
  - a. Let  $propList$  be the result of calling the `[[Enumerate]]` internal method of  $proto$  with arguments `true` and `onlyEnumerable`.
5. For each string  $name$  that is the property key of an own property of  $O$ 
  - a. Let  $desc$  be the result of calling the `[[GetOwnProperty]]` internal method of  $O$  with argument  $name$ .
  - b. If  $name$  is an element of  $propList$ , then remove  $name$  as an element of  $propList$ .
  - c. If `onlyEnumerable` is `false` or  $desc[[\text{Enumerable}]]$  is `true`, then add  $name$  as an element of  $propList$ .
6. Order the elements of  $propList$  in an implementation defined order.
7. Return  $propList$ .

**Comment [AWB67]:** TODO: Finish this up, and turn into iterator definition include a next method

**Deleted:** op

#### 8.12.11 `[[Iterate]] ()`

When the `[[Iterate]]` internal method of  $O$  is called the following steps are taken:

1. Let  $itr$  be the result of performing `Invoke` with arguments `%iterator%`,  $O$  and an empty argument List.
2. Return  $itr$ .

**Comment [AWB68]:** Notation for private names known to the implementation

## 9 Abstract Operations

These operations are not a part of the ECMAScript language; they are defined here to solely to aid the specification of the semantics of the ECMAScript Language.

## 9.1 Type Conversion and Testing

The ECMAScript language implicitly performs automatic type conversion as needed. To clarify the semantics of certain constructs it is useful to define a set of conversion abstract operations. The conversion abstract operations are polymorphic; that is, they can accept a value of any ECMAScript language type, but not of specification types.

### 9.1.1 ToPrimitive

The abstract operation `ToPrimitive` takes an input `argument` and an optional argument `PreferredType`. The abstract operation `ToPrimitive` converts its `input` argument to a non-Object type. If an object is capable of converting to more than one primitive type, it may use the optional hint `PreferredType` to favour that type. Conversion occurs according to Table 11:

Table 11 — `ToPrimitive` Conversions

<b>Input Type</b>	<b>Result</b>
Completion Record	If <code>argument</code> is an abrupt completion, return <code>argument</code> . Otherwise return <code>ToPrimitive(argument.[[value]])</code> .
Undefined	Return <code>argument</code> (no conversion).
Null	Return <code>argument</code> (no conversion).
Boolean	Return <code>argument</code> (no conversion).
Number	Return <code>argument</code> (no conversion).
String	Return <code>argument</code> (no conversion).
Object	<p>Perform the following steps:</p> <ol style="list-style-type: none"> <li>Let <code>default</code> be the result of calling the <code>[[DefaultValue]]</code> internal method of <code>argument</code>, passing the optional hint <code>PreferredType</code>.</li> <li>Return <code>ToPrimitive(default)</code>.</li> </ol> <p>The behaviour of the <code>[[DefaultValue]]</code> internal method is defined by this specification for all native ECMAScript objects in 8.12.8.</p>

### 9.1.2 ToBoolean

The abstract operation `ToBoolean` converts its `argument` to a value of type Boolean according to Table 12:

Table 12 — `ToBoolean` Conversions

<b>Argument Type</b>	<b>Result</b>
Completion Record	If <code>argument</code> is an abrupt completion, return the argument. Otherwise return <code>ToBoolean(argument.[[value]])</code> .
Undefined	Return <code>false</code> .
Null	Return <code>false</code> .
Boolean	Return the input argument (no conversion).
Number	Return <code>false</code> if the argument is <code>+0</code> , <code>-0</code> , or <code>Nan</code> ; otherwise return <code>true</code> .
String	Return <code>false</code> if the argument is the empty String (its length is zero); otherwise return <code>true</code> .
Object	Return <code>true</code> .

### 9.1.3 ToNumber

The abstract operation ToNumber converts its *argument* to a value of type Number according to Table 13:

Table 13 — To Number Conversions

Argument Type	Result
Completion Record	If <i>argument</i> is an abrupt completion, return <i>argument</i> . Otherwise return ToNumber( <i>argument</i> .[ <i>value</i> ]).
Undefined	Return NaN
Null	Return +0
Boolean	Return 1 if <i>argument</i> is true. Return +0 if <i>argument</i> is false.
Number	Return <i>argument</i> (no conversion).
String	See grammar and note below.
Object	Apply the following steps: 1. Let <i>primValue</i> be ToPrimitive( <i>argument</i> , hint Number). 2. Return ToNumber( <i>primValue</i> ).

Deleted: 12

Deleted: 12

Deleted: The result is

Deleted: the

Deleted: The result is

Deleted: the

Deleted: The result equals

Deleted: the *input*

Deleted: *input*

#### 9.1.3.1 ToNumber Applied to the String Type

ToNumber applied to Strings applies the following grammar to the input String. If the grammar cannot interpret the String as an expansion of *StringNumericLiteral*, then the result of ToNumber is NaN.

##### Syntax

```

StringNumericLiteral ::=
  StrWhiteSpaceopt
  StrWhiteSpaceopt StrNumericLiteral StrWhiteSpaceopt

StrWhiteSpace ::=
  StrWhiteSpaceChar StrWhiteSpaceopt

StrWhiteSpaceChar ::=
  WhiteSpace
  LineTerminator

StrNumericLiteral ::=
  StrDecimalLiteral
  HexIntegerLiteral

StrDecimalLiteral ::=
  StrUnsignedDecimalLiteral
  + StrUnsignedDecimalLiteral
  - StrUnsignedDecimalLiteral

StrUnsignedDecimalLiteral ::=
  Infinity
  DecimalDigits . DecimalDigitsopt ExponentPartopt
  . DecimalDigits ExponentPartopt
  DecimalDigits ExponentPartopt

DecimalDigits ::=
  DecimalDigit
  DecimalDigits DecimalDigit

DecimalDigit ::= one of
  0 1 2 3 4 5 6 7 8 9

```

*ExponentPart :::*  
*ExponentIndicator SignedInteger*

*ExponentIndicator :::* one of  
*e   E*

*SignedInteger :::*  
*DecimalDigits*  
*+ DecimalDigits*  
*- DecimalDigits*

*HexIntegerLiteral :::*  
*0x HexDigit*  
*0X HexDigit*  
*HexIntegerLiteral HexDigit*

*HexDigit :::* one of  
*0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F*

Some differences should be noted between the syntax of a *StringNumericLiteral* and a *NumericLiteral* (see 7.8.3):

- A *StringNumericLiteral* may be preceded and/or followed by white space and/or line terminators.
- A *StringNumericLiteral* that is decimal may have any number of leading 0 digits.
- A *StringNumericLiteral* that is decimal may be preceded by + or - to indicate its sign.
- A *StringNumericLiteral* that is empty or contains only white space is converted to +0.

The conversion of a String to a Number value is similar overall to the determination of the Number value for a numeric literal (see 7.8.3), but some of the details are different, so the process for converting a String numeric literal to a value of Number type is given here in full. This value is determined in two steps: first, a mathematical value (MV) is derived from the String numeric literal; second, this mathematical value is rounded as described below.

- The MV of *StringNumericLiteral* :: [empty] is 0.
- The MV of *StringNumericLiteral* :: *StrWhiteSpace* is 0.
- The MV of *StringNumericLiteral* :: *StrWhiteSpace*<sub>opt</sub> *StringNumericLiteral* *StrWhiteSpace*<sub>opt</sub> is the MV of *StringNumericLiteral*, no matter whether white space is present or not.
- The MV of *StringNumericLiteral* :: *StrDecimalLiteral* is the MV of *StrDecimalLiteral*.
- The MV of *StringNumericLiteral* :: *HexIntegerLiteral* is the MV of *HexIntegerLiteral*.
- The MV of *StringNumericLiteral* :: *StrUnsignedDecimalLiteral* is the MV of *StrUnsignedDecimalLiteral*.
- The MV of *StringNumericLiteral* :: + *StrUnsignedDecimalLiteral* is the MV of *StrUnsignedDecimalLiteral*.
- The MV of *StringNumericLiteral* :: - *StrUnsignedDecimalLiteral* is the negative of the MV of *StrUnsignedDecimalLiteral*. (Note that if the MV of *StrUnsignedDecimalLiteral* is 0, the negative of this MV is also 0. The rounding rule described below handles the conversion of this signless mathematical zero to a floating-point +0 or -0 as appropriate.)
- The MV of *StringNumericLiteral* :: **Infinity** is  $10^{10000}$  (a value so large that it will round to +∞).
- The MV of *StringNumericLiteral* :: *DecimalDigits*. is the MV of *DecimalDigits*.
- The MV of *StringNumericLiteral* :: *DecimalDigits*. *DecimalDigits* is the MV of the first *DecimalDigits* plus (the MV of the second *DecimalDigits* times  $10^{-n}$ ), where *n* is the number of characters in the second *DecimalDigits*.
- The MV of *StringNumericLiteral* :: *DecimalDigits*. *ExponentPart* is the MV of *DecimalDigits* times  $10^e$ , where *e* is the MV of *ExponentPart*.
- The MV of *StringNumericLiteral* :: *DecimalDigits*. *DecimalDigits* *ExponentPart* is (the MV of the first *DecimalDigits* plus (the MV of the second *DecimalDigits* times  $10^{-n}$ )) times  $10^e$ , where *n* is the number of characters in the second *DecimalDigits* and *e* is the MV of *ExponentPart*.
- The MV of *StringNumericLiteral* :: *DecimalDigits* is the MV of *DecimalDigits* times  $10^{-n}$ , where *n* is the number of characters in *DecimalDigits*.

- The MV of *StrUnsignedDecimalLiteral*:::. *DecimalDigits ExponentPart* is the MV of *DecimalDigits* times  $10^{e-n}$ , where  $n$  is the number of characters in *DecimalDigits* and  $e$  is the MV of *ExponentPart*.
- The MV of *StrUnsignedDecimalLiteral*:::. *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *StrUnsignedDecimalLiteral*:::. *DecimalDigits ExponentPart* is the MV of *DecimalDigits* times  $10^e$ , where  $e$  is the MV of *ExponentPart*.
- The MV of *DecimalDigits* :::. *DecimalDigit* is the MV of *DecimalDigit*.
- The MV of *DecimalDigits* :::. *DecimalDigits DecimalDigit* is (the MV of *DecimalDigits* times 10) plus the MV of *DecimalDigit*.
- The MV of *ExponentPart* :::. *ExponentIndicator SignedInteger* is the MV of *SignedInteger*.
- The MV of *SignedInteger* :::. *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger* :::. + *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger* :::. - *DecimalDigits* is the negative of the MV of *DecimalDigits*.
- The MV of *DecimalDigit* :::. 0 or of *HexDigit* :::. 0 is 0.
- The MV of *DecimalDigit* :::. 1 or of *HexDigit* :::. 1 is 1.
- The MV of *DecimalDigit* :::. 2 or of *HexDigit* :::. 2 is 2.
- The MV of *DecimalDigit* :::. 3 or of *HexDigit* :::. 3 is 3.
- The MV of *DecimalDigit* :::. 4 or of *HexDigit* :::. 4 is 4.
- The MV of *DecimalDigit* :::. 5 or of *HexDigit* :::. 5 is 5.
- The MV of *DecimalDigit* :::. 6 or of *HexDigit* :::. 6 is 6.
- The MV of *DecimalDigit* :::. 7 or of *HexDigit* :::. 7 is 7.
- The MV of *DecimalDigit* :::. 8 or of *HexDigit* :::. 8 is 8.
- The MV of *DecimalDigit* :::. 9 or of *HexDigit* :::. 9 is 9.
- The MV of *HexDigit* :::. a or of *HexDigit* :::. A is 10.
- The MV of *HexDigit* :::. b or of *HexDigit* :::. B is 11.
- The MV of *HexDigit* :::. c or of *HexDigit* :::. C is 12.
- The MV of *HexDigit* :::. d or of *HexDigit* :::. D is 13.
- The MV of *HexDigit* :::. e or of *HexDigit* :::. E is 14.
- The MV of *HexDigit* :::. f or of *HexDigit* :::. F is 15.
- The MV of *HexIntegerLiteral* :::. 0x *HexDigit* is the MV of *HexDigit*.
- The MV of *HexIntegerLiteral* :::. 0X *HexDigit* is the MV of *HexDigit*.
- The MV of *HexIntegerLiteral* :::. *HexIntegerLiteral HexDigit* is (the MV of *HexIntegerLiteral* times 16) plus the MV of *HexDigit*.

Once the exact MV for a String numeric literal has been determined, it is then rounded to a value of the Number type. If the MV is 0, then the rounded value is +0 unless the first non white space character in the String numeric literal is ‘-’, in which case the rounded value is -0. Otherwise, the rounded value must be the Number value for the MV (in the sense defined in 8.5), unless the literal includes a *StrUnsignedDecimalLiteral* and the literal has more than 20 significant digits, in which case the Number value may be either the Number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit or the Number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit and then incrementing the literal at the 20th digit position. A digit is *significant* if it is not part of an *ExponentPart* and

- it is not 0; or
- there is a nonzero digit to its left and there is a nonzero digit, not in the *ExponentPart*, to its right.

#### 9.1.4 ToInteger

The abstract operation *ToInteger* converts its argument to an integral numeric value. This abstract operation functions as follows:

1. Let *number* be the result of calling *ToNumber* on the input argument.
2. [ReturnIfAbrupt\(\*number\*\)](#).
3. If *number* is NaN, return +0.
4. If *number* is +0, -0, +∞, or -∞, return *number*.

5. Return the result of computing  $\text{sign}(\text{number}) \times \text{floor}(\text{abs}(\text{number}))$ .

### **9.1.5 ToInt32: (Signed 32 Bit Integer)**

The abstract operation ToInt32 converts its argument to one of  $2^{32}$  integer values in the range  $-2^{31}$  through  $2^{31}-1$ , inclusive. This abstract operation functions as follows:

1. Let  $\text{number}$  be the result of calling ToNumber on the input argument.
2. [ReturnIfAbrupt\( \$\text{number}\$ \)](#).
3. If  $\text{number}$  is **NaN**, **+0**, **-0**,  **$+\infty$** , or  **$-\infty$** , return **+0**.
4. Let  $\text{posInt}$  be  $\text{sign}(\text{number}) \times \text{floor}(\text{abs}(\text{number}))$ .
5. Let  $\text{int32bit}$  be  $\text{posInt}$  modulo  $2^{32}$ ; that is, a finite integer value  $k$  of Number type with positive sign and less than  $2^{32}$  in magnitude such that the mathematical difference of  $\text{posInt}$  and  $k$  is mathematically an integer multiple of  $2^{32}$ .
6. If  $\text{int32bit}$  is greater than or equal to  $2^{31}$ , return  $\text{int32bit} - 2^{32}$ , otherwise return  $\text{int32bit}$ .

NOTE Given the above definition of ToInt32:

- The ToInt32 abstract operation is idempotent: if applied to a result that it produced, the second application leaves that value unchanged.
- ToInt32( $\text{ToUint32}(x)$ ) is equal to  $\text{ToInt32}(x)$  for all values of  $x$ . (It is to preserve this latter property that  $+\infty$  and  $-\infty$  are mapped to **+0**.)
- ToInt32 maps **-0** to **+0**.

### **9.1.6 ToUint32: (Unsigned 32 Bit Integer)**

The abstract operation ToUint32 converts its argument to one of  $2^{32}$  integer values in the range 0 through  $2^{32}-1$ , inclusive. This abstraction operation functions as follows:

1. Let  $\text{number}$  be the result of calling ToNumber on the input argument.
2. [ReturnIfAbrupt\( \$\text{number}\$ \)](#).
3. If  $\text{number}$  is **NaN**, **+0**, **-0**,  **$+\infty$** , or  **$-\infty$** , return **+0**.
4. Let  $\text{posInt}$  be  $\text{sign}(\text{number}) \times \text{floor}(\text{abs}(\text{number}))$ .
5. Let  $\text{int32bit}$  be  $\text{posInt}$  modulo  $2^{32}$ ; that is, a finite integer value  $k$  of Number type with positive sign and less than  $2^{32}$  in magnitude such that the mathematical difference of  $\text{posInt}$  and  $k$  is mathematically an integer multiple of  $2^{32}$ .
6. Return  $\text{int32bit}$ .

NOTE Given the above definition of ToUint32:

- Step 5 is the only difference between ToUint32 and ToInt32.
- The ToUint32 abstract operation is idempotent: if applied to a result that it produced, the second application leaves that value unchanged.
- ToUint32( $\text{ToInt32}(x)$ ) is equal to  $\text{ToUint32}(x)$  for all values of  $x$ . (It is to preserve this latter property that  $+\infty$  and  $-\infty$  are mapped to **+0**.)
- ToUint32 maps **-0** to **+0**.

### **9.1.7 ToUint16: (Unsigned 16 Bit Integer)**

The abstract operation ToUint16 converts its argument to one of  $2^{16}$  integer values in the range 0 through  $2^{16}-1$ , inclusive. This abstract operation functions as follows:

1. Let  $\text{number}$  be the result of calling ToNumber on the input argument.
2. [ReturnIfAbrupt\( \$\text{number}\$ \)](#).
3. If  $\text{number}$  is **NaN**, **+0**, **-0**,  **$+\infty$** , or  **$-\infty$** , return **+0**.
4. Let  $\text{posInt}$  be  $\text{sign}(\text{number}) \times \text{floor}(\text{abs}(\text{number}))$ .
5. Let  $\text{int16bit}$  be  $\text{posInt}$  modulo  $2^{16}$ ; that is, a finite integer value  $k$  of Number type with positive sign and less than  $2^{16}$  in magnitude such that the mathematical difference of  $\text{posInt}$  and  $k$  is mathematically an integer multiple of  $2^{16}$ .
6. Return  $\text{int16bit}$ .

NOTE Given the above definition of ToUint16:

- The substitution of  $2^{16}$  for  $2^{32}$  in step 4 is the only difference between ToUint32 and ToUint16.
- ToUint16 maps  $-0$  to  $+0$ .

### 9.1.8 ToString

The abstract operation ToString converts its *argument* to a value of type String according to Table 14:

Table 14 — ToString Conversions

Argument Type	Result
Completion Record	If <i>argument</i> is an abrupt completion, return <i>argument</i> . Otherwise return ToString( <i>argument</i> .[[value]]).
Undefined	"undefined"
Null	"null"
Boolean	If <i>argument</i> is true, then return "true". If <i>argument</i> is false, then return "false".
Number	See 9.8.1.
String	Return <i>argument</i> (no conversion)
Object	Apply the following steps: 1. Let <i>primValue</i> be ToPrimitive( <i>argument</i> , hint String). 2. Return ToString( <i>primValue</i> ).

#### 9.1.8.1 ToString Applied to the Number Type

The abstract operation ToString converts a Number *m* to String format as follows:

- If *m* is NaN, return the String "NaN".
- If *m* is +0 or -0, return the String "0".
- If *m* is less than zero, return the String concatenation of the String "-" and ToString(-*m*).
- If *m* is infinity, return the String "Infinity".
- Otherwise, let *n*, *k*, and *s* be integers such that  $k \geq 1$ ,  $10^{k-1} \leq s < 10^k$ , the Number value for  $s \times 10^{n-k}$  is *m*, and *k* is as small as possible. Note that *k* is the number of digits in the decimal representation of *s*, that *s* is not divisible by 10, and that the least significant digit of *s* is not necessarily uniquely determined by these criteria.
- If  $k \leq n \leq 21$ , return the String consisting of the *k* digits of the decimal representation of *s* (in order, with no leading zeroes), followed by  $n-k$  occurrences of the character '0'.
- If  $0 < n \leq 21$ , return the String consisting of the most significant *n* digits of the decimal representation of *s*, followed by a decimal point '.', followed by the remaining  $k-n$  digits of the decimal representation of *s*.
- If  $-6 < n \leq 0$ , return the String consisting of the character '0', followed by a decimal point '.', followed by  $-n$  occurrences of the character '0', followed by the *k* digits of the decimal representation of *s*.
- Otherwise, if *k* = 1, return the String consisting of the single digit of *s*, followed by lowercase character 'e', followed by a plus sign '+' or minus sign '-' according to whether *n*-1 is positive or negative, followed by the decimal representation of the integer abs(*n*-1) (with no leading zeroes).
- Return the String consisting of the most significant digit of the decimal representation of *s*, followed by a decimal point '.', followed by the remaining *k*-1 digits of the decimal representation of *s*, followed by the lowercase character 'e', followed by a plus sign '+' or minus sign '-' according to whether *n*-1 is positive or negative, followed by the decimal representation of the integer abs(*n*-1) (with no leading zeroes).

NOTE 1 The following observations may be useful as guidelines for implementations, but are not part of the normative requirements of this Standard:

- If *x* is any Number value other than -0, then ToNumber(ToString(*x*)) is exactly the same Number value as *x*.
- The least significant digit of *s* is not always uniquely determined by the requirements listed in step 5.

NOTE 2 For implementations that provide more accurate conversions than required by the rules above, it is recommended that the following alternative version of step 5 be used as a guideline:

Deleted: 13

Deleted: 13

Deleted: the

Deleted: the result is

Deleted: the

Deleted: the result is

Deleted: the input

Deleted: input

Formatted: Outline numbered + Level: 1 + Numbering Style: 1, 2, 3, ... + Start at: 1 + Alignment: Left + Aligned at: 0" + Tab after: 0.25" + Indent at: 0.25"

Otherwise, let  $n$ ,  $k$ , and  $s$  be integers such that  $k \geq 1$ ,  $10^{k-1} \leq s < 10^k$ , the Number value for  $s \times 10^{n-k}$  is  $m$ , and  $k$  is as small as possible. If there are multiple possibilities for  $s$ , choose the value of  $s$  for which  $s \times 10^{n-k}$  is closest in value to  $m$ . If there are two such possible values of  $s$ , choose the one that is even. Note that  $k$  is the number of digits in the decimal representation of  $s$  and that  $s$  is not divisible by 10.

NOTE 3 Implementers of ECMAScript may find useful the paper and code written by David M. Gay for binary-to-decimal conversion of floating-point numbers:

Gay, David M. Correctly Rounded Binary-Decimal and Decimal-Binary Conversions. Numerical Analysis, Manuscript 90-10. AT&T Bell Laboratories (Murray Hill, New Jersey). November 30, 1990. Available as <http://cm.bell-labs.com/cm/cs/doc/90/4-10.ps.gz>. Associated code available as <http://cm.bell-labs.com/netlib/fp/dtoa.c.gz> and as [http://cm.bell-labs.com/netlib/fp/g\\_fmt.c.gz](http://cm.bell-labs.com/netlib/fp/g_fmt.c.gz) and may also be found at the various `netlib` mirror sites.

### 9.1.9 ToObject

The abstract operation `ToObject` converts its *argument* to a value of type `Object` according to Table 15:

Table 15 — `ToObject`

Argument Type	Result
Completion Record	If <i>argument</i> is an abrupt completion, return <i>argument</i> . Otherwise return <code>ToObject(argument.[[value]])</code> .
Undefined	Throw a <b>TypeError</b> exception.
Null	Throw a <b>TypeError</b> exception.
Boolean	Return a new Boolean object whose <code>[[PrimitiveValue]]</code> internal property is set to the value of <i>argument</i> . See 15.6 for a description of Boolean objects.
Number	Return a new Number object whose <code>[[PrimitiveValue]]</code> internal property is set to the value of <i>argument</i> . See 15.7 for a description of Number objects.
String	Return a new String object whose <code>[[PrimitiveValue]]</code> internal property is set to the value of <i>argument</i> . See 15.5 for a description of String objects.
Object	Return <i>argument</i> (no conversion).

### 9.1.10 ToPropertyKey

The abstract operation `ToPropertyKey` converts its *argument* to a value that can be used as a property key by performing the following steps:

1. `ReturnIfAbrupt(argument)`.
2. If `Type(argument)` is `Object`, then
  - a. If *argument* has a `[[NativeBrand]]` internal property whose value is `NativePrivateName`, then
    - i. Return *argument*.
3. Return `ToString(argument)`.

## 9.2 Testing and Comparison Operations

### 9.2.1 CheckObjectCoercible

The abstract operation `CheckObjectCoercible` throws an error if its argument is a value that cannot be converted to an `Object` using `ToObject`. It is defined by Table 16:

**Formatted:** Heading 3

**Deleted:** 14

**Deleted:** 14

**Deleted:** Create

**Deleted:** the

**Deleted:** Create

**Deleted:** the

**Deleted:** Create

**Deleted:** the

**Deleted:** The result is

**Deleted:** the input

**Formatted:** Space Before: 10 pt

**Deleted:** 0

**Deleted:** 15

**Table 16 — CheckObjectCoercible Results**

**Deleted: 15**

Argument Type	Result
Completion Record	If <i>argument</i> is an abrupt completion, return <i>argument</i> . Otherwise return <i>CheckObjectCoercible(argument.[[value]])</i> .
Undefined	Throw a <b>TypeError</b> exception.
Null	Throw a <b>TypeError</b> exception.
Boolean	Return <i>argument</i>
Number	Return <i>argument</i>
String	Return <i>argument</i>
Object	Return <i>argument</i>

### 9.2.2 IsCallable

The abstract operation *IsCallable* determines if its *argument*, which must be an ECMAScript language value, is a callable function Object according to Table 17:

**Table 17 — IsCallable Results**

**Deleted: 11**

**Deleted: 16**

**Deleted: 16**

Argument Type	Result
Completion Record	If <i>argument</i> is an abrupt completion, return <i>argument</i> . Otherwise return <i>IsCallable(argument.[[value]])</i> .
Undefined	Return <b>false</b> .
Null	Return <b>false</b> .
Boolean	Return <b>false</b> .
Number	Return <b>false</b> .
String	Return <b>false</b> .
Object	If <i>argument</i> has a <b>[[Call]]</b> internal method, then return <b>true</b> , otherwise return <b>false</b> .

### 9.2.3 The SameValue Algorithm

The internal comparison abstract operation *SameValue(x, y)*, where *x* and *y* are ECMAScript language values, produces **true** or **false**. Such a comparison is performed as follows:

1. ReturnIfAbrupt(x).
2. ReturnIfAbrupt(y).
3. If *Type(x)* is different from *Type(y)*, return **false**.
4. If *Type(x)* is **Undefined**, return **true**.
5. If *Type(x)* is **Null**, return **true**.
6. If *Type(x)* is **Number**, then
  - a. If *x* is **NaN** and *y* is **NaN**, return **true**.
  - b. If *x* is **+0** and *y* is **-0**, return **false**.
  - c. If *x* is **-0** and *y* is **+0**, return **false**.
  - d. If *x* is the same **Number** value as *y*, return **true**.
  - e. Return **false**.
7. If *Type(x)* is **String**, then return **true** if *x* and *y* are exactly the same sequence of characters (same length and same characters in corresponding positions); otherwise, return **false**.
8. If *Type(x)* is **Boolean**, return **true** if *x* and *y* are both **true** or both **false**; otherwise, return **false**.
9. Return **true** if *x* and *y* are the same **Object** value. Otherwise, return **false**.

**Deleted: the**

**Deleted: object**

**Deleted: 12**

**Deleted: <#>** If *x* is an Completion Record, let *x* be *x.[[value]]*. ¶  
**Deleted: <#>** If *y* is an Completion Record, let *y* be *y.[[value]]*. ¶

### 9.3 Property Access Operations

#### 9.3.1 Invoke

The *Invoke* abstraction abstract operation is used to call a method property of an object. The operation is called with arguments *P, O*, and *args* where *P* is the property key, *O* serves as both the lookup point for the

**Deleted: refer to**

**Deleted: object**

property and the `this` value of the call, and `args` is the list of arguments values passed to the method. This abstract operation perform, the following steps:

1. Let `obj` be `ToObject(O)`.
2. ReturnIfAbrupt(`obj`).
3. Let `func` be the result of calling the `[[Get]]` internal method of `obj` passing `P` as the argument.
4. ReturnIfAbrupt(`func`).
5. If `IsCallable(func)` is `false`, throw a `TypeError` exception.
6. Return the result of calling the `[[Call]]` internal method of `func` passing `O` as the `this` value and argument list `args`.

## 10 Executable Code and Execution Contexts

### 10.1 Types of Executable Code

There are three types of ECMAScript executable code:

- *Global code* is source text that is treated as an ECMAScript *Program*. The global code of a particular *Program* does not include any source text that is parsed as part of a *FunctionBody*.
- *Eval code* is the source text supplied to the built-in `eval` function. More precisely, if the parameter to the built-in `eval` function is a String, it is treated as an ECMAScript *Program*. The eval code for a particular invocation of `eval` is the global code portion of that *Program*.
- *Function code* is source text that is parsed to supply the value of the `[[Code]]` internal property (see 13.5) of function and generator objects. The *function code* of a particular *function or generator* does not include any source text that is parsed as the *function code* of a nested *function or generator*.

**NOTE** Function code is generally provided as the bodies of Function Definitions (13.1), Arrow Function Definitions (13.2), Method Definitions (13.3) and Generator Definitions (13.4). Function code is also derived from the last argument to the Function constructor (15.3).

#### 10.1.1 Strict Mode Code

An ECMAScript *Program* syntactic unit may be processed using either unrestricted or strict mode syntax and semantics. When processed using strict mode the three types of ECMAScript code are referred to as strict global code, strict eval code, and strict function code. Code is interpreted as strict mode code in the following situations:

- Global code is strict global code if it begins with a Directive Prologue that contains a Use Strict Directive (see 14.1).
- Eval code is strict eval code if it begins with a Directive Prologue that contains a Use Strict Directive or if the call to eval is a direct call (see 15.1.2.1.1) to the eval function that is contained in strict mode code.
- Function code that is part of a *FunctionDeclaration*, *FunctionExpression*, or accessor *PropertyDefinition* is strict function code if its *FunctionDeclaration*, *FunctionExpression*, or *PropertyDefinition* is contained in strict mode code or if the function code begins with a Directive Prologue that contains a Use Strict Directive.
- Function code that is supplied as the last argument to the built-in Function constructor is strict function code if the last argument is a String that when processed as a *FunctionBody* begins with a Directive Prologue that contains a Use Strict Directive.

The term "base code" is used to designate code that is not strict code.

**Comment [AWB79]:** TODO: may need an additional reference to the `[[Code]]` of a generator.

**Deleted:** as part of a *FunctionBody*.

**Deleted:** *FunctionBody*

**Deleted:** part

**Deleted:** *FunctionBody*

**Deleted:** *Function code* also denotes the source text supplied when using the built-in *Function* object as a constructor. More precisely, the last parameter provided to the *Function* constructor is converted to a String and treated as the *FunctionBody*. If more than one parameter is provided to the *Function* constructor, all parameters except the last one are converted to Strings and concatenated together, separated by commas. The resulting String is interpreted as the *FormalParameterList* for the *FunctionBody* defined by the last parameter. The function code for a particular instantiation of a *Function* does not include any source text that is parsed as part of a nested *FunctionBody*.

**Formatted:** Note

**Deleted:** *PropertyAssignment*

**Deleted:** *PropertyAssignment*

**Comment [AWB10]:** Need to replace this with a statement about module bodies being strict code

**Deleted:** <#> Unless specified otherwise, extended code (10.21.12) is also considered strict mode code.¶

**Deleted: 10.1.2 Extended Code¶**

Extended code is any any code contained in ECMAScript *Program* syntactic unit that contains occurrences of lexical or syntactic production defined subsequent to the Fifth Edition of the ECMAScript specification. Code is interpreted as extended code in the following situations:  
 ↪ Global code is extended global code if it is contained in an ECMAScript *Program* syntactic unit that has been designated as an extended *Program* unit in an implementation defined manner or if ???¶

↪ Eval code is extended eval code if the call to eval is a direct call (see 15.1.2.1.1) to the eval function that is contained in extended mode code or if it begins with ???¶

↪ Function code that is part of a *FunctionDeclaration*, *FunctionExpression*, or accessor *PropertyAssignment* is extended function code if its *FunctionDeclaration*, *FunctionExpression*, or *PropertyAssignment* is contained in extended mode code or if the function code begins with ???¶

↪ Function code that is supplied as the last argument to the built-in Function constructor is strict function code if the last argument is a String that when processed as a *FunctionBody* begins with ???¶

The term "strict code" includes used to designate both actual strict mode code and extended code while the term "extended code" is used only to mean actual extended code.

**Deleted:** extended

### 10.1.2 Code Realms

Before it is evaluated, all ECMAScript code must be associated with a *Realm*. Conceptually, a realm consists as of an set of intrinsic objects, an ECMAScript global environment, all of the ECMAScript code that is loaded within the scope of that global environment, a Loader object that can associate new ECMAScript code with the realm, and other associated state.

A Realm can also be v

## 10.2 Lexical Environments

A *Lexical Environment* is a specification type used to define the association of *Identifiers* to specific variables and functions based upon the lexical nesting structure of ECMAScript code. A Lexical Environment consists of an Environment Record and a possibly null reference to an outer Lexical Environment. Usually a Lexical Environment is associated with some specific syntactic structure of ECMAScript code such as a *FunctionDeclaration*, a *BlockStatement*, or a *Catch* clause of a *TryStatement* and a new Lexical Environment is created each time such code is evaluated.

An *Environment Record* records the identifier bindings that are created within the scope of its associated Lexical Environment.

The outer environment reference is used to model the logical nesting of Lexical Environment values. The outer reference of a (inner) Lexical Environment is a reference to the Lexical Environment that logically surrounds the inner Lexical Environment. An outer Lexical Environment may, of course, have its own outer Lexical Environment. A Lexical Environment may serve as the outer environment for multiple inner Lexical Environments. For example, if a *FunctionDeclaration* contains two nested *FunctionDeclarations* then the Lexical Environments of each of the nested functions will have as their outer Lexical Environment the Lexical Environment of the current execution of the surrounding function.

A *global environment* is a Lexical Environment which does not have an outer environment. The global environment's outer environment reference is null. A global environment's environment record may be prepopulated with identifier bindings and includes an associated *global object* whose properties provide some of the global environment's identifier bindings. This global object is the value of a global environment's *this* bindingAs ECMAScript code is executed, additional properties may be added to the global object and the initial properties may be modified.

A method environment is a Lexical Environment that corresponds to the invocation of an ECMAScript function object that establishes a new *this* binding. A method environment also captures the state necessary to support the *super* method invocations.

Lexical Environments and Environment Record values are purely specification mechanisms and need not correspond to any specific artefact of an ECMAScript implementation. It is impossible for an ECMAScript program to directly access or manipulate such values.

### 10.2.1 Environment Records

There are two kinds of Environment Record values used in this specification: *declarative environment records* and *object environment records*. Declarative environment records are used to define the effect of ECMAScript language syntactic elements such as *FunctionDeclarations*, *VariableDeclarations*, and *Catch* clauses that directly associate identifier bindings with ECMAScript language values. Object environment records are used to define the effect of ECMAScript elements such as *Program* and *WithStatement* that associate identifier bindings with the properties of some object.

For specification purposes Environment Record values can be thought of as existing in a simple object-oriented hierarchy where Environment Record is an abstract class with two concrete subclasses, declarative environment record and object environment record. The abstract class includes the abstract specification methods defined in Table 18. These abstract methods have distinct concrete algorithms for each of the concrete subclasses.

**Deleted:** WithStatement

**Deleted:** 17

**Table 18 — Abstract Methods of Environment Records**

**Deleted: 17**

<b>Method</b>	<b>Purpose</b>
HasBinding(N)	Determine if an environment record has a binding for an identifier. Return <b>true</b> if it does and <b>false</b> if it does not. The String value <i>N</i> is the text of the identifier.
CreateMutableBinding(N, D)	Create a new mutable binding in an environment record. The String value <i>N</i> is the text of the bound name. If the optional Boolean argument <i>D</i> is <b>true</b> the binding is may be subsequently deleted.
SetMutableBinding(N,V, S)	Set the value of an already existing mutable binding in an environment record. The String value <i>N</i> is the text of the bound name. <i>V</i> is the value for the binding and may be a value of any ECMAScript language type. <i>S</i> is a Boolean flag. If <i>S</i> is <b>true</b> and the binding cannot be set throw a <b>TypeError</b> exception. <i>S</i> is used to identify strict mode references.
GetBindingValue(N,S)	Returns the value of an already existing binding from an environment record. The String value <i>N</i> is the text of the bound name. <i>S</i> is used to identify strict mode references. If <i>S</i> is <b>true</b> and the binding does not exist or is uninitialised throw a <b>ReferenceError</b> exception.
DeleteBinding(N)	Delete a binding from an environment record. The String value <i>N</i> is the text of the bound name If a binding for <i>N</i> exists, remove the binding and return <b>true</b> . If the binding exists but cannot be removed return <b>false</b> . If the binding does not exist return <b>true</b> .
<u>CreateVarBinding(N, D)</u>	<u>Used for global <code>var</code> and <code>function</code> bindings. Create a new mutable binding in an environment record. The String value <i>N</i> is the text of the bound name. If the optional Boolean argument <i>D</i> is <b>true</b> the binding is may be subsequently deleted. This is logically equivalent to CreateMutableBinding but it allows <code>var</code> and <code>function</code> declaration to receive special treatment.</u>
<u>HasThisBinding()</u>	<u>Determine if an environment record establishes a <code>this</code> binding. Return <b>true</b> if it does and <b>false</b> if it does not.</u>
<u>HasSuperBinding()</u>	<u>Determine if an environment record establishes a <code>super</code> method binding. Return <b>true</b> if it does and <b>false</b> if it does not.</u>
<u>WithBaseObject()</u>	<u>If this environment record is associated with a <code>with</code> statement, return the <code>with</code> object. Otherwise, return <code>undefined</code>.</u>

**Deleted: ImplicitThisValue**

**Formatted Table**

**Deleted:** Returns the value to use as the this value on calls to function objects that are obtained as binding values from this environment record.

**Deleted: immutable**

**Deleted: 18**

#### 10.2.1.1 Declarative Environment Records

Each declarative environment record is associated with an ECMAScript program scope containing variable, constant and/or function declarations. A declarative environment record binds the set of identifiers defined by the declarations contained within its scope.

In addition to the mutable bindings supported by all Environment Records, declarative environment records also provide for immutable bindings. An immutable binding is one where the association between an identifier and a value may not be modified once it has been established. Creation and initialisation of declarative binding are distinct steps so it is possible for such bindings to exist in either an initialised or uninitialised state. Declarative environment records support the methods listed in Table 19 in addition to the Environment Record abstract specification methods:

**Table 19 — Additional Methods of Declarative Environment Records**

Method	Purpose
CreateImmutableBinding(N)	Create a new but uninitialised immutable binding in an environment record. The String value <i>N</i> is the text of the bound name.
InitializeBinding(N,V)	Set the value of an already existing but uninitialised binding in an environment record. The String value <i>N</i> is the text of the bound name. <i>V</i> is the value for the binding and is a value of any ECMAScript language type.

**Deleted:** 18

**Formatted Table**

**Deleted:** Immutable

**Deleted:** immutable

The behaviour of the concrete specification methods for Declarative Environment Records is defined by the following algorithms.

#### 10.2.1.1.1 HasBinding(*N*)

The concrete environment record method HasBinding for declarative environment records simply determines if the argument identifier is one of the identifiers bound by the record:

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. If *envRec* has a binding for the name that is the value of *N*, return **true**.
3. If it does not have such a binding, return **false**.

#### 10.2.1.1.2 CreateMutableBinding (*N, D*)

The concrete Environment Record method CreateMutableBinding for declarative environment records creates a new mutable binding for the name *N* that is initialised to the value **undefined**. A binding must not already exist in this Environment Record for *N*. If Boolean argument *D* is provided and has the value **true** the new binding is marked as being subject to deletion.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Assert: *envRec* does not already have a binding for *N*.
3. Create a mutable binding in *envRec* for *N* and and record that it is uninitialised. If *D* is true record that the newly created binding may be deleted by a subsequent DeleteBinding call.

**Deleted:** set its bound value to **undefined**

#### 10.2.1.1.3 SetMutableBinding (*N,V,S*)

The concrete Environment Record method SetMutableBinding for declarative environment records attempts to change the bound value of the current binding of the identifier whose name is the value of the argument *N* to the value of argument *V*. A binding for *N* must already exist. If the binding is an immutable binding, a **TypeError** is thrown if *S* is **true**.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Assert: *envRec* must have a binding for *N*.
3. Assert: The binding for *N* in *envRec* has already been initialised.
4. If the binding for *N* in *envRec* is a mutable binding, change its bound value to *V*.
5. Else if binding for *N* in *envRec* has not yet been initialized throw a ReferenceError exception.
6. Else this must be an attempt to change the value of an immutable binding so if *S* is **true** throw a **TypeError** exception.

**Deleted:** if

#### 10.2.1.1.4 GetBindingValue(*N,S*)

The concrete Environment Record method GetBindingValue for declarative environment records simply returns the value of its bound identifier whose name is the value of the argument *N*. The binding must already exist. If *S* is **true** and the binding is an uninitialised immutable binding throw a **ReferenceError** exception.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Assert: *envRec* has a binding for *N*.
3. If the binding for *N* in *envRec* is an uninitialised binding, then

**Deleted:** immutable

- a. If  $S$  is **false**, return the value **undefined**, otherwise throw a **ReferenceError** exception.
4. Else, return the value currently bound to  $N$  in  $envRec$ .

#### **10.2.1.1.5 DeleteBinding (N)**

The concrete Environment Record method `DeleteBinding` for declarative environment records can only delete bindings that have been explicitly designated as being subject to deletion.

1. Let  $envRec$  be the declarative environment record for which the method was invoked.
2. If  $envRec$  does not have a binding for the name that is the value of  $N$ , return **true**.
3. If the binding for  $N$  in  $envRec$  is cannot be deleted, return **false**.
4. Remove the binding for  $N$  from  $envRec$ .
5. Return **true**.

#### **10.2.1.1.6 CreateVarBinding (N, D)**

The concrete Environment Record method `CreateVarBinding` for declarative environment records performs the same action as the `CreateMutableBinding` concrete method of the same environment Record.

1. Let  $envRec$  be the declarative environment record for which the method was invoked.
2. Return the result of calling  $envRec$ 's `CreateMutableBinding` concrete method with arguments  $N$  and  $D$ .

#### **10.2.1.1.7 HasThisBinding ()**

Regular Declarative Environment Records do not provide a `this` binding.

1. Return **false**.

#### **10.2.1.1.8 HasSuperBinding ()**

Regular Declarative Environment Records do not provide a `super` binding.

1. Return **false**.

#### **10.2.1.1.9 WithBaseObject()**

Declarative Environment Records always return **undefined** as their `WithBaseObject`.

1. Return **undefined**.

#### **10.2.1.1.10 CreateImmutableBinding (N)**

The concrete Environment Record method `CreateImmutableBinding` for declarative environment records creates a new immutable binding for the name  $N$  that is initialised to the value **undefined**. A binding must not already exist in this environment record for  $N$ .

1. Let  $envRec$  be the declarative environment record for which the method was invoked.
2. Assert:  $envRec$  does not already have a binding for  $N$ .
3. Create an immutable binding in  $envRec$  for  $N$  and record that it is uninitialised.

#### **10.2.1.1.11 InitializeBinding (N,V)**

The concrete Environment Record method `InitializeBinding` for declarative environment records is used to set the bound value of the current binding of the identifier whose name is the value of the argument  $N$  to the value of argument  $V$ . An uninitialised binding for  $N$  must already exist.

1. Let  $envRec$  be the declarative environment record for which the method was invoked.
2. Assert:  $envRec$  must have an uninitialised binding for  $N$ .
3. Set the bound value for  $N$  in  $envRec$  to  $V$ .
4. Record that the binding for  $N$  in  $envRec$  has been initialised.

**Deleted:** 6  
**Deleted:** ImplicitThisValue  
**Deleted:** ImplicitThisValue

**Comment [AWB11]:** This probably needs a option argument, just like `createMutableEnvironment`  
**Deleted:** 7

**Deleted:** 8  
**Deleted:** Immutable  
**Deleted:** Immutable  
**Deleted:** immutable  
**Deleted:** immutable  
**Deleted:** immutable

### 10.2.1.2 Object Environment Records

Each object environment record is associated with an object called its *binding object*. An object environment record binds the set of identifier names that directly correspond to the property names of its binding object. Property names that are not an *IdentifierName* are not included in the set of bound identifiers. Both own and inherited properties are included in the set regardless of the setting of their `[[Enumerable]]` attribute. Because properties can be dynamically added and deleted from objects, the set of identifiers bound by an object environment record may potentially change as a side-effect of any operation that adds or deletes properties. Any bindings that are created as a result of such a side-effect are considered to be a mutable binding even if the `Writable` attribute of the corresponding property has the value `false`. Immutable bindings do not exist for object environment records.

Object environment records [created for `with` statements \(12.10\)](#) can provide their binding object as an implicit this value for use in function calls. The capability is controlled by a `withEnvironment` Boolean value that is associated with each object environment record. By default, the value of `withEnvironment` is `false` for any object environment record.

The behaviour of the concrete specification methods for Object Environment Records is defined by the following algorithms.

#### 10.2.1.2.1 HasBinding(*N*)

The concrete Environment Record method `HasBinding` for object environment records determines if its associated binding object has a property whose name is the value of the argument *N*:

1. Let *envRec* be the object environment record for which the method was invoked.
2. Let *bindings* be the binding object for *envRec*.
3. Return the result of calling the `[[HasProperty]]` internal method of *bindings*, passing *N* as the property name.

#### 10.2.1.2.2 CreateMutableBinding (*N, D*)

The concrete Environment Record method `CreateMutableBinding` for object environment records creates in an environment record's associated binding object a property whose name is the String value and initialises it to the value `undefined`. A property named *N* must not already exist in the binding object. If Boolean argument *D* is provided and has the value `true` the new property's `[[Configurable]]` attribute is set to `true`, otherwise it is set to `false`.

1. Let *envRec* be the object environment record for which the method was invoked.
2. Let *bindings* be the binding object for *envRec*.
3. Assert: The result of calling the `[[HasProperty]]` internal method of *bindings*, passing *N* as the property name, is `false`.
4. If *D* is `true` then let *configValue* be `true` otherwise let *configValue* be `false`.
5. Call the `[[DefineOwnProperty]]` internal method of *bindings*, passing *N*, Property Descriptor `{[[Value]]:undefined, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: configValue}`, and *true* as arguments.

#### 10.2.1.2.3 SetMutableBinding (*N,V,S*)

The concrete Environment Record method `SetMutableBinding` for object environment records attempts to set the value of the environment record's associated binding object's property whose name is the value of the argument *N* to the value of argument *V*. A property named *N* should already exist but if it does not or is not currently writable, error handling is determined by the value of the Boolean argument *S*.

1. Let *envRec* be the object environment record for which the method was invoked.
2. Let *bindings* be the binding object for *envRec*.
3. Call the `[[Put]]` internal method of *bindings* with arguments *N*, *V*, and *S*.

**Deleted:** be configured to

**Formatted:** Font: (Default) Courier New, Bold

**Deleted:** This capability is used to specify the behavior of With Statement (12.10) induced bindings.

**Deleted:** *provideThis*

**Deleted:** *provideThis*

#### 10.2.1.2.4 GetBindingValue(N,S)

The concrete Environment Record method `GetBindingValue` for object environment records returns the value of its associated binding object's property whose name is the String value of the argument identifier `N`. The property should already exist but if it does not the result depends upon the value of the `S` argument:

1. Let `envRec` be the object environment record for which the method was invoked.
2. Let `bindings` be the binding object for `envRec`.
3. Let `value` be the result of calling the `[[HasProperty]]` internal method of `bindings`, passing `N` as the property name.
4. If `value` is `false`, then
  - a. If `S` is `false`, return the value `undefined`, otherwise throw a **ReferenceError** exception.
5. Return the result of calling the `[[Get]]` internal method of `bindings`, passing `N` for the argument.

#### 10.2.1.2.5 DeleteBinding (N)

The concrete Environment Record method `DeleteBinding` for object environment records can only delete bindings that correspond to properties of the environment object whose `[[Configurable]]` attribute have the value `true`.

1. Let `envRec` be the object environment record for which the method was invoked.
2. Let `bindings` be the binding object for `envRec`.
3. Return the result of calling the `[[Delete]]` internal method of `bindings`, passing `N` and `false` as arguments.

#### 10.2.1.1.6 CreateVarBinding (N, D)

The concrete Environment Record method `CreateVarBinding` for object environment records performs the same action as the `CreateMutableBinding` concrete method of the same environment Record.

1. Let `envRec` be the declarative environment record for which the method was invoked.
2. Return the result of calling `envRec`'s `CreateMutableBinding` concrete method with arguments `N` and `D`.

#### 10.2.1.2.7 HasThisBinding ()

Regular Object Environment Records do not provide a `this` binding.

1. Return `false`.

#### 10.2.1.2.8 HasSuperBinding ()

Regular Object Environment Records do not provide a `super` binding.

1. Return `false`.

#### 10.2.1.2.9 WithBaseObject()

Object Environment Records return `undefined` as their `WithBaseObject` unless their `withEnvironment` flag is `true`.

1. Let `envRec` be the object environment record for which the method was invoked.
2. If the `withEnvironment` flag of `envRec` is `true`, return the binding object for `envRec`.
3. Otherwise, return `undefined`.

**Deleted:** 6  
**Deleted:** ImplicitThisValue  
**Deleted:** ImplicitThisValue  
**Deleted:** provideThis  
**Deleted:** provideThis

#### 10.2.1.4 Method Environment Records

A method environment record is a declarative environment record that is used to represent the outer most scope of a function that provides a `this` binding. In addition to its identifier bindings, a method environment record contains the `this` value used within its scope. If such a function references `super`, its method environment record also contains the state that is used to perform `super` method invocations from within the function.

Method environment records store their `this` binding as the value of their `thisValue`. If the associated function references `super`, the environment record stores in `HomeObject` the object that the function is bound to as a method and in `MethodName` the property key used for unqualified super invocations from within the function. The default value for `HomeObject` and `MethodName` is `undefined`.

Methods environment records support all of Declarative Environment Record methods listed in Table 19 and share the same specifications for those methods. In addition, declarative environment records support the methods listed in Table 20:

**Table 20 — Additional Methods of Method Environment Records**

<u>Method</u>	<u>Purpose</u>
<code>GetThisBinding()</code>	Return the value of this environment record's <code>this</code> binding.
<code>GetHomeObject()</code>	Return the value of this environment record's <code>HomeObject</code> binding. If the value is Empty, return <code>undefined</code> .
<code>GetMethodname()</code>	Return the value of this environment record's <code>MethodName</code> binding.

The behaviour of the additional concrete specification methods for Method Environment Records is defined by the following algorithms:

#### **10.2.1.3.1 HasThisBinding ()**

Regular Method Environment Records always provide a `this` binding.

1. Return `true`.

#### **10.2.1.3.2 HasSuperBinding ()**

1. If this environment record's `HomeObject` has the value `Empty`, then return `false`. Otherwise, return `true`.

#### **10.2.1.3.3 GetThisBinding ()**

1. Return the value of this environment record's `thisValue`.

#### **10.2.1.3.4 GetHomeObject ()**

1. If this environment record's `HomeObject` has the value `Empty`, then return `undefined`. Otherwise, return the `HomeObject` value.

#### **10.2.1.3.5 GetMethodName ()**

1. Return the value of this environment record's `MethodName`.

#### **10.2.1.4 Global Environment Records**

A global environment is used to represent the outer most scope of a ECMA Script *Program*. If such a function references `super`, A global environment provides the bindings for built-in globals (15.1), properties of the global object, and for all declarations that are not function code and that occur within *Program* productions.. All ECMA Script *Program* productions that are processed in a specific Realm (10.3) share the same global environment record.

The behaviour of the concrete specification methods for Global Environment Records is defined by the following algorithms.

**Comment [AWB712]: TODO**

#### **10.2.2 Lexical Environment Operations**

The following abstract operations are used in this specification to operate upon lexical environments:

#### 10.2.2.1 GetIdentifierReference (lex, name, strict)

The abstract operation GetIdentifierReference is called with a Lexical Environment *lex*, a String *name*, and a Boolean flag *strict*. The value of *lex* may be **null**. When called, the following steps are performed:

**Deleted:** n  
**Deleted:** identifier

1. If *lex* is the value **null**, then
  - a. Return a value of type Reference whose base value is **undefined**, whose referenced name is *name*, and whose strict mode flag is *strict*.
2. Let *envRec* be *lex*'s environment record.
3. Let *exists* be the result of calling the HasBinding(*N*) concrete method of *envRec* passing *name* as the argument *N*.
4. If *exists* is **true**, then
  - a. Return a value of type Reference whose base value is *envRec*, whose referenced name is *name*, and whose strict mode flag is *strict*.
5. Else
  - a. Let *outer* be the value of *lex*'s outer environment reference.
  - b. Return the result of calling GetIdentifierReference passing *outer*, *name*, and *strict* as arguments.

#### 10.2.2.2 NewDeclarativeEnvironment (E)

When the abstract operation NewDeclarativeEnvironment is called with either a Lexical Environment or **null** as argument *E* the following steps are performed:

1. Let *env* be a new Lexical Environment.
2. Let *envRec* be a new declarative environment record containing no bindings.
3. Set *env*'s environment record to be *envRec*.
4. Set the outer lexical environment reference of *env* to *E*.
5. Return *env*.

#### 10.2.2.3 NewObjectEnvironment (O, E)

When the abstract operation NewObjectEnvironment is called with an Object *O* and a Lexical Environment *E* (or **null**) as arguments, the following steps are performed:

1. Let *env* be a new Lexical Environment.
2. Let *envRec* be a new object environment record containing *O* as the binding object.
3. Set *env*'s environment record to be *envRec*.
4. Set the outer lexical environment reference of *env* to *E*.
5. Return *env*.

#### 10.2.2.4 NewMethodEnvironment (F, T)

When the abstract operation NewObjectEnvironment is called with an ECMAScript function Object *F* and a ECMAScript value *T* as arguments, the following steps are performed:

1. Let *env* be a new Lexical Environment.
2. Let *envRec* be a new method environment record containing containing no bindings.
3. Set *envRec*'s *thisValue* to *T*.
4. If *F* has a **[[Home]]** internal property, then
  - a. Set *envRec*'s *HomeObject* to the value of *F*'s **[[Home]]** internal property.
  - b. Set *envRec*'s *MethodName* to the value of *F*'s **[[MethodName]]** internal property.
5. Else, Set *envRec*'s *home* to **Empty**.
6. Set *env*'s environment record to be *envRec*.
7. Set the outer lexical environment reference of *env* to the value of *F*'s **[[Scope]]** internal property.
8. Return *env*.

### 10.3 Code Realms

Before it is evaluated, all ECMAScript code must be associated with a *Realm*. Conceptually, a realm consists as of an set of intrinsic objects, an ECMAScript global environment, all of the ECMAScript code that is loaded

within the scope of that global environment, a Loader object that can associate new ECMAScript code with the realm, and other associated state.

A Realm is specified as a Record with the following fields:

**Table xxx — Realm Record Fields**

Field Name	Value	Meaning
<code>[[intrinsics]]</code>	A record whose field names are intrinsic keys and whose values are objects	These are the intrinsic values used by code associated with this Realm
<code>[[this]]</code>	An ECMAScript object	The global object for this Realm
<code>[[globalEnv]]</code>	A ECMAScript environment	The global environment for this Realm
<code>[[loader]]</code>	any ECMAScript identifier or empty	The Loader object that can associate ECMAScript code with this Realm

## 10.4 Execution Contexts

When control is transferred to ECMAScript executable code, control is entering an *execution context*. Active execution contexts logically form a stack. The top execution context on this logical stack is the running execution context. A new execution context is created whenever control is transferred from the executable code associated with the currently running execution context to executable code that is not associated with that execution context. The newly created execution context is pushed onto the stack and becomes the running execution context.

An execution context contains whatever *implementation-specific* state is necessary to track the execution progress of its associated code. In addition, each execution context has the state components listed in Table 20.

**Table 20 —Execution Context State Components**

Component	Purpose
<code>code evaluation state</code>	Any state needed to suspend and resume evaluation of the ECMAScript code associated with this execution context.
<code>PreviousContext</code>	The execution context that was active when this execution context was created. The value <code>Empty</code> indicates that there is no such context.
<code>Realm</code>	The Realm within which the associated ECMAScript code is evaluated
<code>LexicalEnvironment</code>	Identifies the Lexical Environment used to resolve identifier references made by code within this execution context.
<code>VariableEnvironment</code>	Identifies the Lexical Environment whose environment record holds bindings created by <i>VariableStatements</i> and <i>FunctionDeclarations</i> within this execution context.

The code evaluation state corresponds the implementation state needed to perform, suspend, and subsequently resume evaluation of the ECMAScript code associated with the execution context. Only one execution context may be actively evaluating ECMAScript code at any point in time. The execution context is active is called the current execution context.

Evaluation of code by the current execution context may be suspended at various points defined within this specification. Once the current execution context has been suspended a different execution context may become the current execution context and commence evaluating its ECMAScript code. At some latter time a suspended execution context may again become the current execution context and continue evaluating its ECMAScript code at the point where it had previously been suspended. Transition of of the current execution context status among execution contexts usually occurs in last-in/first-out manner. However, some ECMAScript features require non-LIFO changes to the current execution context.

**Deleted: 10.2.3 The Global Environment**  
 The *global environment* is a unique Lexical Environment which is created before any ECMAScript code is executed. The global environment's Environment Record is an object environment record whose binding object is the global object (15.1). The global environment's outer environment reference is `null`.  
 As ECMAScript code is executed, additional properties may be added to the global object and the initial properties may be modified.

**Deleted: 3**

**Deleted: 19**

**Deleted: 19**

**Formatted Table**

**Formatted Table**

**Deleted: ThisBinding**

**Formatted: English (U.K.)**

The LexicalEnvironment and VariableEnvironment components of an execution context are always Lexical Environments. When an execution context is created its LexicalEnvironment and VariableEnvironment components initially have the same value. The value of the VariableEnvironment component never changes while the value of the LexicalEnvironment component may change during execution of code within an execution context.

In most situations only the running execution context (the top of the execution context stack) is directly manipulated by algorithms within this specification. Hence when the terms “LexicalEnvironment”, and “VariableEnvironment” are used without qualification they are in reference to those components of the running execution context.

An execution context is purely a specification mechanism and need not correspond to any particular artefact of an ECMAScript implementation. It is impossible for an ECMAScript program to access an execution context.

#### **10.4.1 Identifier Resolution**

Identifier resolution is the process of determining the binding of an *IdentifierName* using the LexicalEnvironment of the running execution context. During execution of ECMAScript code, Identifier Resolution is performed using the following algorithm:

1. Let *env* be the running execution context’s LexicalEnvironment.
2. If the syntactic production that is being evaluated is contained in a strict mode code, then let *strict* be **true**, else let *strict* be **false**.
3. Return the result of calling GetIdentifierReference abstract operation passing *env*, the String value containing the same sequence of characters as *IdentifierName*, and *strict* as arguments.

The result of evaluating an identifier is always a value of type Reference with its referenced name component equal to the *Identifier* String.

#### **10.4.3 GetThisEnvironment**

The abstract operation GetThisEnvironment find the lexical environment that currently supplies the binding of the keyword *this*. It starts search with the LexicalEnvironment of the current execution context. GetThisEnvironment performs the following steps:

1. Let *lex* be the current execution context’s LexicalEnvironment.
2. Repeat
  - a. Let *envRec* be *lex*’s environment record.
  - b. Let *exists* be the result of calling the HasThisBinding concrete method of *envRec*.
  - c. If *exists* is **true**, then return *envRec*.
  - d. Let *outer* be the value of *lex*’s outer environment reference.
  - e. Let *lex* be *outer*.

#### **10.4.3 This Resolution**

The abstract operation ThisResolution is the process of determining the binding of the keyword *this* using the LexicalEnvironment of the current execution context. ThisResolution performs the following steps:

1. Let *env* be the result of performing the GetThisEnvironment abstract operation.
2. Return the result of calling the GetThisBinding concrete method of *env*.

#### **10.5 Declaration Binding Instantiation**

##### **10.5.1 Top-Level Declaration Instantiation**

**NOTE** When an execution context is established for evaluating non-function code declarations are instantiated in the current VariableEnvironment. Each top-level level variable, constant, or function declared in the code is instantiated.

Top-level Declaration Instantiation for base code is performed as follows:

<b>Deleted:</b> and “ThisBinding”
<b>Deleted:</b> 3
<b>Deleted:</b> the syntactic production <i>PrimaryExpression : Identifier</i> is evaluated
<b>Deleted:</b> function
<b>Deleted:</b> <i>Identifier</i>
<b>Comment [AWB713]:</b> TODO: move this section into the eval function
<b>Deleted:</b> 10.4 Establishing an Execution Context¶ Evaluation of global code or code using the eval function (15.1.2.1) establishes and enters a new execution context. Every invocation of an ECMAScript code function (13.2.1) also establishes and enters a new execution context, even if a function is calling itself recursively. Every return exits an execution context. A thrown exception may also exit one or more execution contexts.¶ When control enters an execution context the execution context’s ThisBinding is set its VariableEnvironment and initial LexicalEnvironment are defined, and declaration binding instantiation (10.5) is performed. The exact manner in which the actions occur depend on the type of code being entered.¶ 10.4.1 . Entering Global Code¶ The following steps are performed when control enters the execution context for global code:¶ Initialise the execution context using the global code as described in 10.4.1.1.¶ Perform Declaration Binding Instantiation described in 10.5 using the global code.¶ 10.4.1.1 Initial Global Execution Context¶ The following steps are performed to initialise a global execution context for ECMAScript code C:¶ Set the VariableEnvironment to the Global Environment.¶ Set the LexicalEnvironment to the Global Environment.¶ Set the ThisBinding to the global object.¶ 10.4.2 . Entering Eval Code¶ The following steps are performed when control enters the execution context for eval code:¶ If there is no calling context or if the eval code is not being evaluated by a direct call (15.1.2.1) to the eval function then,¶
<b>Formatted:</b> Font: 9 pt
<b>Formatted:</b> Font: 9 pt
<b>Comment [AWB14]:</b> It may make sense to split eval code out into a separate subsection
<b>Deleted:</b> Every execution context has an associated VariableEnvironment. Variables and functions declared in ECMAScript code evaluated in an execution context are added
<b>Deleted:</b> Base Code
<b>Deleted:</b> that is base code
<b>Comment [AWB15]:</b> This algorithm needs be update to use newly defined static semantic functions.

1. Let `env` be the environment record component of the current execution context's VariableEnvironment.
2. If `code` is eval code, then let `configurableBindings` be **true** else let `configurableBindings` be **false**.
3. If `code` is strict code, then let `strict` be **true** else let `strict` be **false**.
4. For each `FunctionDeclaration f` in `code`, in source text order do
  - a. Let `fn` be the Identifier in `FunctionDeclaration f`.
  - b. Let `fo` be the result of performing Function Declaration for f.
  - c. Let `funcAlreadyDeclared` be the result of calling `env`'s HasBinding concrete method passing `fn` as the argument.
  - d. If `funcAlreadyDeclared` is **false**, then
    - i. Call env's CreateMutableVarBinding concrete method passing fn and configurableBindings as the arguments.
    - ii. Call env's InitializeBinding concrete method passing fn, and undefined as the arguments.
  - e. Else if `env` is the environment record component of the global environment then
    - i. Let `go` be the global object.
    - ii. Let `existingProp` be the result of calling the `[[GetOwnProperty]]` internal method of `go` with argument `fn`.
    - iii. If `existingProp` is undefined or `existingProp.[[Configurable]]` is true, then
      1. Call the `[[DefineOwnProperty]]` internal method of `go`, passing `fn`, Property Descriptor `{[[Value]]: undefined, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: configurableBindings }`, and **true** as arguments.
    - iv. Else if `IsAccessorDescriptor(existingProp)` or `existingProp` does not have attribute values `{[[Writable]]: true, [[Enumerable]]: true}`, then
      1. Throw a `TypeError` exception.
  - f. Call `env`'s `SetMutableBinding` concrete method passing `fn, fo`, and `strict` as the arguments.
5. For each `VariableDeclaration` and `VariableDeclarationNoIn d` in `code`, in source text order do
  - a. Let `dn` be the Identifier in `d`.
  - b. Let `varAlreadyDeclared` be the result of calling `env`'s `HasBinding` concrete method passing `dn` as the argument.
  - c. If `varAlreadyDeclared` is **false**, then
    - i. Call `env`'s `CreateMutableBinding` concrete method passing `dn` and `configurableBindings` as the arguments.
    - ii. Call env's InitializeBinding concrete method passing dn, and undefined as the arguments.
    - iii. Call env's SetMutableBinding concrete method passing dn, undefined, and strict as the arguments.
  - d. else if `env` is the environment record component of the global environment then
    - i. Let go be the global object.
    - ii. Let `existingProp` be the result of calling the `[[GetOwnProperty]]` internal method of `go` with argument `dn`.
    - iii. If `existingProp` is undefined, then
      1. Call the `[[DefineOwnProperty]]` internal method of `go`, passing `dn`, Property Descriptor `{[[Value]]: undefined, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: configurableBindings }`, and **true** as arguments.

## 10.5.2 Module Declaration Instantiation

### 10.5.3 Function Declaration Instantiation

**NOTE** When an execution context is established for evaluating function code a new Declarative Environment Record is created and bindings for each formal parameter, and each function level variable, constant, or function declared in the function are instantiated in the environment record. Formal parameters and functions are initialized as part of this process. All other bindings are initialized during execution of the function code.

Function Declaration Instantiation is performed as follows using arguments `func`, `argumentsList`, and `env`. `func` is the function object that for which the execution context is being established. `env` is the declarative environment record in which bindings are to be created.

1. Let `code` be the value of the `[[Code]]` internal property of `func`.
2. Let `strict` be the value of the `[[Strict]]` internal property of `func`.
3. Let `formals` be the value of the `[[FormalParameterList]]` internal property of `func`.
4. Let `parameterNames` be the `BoundNames` of `formals`.

**Deleted:** running

**Deleted:** mode

**Deleted:** <#>If `code` is function code, then<br/><#>Let `func` be the function whose `[[Call]]` internal method initiated execution of `code`. Let `names` be the value of `func`'s `[[FormalParameters]]` internal property. ¶<br/><#>Let `argCount` be the number of elements in `args`. ¶<br/><#>Let `n` be the number 0. ¶<br/><#>For each String `argName` in `names`, in list order do¶<br/><#>Let `n` be the current value of `n` plus 1. ¶<br/><#>If `n` is greater than `argCount`, let `v` be `undefined` otherwise let `v` be the value of the `n`th element of `args`. ¶<br/><#>Let `argAlreadyDeclared` be the result of calling `env`'s `HasBinding` concrete method passing `argName` as the argument. ¶<br/><#>If `argAlreadyDeclared` is **false**, then¶<br/><#>call `Call env's CreateMutableBinding` concrete method passing `argName` as the argument. ¶<br/><#>Call `env's InitializeBinding` concrete method passing `argName`, and `undefined` as the arguments. ¶<br/><#>Call `env's SetMutableBinding` concrete method passing `argName, v`, and `strict` as the arguments. ¶

**Deleted:** instantiating `FunctionDeclaration`

**Deleted:** as described in Clause 13

**Deleted:** call

**Deleted:** ing

**Deleted:**

**Deleted:** <#>Let `argumentsAlreadyDeclared` be the result of calling `env`'s `HasBinding` concrete method passing `"arguments"` as the argument. ¶<br/><#>If `code` is function code and `argumentsAlreadyDeclared` is **false**, then¶<br/><#>Let `argsObj` be the result of calling the abstract operation `CreateArgumentsObject` (10.6) passing `func, names, args, e` and `strict` as arguments. ¶<br/><#>If `strict` is **true**, then¶<br/><#>Call `env's CreateImmutableBinding` concrete method passing the String `"arguments"` as the argument. ¶<br/><#>Call `env's InitializeImmutableBinding` concrete method passing `"arguments"` and `argsObj` as arguments. ¶<br/><#>Else, ¶<br/><#>Call `env's CreateMutableBinding` concrete method passing the String `"arguments"` as the argument. ¶<br/><#>Call `env's SetMutableBinding` `InitializeBinding` concrete method passing `"arguments"`, `"` and `argsObj`, and `false` as arguments. ¶

**Comment [AWB 316]:** This calls is needed to force strict mode error detection of object environment records. If this doesn't go away after completing refactoring it probably will require a note to explain.

**Deleted:** ing

**Deleted:** f

**Comment [AWB17]:** The exact details of declaration instantiation for module code still need to be worked out.

**Deleted:** Extended Code Top-Level

**Formatted:** Font: 9 pt

**Formatted:** Font: 9 pt

**Formatted:** Font: 9 pt

**Deleted:** `FormalParameterList`

**Deleted:** `FormalParameterList`

5. For each String *argName* in *parameterNames*, in list order do
  - a. Let *alreadyDeclared* be the result of calling *env*'s *HasBinding* concrete method passing *argName* as the argument.
  - b. NOTE Duplicate parameter names can only occur in non-strict Normal functions.
  - c. If *alreadyDeclared* is false, then
    - i. Call *env*'s *CreateMutableBinding* concrete method passing *argName* as the argument.
  - d. If *strict* is false, then
    - i. Call *env*'s *InitializeBinding* concrete method passing *argName*, and *undefined* as the arguments.
6. Let *declarations* be the LexicalDeclarations of *code*.
7. If *strict* is true, then
  - a. Let *ao* be the result of *CreateStrictArgumentsObject* with argument *argumentsList*.
  - b. Let *formalStatus* be the result of performing *BindingInitialisation* for *formals* with *ao* and *env* as arguments.
8. Else,
  - a. Let *names* be *BoundNames* of *formals*.
  - b. NOTE Because *F* is a non strict function it is not extended code. Hence *formals* does not contain the names of any destructuring *BindingProperties*, rest parameters, or parameters with default value initialisers.
  - c. Let *ao* be the result of performing the abstract operation *CreateMappedArgumentsObject* with arguments *names*, *env*, and *argumentsList*.
  - d. Let *formalStatus* be the result of performing *BindingInitialisation* for *formals* with *ao* and *undefined* as arguments.
9. NOTE *BindingInitialisation* for *formals* is performed prior to instantiating any non-parameter declarations in order to ensure that any such local declarations are not visible to any parameter *Initialisation* code that may be evaluated.
10. ReturnIfAbrupt(*formalStatus*).
11. For each element *d* in *declarations* do
  - a. For each element *dn* of the *BoundNames* of *d* do
    - i. Let *alreadyDeclared* be the result of calling *env*'s *HasBinding* concrete method passing *dn* as the argument.
    - ii. If *alreadyDeclared* is false, then
      1. If *IsConstantDeclaration* of *d* is true, then
        - a. Call *env*'s *CreateImmutableBinding* concrete method passing *dn* as the argument.
      2. Else,
        - a. Call *env*'s *CreateMutableBinding* concrete method passing *dn* and false as the arguments.
12. Let *argumentsAlreadyDeclared* be the result of calling *env*'s *HasBinding* concrete method passing "arguments" as the argument.
13. NOTE If *argumentsAlreadyDeclared* is true then the value of *ao* is not directly observable to ECMAScript code and need not actually exist. In that case, its use in the above steps is strictly as a device for specifying formal parameter initialisation semantics.
14. If *argumentsAlreadyDeclared* is false, then
  - a. If *strict* is true, then
    - i. Call *env*'s *CreateImmutableBinding* concrete method passing the String "arguments" as the argument.
  - b. Else,
    - i. Call *env*'s *CreateMutableBinding* concrete method passing the String "arguments" as the argument.
  - c. Call *env*'s *InitializeBinding* concrete method passing "arguments" and *ao* as arguments.
15. Let *varNames* be the VarDeclaredNames of *code*.
16. For each String *varName* in *varNames*, in list order do
  - a. Let *alreadyDeclared* be the result of calling *env*'s *HasBinding* concrete method passing *varName* as the argument.
  - b. NOTE A VarDeclaredNames is only instantiated and initialised here if it is not also the name of a formal parameter or a *FunctionDeclarations*. Such duplicate declarations may only occur in non-extended code.
  - c. If *alreadyDeclared* is false, then
    - i. Call *env*'s *CreateMutableBinding* concrete method passing *varName* as the argument.

**Deleted:** code

**Comment [AWB18]:** In non-extended code this will only contain function declarations

**Deleted:** Perform

**Deleted:** Initialization

**Comment [AWB619]:** TODO This is no longer true and needs to be taken into account

**Deleted:** Perform

**Deleted:** Initialization

**Deleted:** initialization

**Deleted:** initialization

**Formatted:** Font: Italic

**Comment [AWB20]:** Only function declarations may be multiply declared

**Deleted:** a *ConstDeclaration*

**Deleted:** initialization

**Comment [AWB721]:** TODO: don't create arguments binding for arrow functions (and perhaps for concise methods)

**Deleted:** <#>Let *hasArgumentsObject* be the Boolean negation of *argumentsAlreadyDeclared*.

ii. Call *env*'s `InitializeBinding` concrete method passing *fn*, and `undefined` as the arguments.

17. Let *initializedFunctions* be an emptyList.

18. For each `FunctionDeclaration` *f* in *declarations*, in reverse list order do

- a. NOTE If there are multiple `FunctionDeclarations` for the same name, the last declaration is used.  
Multiple `FunctionDeclarations` for the same name is only valid in non-extended code.
- b. Let *fn* be the sole element of the `BoundNames` of *f*.
- c. If *fn* is not an element of *initializedFunctions*, then
  - i. Append *fn* to *initializedFunctions*.
  - ii. Let *fo* be the result of performing `Instantiate Function Declaration for f`.
  - iii. Call *env*'s `InitializeBinding` concrete method passing *fn*, and *fo* as the arguments.

19. Return `NormalCompletion(empty)`.

#### 10.5.4 Block Declaration Instantiation

NOTE When a `Block` or `CaseBlock` production is evaluated a new Declarative Environment Record is created and bindings for each block scoped variable, constant, or function declared in the block are instantiated in the environment record.

Block Declaration Instantiation is performed as follows using arguments *code* and *env*. *code* is the grammar production corresponding to the body of the block. *env* is the declarative environment record in which bindings are to be.

1. Let *declarations* be the `LexicalDeclarations` of *code*.
2. For each element *d* in *declarations* do
  - a. For each element *dn* of the `BoundNames` of *d* do
    - i. If `IsConstantDeclaration` of *d* is `true`, then
      1. Call *env*'s `CreateImmutableBinding` concrete method passing *dn* as the argument.
    - ii. Else,
      1. Call *env*'s `CreateMutableBinding` concrete method passing *dn* and `false` as the arguments.
3. For each `FunctionDeclaration` *f* in *declarations*, in list order do
  - a. Let *fn* be the sole element of the `BoundNames` of *f*.
  - b. Let *fo* be the result of performing `Instantiate Function Declaration for f`.
  - c. Call *env*'s `InitializeBinding` concrete method passing *fn*, and *fo* as the arguments.

#### 10.6 Arguments Object

When function *code* is evaluated, an arguments object is created unless (as specified in 10.5) the identifier *arguments* occurs as an Identifier in the function's `FormalParameterList` or occurs as the `BindingIdentifier` of a `Declaration` contained in the function code.

The abstract operation `CreateStrictArgumentsObject` called with argument list *args* performs the following steps:

1. Let *len* be the number of elements in *args*.
2. Let *obj* be the result of the abstraction operation `InstantiateArgumentsObject` with argument *len*.
3. Let *indx* = *len* - 1.
4. Repeat while *indx* >= 0.
  - a. Let *val* be the element of *args* at 0-origin list position *indx*.
  - b. Call the `[[DefineOwnProperty]]` internal method on *obj* passing `ToString(indx)`, the property descriptor `{[[Value]]: val, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, and `false` as arguments.
  - c. Let *indx* = *indx* - 1
5. Let *thrower* be the `[[ThrowTypeError]]` function Object (13.2.3).
6. Call the `[[DefineOwnProperty]]` internal method of *obj* with arguments "caller", `PropertyDescriptor {[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: false}`, and `false`.
7. Call the `[[DefineOwnProperty]]` internal method of *obj* with arguments "callee", `PropertyDescriptor {[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: false}`, and `false`.
8. Return *obj*

**Deleted:** instantiating

**Deleted:** `FunctionDeclaration`

**Deleted:** as described in Clause 13

**Formatted:** Font: Times New Roman, Not Bold

**Deleted:** `hasArgumentsObject`

**Comment [AWB722]:** Jan 19 meeting notes: Current tentative decision is to support let, const, and local functions in nonstrict ES5 in the same way as in strict ES6. Fallback to either specifying limited cases or doing the ES5 nonstrict status quo (i.e. syntax error + Clause 16) if experiments show this to not be viable. We won't resolve this discussion without running some experiments.

**Formatted:** Font: 9 pt

**Deleted:** contained in extended code

**Formatted:** Font: 9 pt

**Deleted:** `<#>Assert: parameterNames has no duplicate entries.`

**Deleted:** a `ConstDeclaration`

**Formatted:** Font: Bold

**Deleted:** `FunctionDeclaration`

**Deleted:** instantiating `FunctionDeclaration`

**Deleted:** as described in Clause 13

**Comment [AWB 323]:** Why shouldn't function declarations with in blocks be constant bindings? For consistency with function scope bindings? There are already other inconsistencies such as duplicate function declarations being disallowed.

**Deleted:** `<#>Call env's CreateMutableBinding concrete method passing fn and configurableBindings as the arguments.`

**Deleted:** `<#>For each LetDeclaration and ConstDeclaration d in code do`  
`<#>For each element dn of the BoundNames of d do`  
`<#>If d is a LetDeclaration, then`  
`<#>Call env's CreateMutableBinding concrete method passing dn and false as the arguments.`  
`<#>Else,`  
`<#>Call env's CreateImmutableBinding concrete method passing dn as the argument.`

**Deleted:** control enters an execution context for

**Comment [AWB 224]:** Additional modification to this text will probably be need to account for the new declaration statements.

**Deleted:** `VariableDeclaration` or

**Deleted:** `Function`

The abstract operation `CreateMappedArgumentsObject` called with List `names`, environment record `env`, and argument list `args` performs the following steps:

1. Let `len` be the number of elements in `args`.
2. Let `obj` be the result of the abstraction operation `InstantiateArgumentsObject` with argument `len`.
3. Let `map` be the result of creating a new ECMAScript object.
4. Set all the internal methods of `map` as specified in 8.12.
5. Let `mappedNames` be an empty List.
6. Let `indx = len - 1`.
7. Repeat while `indx >= 0`,
  - a. Let `val` be the element of `args` at 0-originated list position `indx`.
  - b. Call the `[[DefineOwnProperty]]` internal method on `obj` passing `ToString(indx)`, the property descriptor `{[[Value]]: val, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, and `false` as arguments.
  - c. If `indx` is less than the number of elements in `names`, then
    - i. Let `name` be the element of `names` at 0-originated list position `indx`.
    - ii. If `name` is not an element of `mappedNames`, then
      1. Add `name` as an element of the list `mappedNames`.
      2. Let `g` be the result of calling the `MakeArgGetter` abstract operation with arguments `name` and `env`.
      3. Let `p` be the result of calling the `MakeArgSetter` abstract operation with arguments `name` and `env`.
      4. Call the `[[DefineOwnProperty]]` internal method of `map` passing `ToString(indx)`, the Property Descriptor `{[[Set]]: p, [[Get]]: g, [[Configurable]]: true}`, and `false` as arguments.
  - d. Let `indx = indx - 1`
8. If `mappedNames` is not empty, then
  - a. Set the `[[ParameterMap]]` internal property of `obj` to `map`.
  - b. Set the `[[Get]]`, `[[GetOwnProperty]]`, `[[DefineOwnProperty]]`, and `[[Delete]]` internal methods of `obj` to the definitions provided below.
9. Call the `[[DefineOwnProperty]]` internal method on `obj` passing `"callee"`, the property descriptor `{[[Value]]: func, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true}`, and `false` as arguments.
10. Return `obj`

The abstract operation `InstantiateArgumentsObject` called with an argument `len` performs the following steps:

1. Let `obj` be the result of creating a new ECMAScript object.
2. Set all the internal methods of `obj` as specified in 8.12.
3. Add the `[[NativeBrand]]` internal property to `obj` with value `NativeArguments`.
4. Set the `[[Prototype]]` internal property of `obj` to the standard built-in Object prototype object (15.2.4).
5. Call the `[[DefineOwnProperty]]` internal method on `obj` passing `"length"`, the Property Descriptor `{[[Value]]: len, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true}`, and `false` as arguments.
6. Return `obj`

The abstract operation `MakeArgGetter` called with String `name` and environment record `env` creates a function object that when executed returns the value bound for `name` in `env`. It performs the following steps:

1. Let `bodyText` be the result of concatenating the Strings `"return "`, `name`, and `";"`.
2. Let `body` be the result of parsing `bodyText` using `FunctionBody` as the goal symbol.
3. Let `parameters` be a `FormalParameterList : [empty]` production.
4. Return the result of creating a function object as described in 13.2 using `parameters as FormalParameterList, body for FunctionBody, env as Scope, and true for Strict`.

The abstract operation `MakeArgSetter` called with String `name` and environment record `env` creates a function object that when executed sets the value bound for `name` in `env`. It performs the following steps:

1. Let `paramText` be the String `name` concatenated with the String `_arg`.
2. Let `parameters` be the result of parsing `paramText` using `FormalParameterList` as the goal symbol.
3. Let `bodyText` be the String `"<name> = <param>;"` with `<name>` replaced by the value of `name` and `<param>` replaced by the value of `paramText`.
4. Let `body` be the result of parsing `bodyText` using `FunctionBody` as the goal symbol.

<b>Deleted:</b> Set
<b>Deleted:</b> Class
<b>Deleted:</b> IsArguments
<b>Deleted:</b> y
<b>Deleted:</b> of
<b>Deleted:</b> to "Arguments"
<b>Deleted:</b> <#>Let <code>Object</code> be the standard built-in Object constructor (15.2.2).¶
<b>Deleted:</b> <#>Let <code>indx = len - 1</code> ¶
<#>Repeat while <code>indx &gt;= 0</code> , ¶
<#>Let <code>val</code> be the element of <code>args</code> at 0-originated list position <code>indx</code> .¶
<#>Call the <code>[[DefineOwnProperty]]</code> internal method on <code>obj</code> passing <code>ToString(indx)</code> , the property descriptor <code>{[[Value]]: val, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}</code> , and <code>false</code> as arguments.¶
<#>If <code>indx</code> is less than the number of elements in <code>names</code> , then¶
<#>Let <code>name</code> be the element of <code>names</code> at 0-originated list position <code>indx</code> .¶
<#>If <code>strict</code> is <code>false</code> and <code>name</code> is not an element of <code>mappedNames</code> , then¶
<#>Add <code>name</code> as an element of the list <code>mappedNames</code> .¶
<#>Let <code>g</code> be the result of calling the <code>MakeArgGetter</code> abstract operation with arguments <code>name</code> and <code>env</code> .¶
<#>Let <code>p</code> be the result of calling the <code>MakeArgSetter</code> abstract operation with arguments <code>name</code> and <code>env</code> .¶
<#>Call the <code>[[DefineOwnProperty]]</code> internal method of <code>map</code> passing <code>ToString(indx)</code> , the Property Descriptor <code>{[[Set]]: p, [[Get]]: g, [[Configurable]]: true}</code> , and <code>false</code> as arguments.¶
<#>Let <code>indx = indx - 1</code> ¶
<#>If <code>mappedNames</code> is not empty, then¶
<#>Set the <code>[[ParameterMap]]</code> internal property of <code>obj</code> to <code>map</code> .¶
<#>Set the <code>[[Get]]</code> , <code>[[GetOwnProperty]]</code> , <code>[[DefineOwnProperty]]</code> , and <code>[[Delete]]</code> internal methods of <code>obj</code> to the definitions provided below.¶
<#>If <code>strict</code> is <code>false</code> , then¶
<#>Call the <code>[[DefineOwnProperty]]</code> internal method of <code>obj</code> passing <code>"callee"</code> , the property descriptor <code>{[[Value]]: func, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true}</code> , and <code>false</code> as arguments.¶
<#>Else, <code>strict</code> is <code>true</code> so¶
<#>Let <code>thrower</code> be the <code>[[ThrowTypeError]]</code> function Object (13.2.3).¶
<#>Call the <code>[[DefineOwnProperty]]</code> internal method of <code>obj</code> with arguments <code>"caller"</code> , PropertyDescriptor <code>{[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: false}</code> , and <code>false</code> .¶
<#>Call the <code>[[DefineOwnProperty]]</code> internal method of <code>obj</code> with arguments <code>"callee"</code> , PropertyDescriptor <code>{[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: false}</code> , and <code>false</code> .¶
<b>Deleted:</b> no <code>FormalParameterList</code>

5. Return the result of creating a function object as described in 13.2 using *parameters*, *FormalParameterList*, *body* for *FunctionBody*, *env* as *Scope*, and **true** for *Strict*.

**Deleted:** a List containing the single String

**Deleted:** *param*

The **[[Get]]** internal method of an arguments object for a non-strict mode function with formal parameters when called with a property name *P* performs the following steps:

1. Let *map* be the value of the **[[ParameterMap]]** internal property of the arguments object.
2. Let *isMapped* be the result of calling the **[[GetOwnProperty]]** internal method of *map* passing *P* as the argument.
3. If the value of *isMapped* is **undefined**, then
  - a. Let *v* be the result of calling the default **[[Get]]** internal method (8.12.3) on the arguments object passing *P* as the argument.
  - b. If *P* is **"caller"** and *v* is a strict mode Function object, throw a **TypeError** exception.
  - c. Return *v*.
4. Else, *map* contains a formal parameter mapping for *P* so,
  - a. Return the result of calling the **[[Get]]** internal method of *map* passing *P* as the argument.

The **[[GetOwnProperty]]** internal method of an arguments object for a non-strict mode function with formal parameters when called with a property name *P* performs the following steps:

1. Let *desc* be the result of calling the default **[[GetOwnProperty]]** internal method (8.12.1) on the arguments object passing *P* as the argument.
2. If *desc* is **undefined** then return *desc*.
3. Let *map* be the value of the **[[ParameterMap]]** internal property of the arguments object.
4. Let *isMapped* be the result of calling the **[[GetOwnProperty]]** internal method of *map* passing *P* as the argument.
5. If the value of *isMapped* is not **undefined**, then
  - a. Set *desc.[[Value]]* to the result of calling the **[[Get]]** internal method of *map* passing *P* as the argument.
6. Return *desc*.

The **[[DefineOwnProperty]]** internal method of an arguments object for a non-strict mode function with formal parameters when called with a property name *P*, Property Descriptor *Desc*, and Boolean flag *Throw* performs the following steps:

1. Let *map* be the value of the **[[ParameterMap]]** internal property of the arguments object.
2. Let *isMapped* be the result of calling the **[[GetOwnProperty]]** internal method of *map* passing *P* as the argument.
3. Let *allowed* be the result of calling the default **[[DefineOwnProperty]]** internal method (8.12.9) on the arguments object passing *P*, *Desc*, and **false** as the arguments.
4. If *allowed* is **false**, then
  - a. If *Throw* is **true** then throw a **TypeError** exception, otherwise return **false**.
5. If the value of *isMapped* is not **undefined**, then
  - a. If *IsAccessorDescriptor(Desc)* is **true**, then
    - i. Call the **[[Delete]]** internal method of *map* passing *P*, and **false** as the arguments.
  - b. Else
    - i. If *Desc.[[Value]]* is present, then
      1. Call the **[[Put]]** internal method of *map* passing *P*, *Desc.[[Value]]*, and *Throw* as the arguments.
    - ii. If *Desc.[[Writable]]* is present and its value is **false**, then
      1. Call the **[[Delete]]** internal method of *map* passing *P* and **false** as arguments.
6. Return **true**.

The **[[Delete]]** internal method of an arguments object for a non-strict mode function with formal parameters when called with a property name *P* and Boolean flag *Throw* performs the following steps:

1. Let *map* be the value of the **[[ParameterMap]]** internal property of the arguments object.
2. Let *isMapped* be the result of calling the **[[GetOwnProperty]]** internal method of *map* passing *P* as the argument.
3. Let *result* be the result of calling the default **[[Delete]]** internal method (8.12.7) on the arguments object passing *P* and *Throw* as the arguments.

4. If *result* is **true** and the value of *isMapped* is not **undefined**, then
  - a. Call the **[[Delete]]** internal method of *map* passing *P*, and **false** as the arguments.
5. Return *result*.

**NOTE 1** For non-strict mode functions the array index (defined in 15.4) named data properties of an arguments object whose numeric name values are less than the number of formal parameters of the corresponding function object initially share their values with the corresponding argument bindings in the function's execution context. This means that changing the property changes the corresponding value of the argument binding and vice-versa. This correspondence is broken if such a property is deleted and then redefined or if the property is changed into an accessor property. For strict mode functions, the values of the arguments object's properties are simply a copy of the arguments passed to the function and there is no dynamic linkage between the property values and the formal parameter values.

**NOTE 2** The ParameterMap object and its property values are used as a device for specifying the arguments object correspondence to argument bindings. The ParameterMap object and the objects that are the values of its properties are not directly accessible from ECMAScript code. An ECMAScript implementation does not need to actually create or use such objects to implement the specified semantics.

**NOTE 3** Arguments objects for strict mode functions define non-configurable accessor properties named "**caller**" and "**callee**" which throw a **TypeError** exception on access. The "**callee**" property has a more specific meaning for non-strict mode functions and a "**caller**" property has historically been provided as an implementation-defined extension by some ECMAScript implementations. The strict mode definition of these properties exists to ensure that neither of them is defined in any other manner by conforming ECMAScript implementations.

## 11 Expressions

### 11.1 Primary Expressions

#### Syntax

*PrimaryExpression* :

this  
Identifier  
Literal  
SealedArrayLiteral  
SealedObjectLiteral  
FunctionExpression  
GeneratorExpression  
GeneratorComprehension  
( Expression )

**Deleted:** This

#### 11.1.1 The **this** Keyword

##### Runtime Semantics: Evaluation

*PrimaryExpression* : this

1. Let *env* be the result of performing the **GetThisEnvironment** abstract operation.
2. Return the result of calling the **GetThisBinding** concrete method of *env*.

**Deleted:** <#>The **this** keyword evaluates to *Ret* the value of the **ThisBinding** of the current execution context.¶

#### 11.1.2 Identifier Reference

1. Let *ref* be the result of performing Identifier Resolution as specified in 10.3.1 using the *IdentifierName* corresponding to *Identifier*.
2. Return *ref*.

**Deleted:** An *Identifier* is evaluated by performing Identifier Resolution as specified in 10.3.1 using the *IdentifierName* corresponding to *Identifier*.

**Formatted:** Font: 9 pt

**NOTE:** The result of evaluating an *Identifier* is always a value of type **Reference**.

#### 11.1.3 Literal Reference

A *Literal* is evaluated as described in 7.8.

## 11.1.4 Array Initialiser

### Syntax

SealedArrayInitialiser :

ArrayInitialiser  
# ArrayInitialiser

ArrayInitialiser :

ArrayLiteral  
ArrayComprehension

#### 11.1.4.1 Array Literal

**NOTE** An ArrayLiteral is an expression describing the initialisation of an Array object, using a list of zero or more expressions, each of which represents an array element, enclosed in square brackets. The elements need not be literals; they are evaluated each time the array initialiser is evaluated.

Array elements may be elided at the beginning, middle or end of the element list. Whenever a comma in the element list is not preceded by an AssignmentExpression (i.e., a comma at the beginning or after another comma), the missing array element contributes to the length of the Array and increases the index of subsequent elements. Elided array elements are not defined. If an element is elided at the end of an array, that element does not contribute to the length of the Array.

### Syntax

ArrayLiteral :

[ Elision<sub>opt</sub> ]  
[ ElementList ]  
[ ElementList , Elision<sub>opt</sub> ]

ElementList :

Elision<sub>opt</sub> AssignmentExpression  
Elision<sub>opt</sub> ... AssignmentExpression  
ElementList , Elision<sub>opt</sub> AssignmentExpression  
ElementList , Elision<sub>opt</sub> SpreadElement

Elision :

'  
Elision ,

SpreadElement :

... AssignmentExpression

### Static Semantics

#### Static Semantics: Elision Width

Elision : [empty]

1. Return the numeric value 0.

Elision : ,

2. Return the numeric value 1.

Elision : Elision ,

3. Let preceding be the Elision Width of Elision.

4. Return preceding+1.

**Formatted:** Heading 4

**Deleted:** array initialiser

**Deleted:** written in a form of a literal. It is

**Deleted:** ,

**Deleted:** ... AssignmentExpression

**Deleted:** ArrayComprehension :  
Expression ComprehensionForList  
Expression ComprehensionForList if ( Expression )  
ComprehensionForList :  
ComprehensionFor ..  
ComprehensionForList ComprehensionFor :  
ComprehensionFor :  
for ( LeftHandSideExpression of Expression )

**Deleted:** 11.1.4.1

**Deleted:** Static Semantics: Early Errors

Early Errors  
The static semantics of the production  
SpreadElementElementList : Elision<sub>opt</sub> ...  
AssignmentExpression are:  
It is a Syntax Error if the source code parsed with this production is not extended code.

**Deleted:** ElementList : ElementList , Elision<sub>opt</sub> ...  
AssignmentExpression  
It is a Syntax Error if the source code parsed with this production is not extended code.

## Runtime Semantics

**Deleted: 11.1.4.2**

### Runtime Semantics: Array Accumulation

With parameters *array* and *nextIndex*.

ElementList : Elision<sub>opt</sub> AssignmentExpression

1. Let *padding* be the Elision Width of *Elision<sub>opt</sub>*.
2. Let *initResult* be the result of evaluating *AssignmentExpression*.
3. Let *initValue* be *GetValue(initResult)*.
4. If *initValue* is an abrupt completion, return *initValue*.
5. Call the **[[DefineOwnProperty]]** internal method of *array* with arguments  
*ToUint32(nextIndex+padding)*, the Property Descriptor { **[[Value]]: *initValue*, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true** }, and false.
6. Return *nextIndex+padding+1*.

ElementList : Elision<sub>opt</sub> SpreadElement

1. Let *padding* be the Elision Width of *Elision<sub>opt</sub>*.
2. Return the result of performing Array Accumulation for *SpreadElement* with arguments *array* and *nextIndex+padding*.

ElementList : ElementList , Elision<sub>opt</sub> AssignmentExpression

1. Let *postIndex* be the result of performing Array Accumulation for *ElementList* with arguments *array* and *nextIndex*.
2. If *postIndex* is an abrupt completion, return *postIndex*.
3. Let *padding* be the Elision Width of *Elision<sub>opt</sub>*.
4. Let *initResult* be the result of evaluating *AssignmentExpression*.
5. Let *initValue* be *GetValue(initResult)*.
6. If *initValue* is an abrupt completion, return *initValue*.
7. Call the **[[DefineOwnProperty]]** internal method of *array* with arguments  
*ToUint32((postIndex+padding))* and the Property Descriptor { **[[Value]]: *initValue*, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true** }, and false.
8. Return *postIndex+padding+1*.

ElementList : ElementList , Elision<sub>opt</sub> SpreadElement

1. Let *postIndex* be the result of performing Array Accumulation for *ElementList* with arguments *array* and *nextIndex*.
2. If *postIndex* is an abrupt completion, return *postIndex*.
3. Let *pad* be the result of evaluating *Elision<sub>opt</sub>*; if not present, use the numeric value zero.
4. Return the result of performing Array Accumulation for *SpreadElement* with arguments *array* and *postIndex+padding*.

SpreadElement : ... AssignmentExpression

1. Let *spreadRef* be the result of evaluating *AssignmentExpression*.
2. Let *spreadValue* be *GetValue(spreadRef)*.
3. Let *spreadObj* be *ToObject(spreadValue)*.
4. If *spreadObj* is an abrupt completion, return *spreadObj*.
5. Let *lenVal* be the result of calling the **[[Get]]** internal method of *spreadObj* with argument "**length**".
6. Let *spreadLen* be *ToUint32(lenVal)*.
7. **ReturnIfAbrupt(spreadLen)**.
8. Let *n*=0;
9. Repeat, while *n* < *spreadLen*
  - a. Let *exists* be the result of calling the **[[HasProperty]]** internal method of *spreadObj* with *ToString(n)*.
  - b. If *exists* is true then,

**Comment [AW25]:** Note that the value the spread operator is applied to is coerced to an Object.

- i. Let  $v$  be the result of calling the `[[Get]]` internal method of `spreadObj` passing `ToString(n)` as the argument.
  - ii. **ReturnIfAbrupt(v).**
  - iii. Call the `[[DefineOwnProperty]]` internal method of `array` with arguments  
`ToString(ToUint32(nextIndex))`, Property Descriptor `{[[Value]]: v, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, and `false`.
  - c. Let  $n = n+1$ .
  - d. Let  $nextIndex = nextIndex + 1$ .
- 10. Return `nextIndex`.**

**NOTE** `[[DefineOwnProperty]]` is used to ensure that own properties are defined for the array even if the standard built-in `Array` prototype object has been modified in a manner that would preclude the creation of new own properties using `[[Put]]`.

### Runtime Semantics: Evaluation

**ArrayLiteral : [ Elision<sub>opt</sub> ]**

1. Let `array` be the result of the abstraction operation `ArrayCreate (15.4)` with argument `0`.
2. Let `pad` be the Elision Width of `Elisionopt`.
3. Call the `[[Put]]` internal method of `array` with arguments `"length"`, `pad`, and `false`.
4. Return `array`.

**ArrayLiteral : [ ElementList ]**

1. Let `array` be the result of the abstraction operation `ArrayCreate (15.4)` with argument `0`.
2. Let `len` be result of performing `Array Accumulation` for `ElementList` with arguments `array` and `0`.
3. If `len` is an abrupt completion, return `len`.
4. Call the `[[Put]]` internal method of `array` with arguments `"length"`, `len`, and `false`.
5. Return `array`.

**ArrayLiteral : [ ElementList , Elision<sub>opt</sub> ]**

1. Let `array` be the result of the abstraction operation `ArrayCreate (15.4)` with argument `0`.
2. Let `len` be result of performing `Array Accumulation` for `ElementList` with arguments `array` and `0`.
3. If `len` is an abrupt completion, return `len`.
4. Let `pad` be the Elision Width of `Elisionopt`.
5. Call the `[[Put]]` internal method of `array` with arguments `"length"`, `ToUint32(pad+len)`, and `false`.
6. Return `array`.

### 11.1.4.2 Array Comprehension

#### Syntax

**ArrayComprehension :**  
`[ Expression ComprehensionForList ]`  
`[ Expression ComprehensionForList if Expression ]`

**ComprehensionForList :**  
`ComprehensionFor`  
`ComprehensionForList ComprehensionFor`

**ComprehensionFor :**  
`for ForBinding of Expression`

**ForBinding :**  
`BindingIdentifier`  
`BindingPattern`

#### Static Semantics

**Comment [AW26]:** Note that indices wrap. For example consider:

`[,,,{ 4294967293: "x", length: Math.pow(2,32)-2}]`

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** the result of creating a new object as if by the expression `new Array()` where `Array` is the standard built-in constructor with that name

**Deleted:** result of evaluating

**Deleted:** ; if not present, use the numeric value zero

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** the result of creating a new object as if by the expression `new Array()` where `Array` is the standard built-in constructor with that name

**Deleted:** Return the

**Formatted:** Font: Italic

**Deleted:** evaluating

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** the result of creating a new object as if by the expression `new Array()` where `Array` is the standard built-in constructor with that name

**Deleted:** <#>Let `array` be the result of evaluating `ElementList.`<

**Deleted:** result of evaluating

**Deleted:** ; if not present, use the numeric value zero

**Deleted:** <#>Let `len` be the result of calling the `[[Get]]` internal method of `array` with argument `"length".`<

**Deleted:** (

**Deleted:** )

**Deleted:** (

**Deleted:** LeftHandSideExpression

**Deleted:** )

## Static Semantics: Bound Names

### ForBinding : BindingIdentifier

1. Return the Bound Names of *BindingIdentifier*.

### ForBinding : BindingPattern

1. Return the Bound Names of *BindingPattern*.

## Runtime Semantics

### Runtime Semantics: Binding Initialisation

With arguments *value* and *environment*.

**NOTE** *undefined* is passed for *environment* to indicate that a *PutValue* operation should be used to assign the initialisation value. This is the case for *var* statements formal parameter lists of non-strict functions. In those cases a lexical binding is hosted and preinitialized prior to evaluation of its initializer.

### ForBinding : BindingIdentifier

1. Perform Binding Initialisation for *BindingIdentifier* passing *value* and *env* as the arguments.

### ForBinding: BindingPattern

1. Perform Binding Initialisation for *BindingPattern* passing *value* and *env* as the arguments.

## Runtime Semantics: Evaluation

### ArrayLiteral : [ ElementList ]

1. Let *array* be the result of the abstraction operation *ArrayCreate* (15.4) with argument 0.
2. Let *len* be result of performing Array Accumulation for *ElementList* with arguments *array* and 0.
3. If *len* is an abrupt completion, return *len*.
4. Call the *[[Put]]* internal method of *array* with arguments "length", *len*, and false.
5. Return *array*.

### 11.1.5 Object Initialiser

**NOTE** An object initialiser is an expression describing the initialisation of an Object, written in a form resembling a literal. It is a list of zero or more pairs of property names and associated values, enclosed in curly braces. The values need not be literals; they are evaluated each time the object initialiser is evaluated.

## Syntax

### SealedObjectLiteral :

```

ObjectLiteral
# ObjectLiteral

```

### ObjectLiteral :

```

{ }
{ PropertyDefinitionList }
{ PropertyDefinitionList , }

```

### PropertyDefinitionList :

```

PropertyDefinition
PropertyDefinitionList , PropertyDefinition

```

**Deleted:** The production *ElementList : Elision*  
*AssignmentExpression* is evaluated as follows:  
<#> Let *array* be the result of creating a new object as if by the expression **new Array()** where **Array** is the standard built-in constructor with that name.  
<#> Let *firstIndex* be the result of evaluating *Elision*; if not present, use the numeric value zero.  
<#> Let *initResult* be the result of evaluating *AssignmentExpression*.  
<#> Let *initValue* be *GetValue(initResult)*.  
<#> Call the *[[DefineOwnProperty]]* internal method of *array* with arguments *Tostring(firstIndex)*, the Property Descriptor { *[[Value]]*: *initValue*, *[[Writable]]*: true, *[[Enumerable]]*: true, *[[Configurable]]*: true }, and false.  
<#> Return *array*.  
The production *ElementList : Elision<sub>opt</sub>* ...  
*AssignmentExpression* is evaluated as follows:  
Let *array* be the result of creating a new object as if by the expression **new Array()** where **Array** is the standard built-in constructor with that name.  
Let *index* be the result of evaluating *Elision*; if not present, use the numeric value zero.  
<#> Let *spreadRef* be the result of evaluating *AssignmentExpression*.  
<#> Let *spreadValue* be *GetValue(spreadRef)*.  
<#> Let *spreadObj* be *ToObject(spreadValue)*.  
<#> Let *lenVal* be the result of calling the *[[Get]]* internal method of *spreadObj* with argument "length".  
<#> Let *spreadLen* be *ToUint32(lenVal)*.  
<#> Let *n*=0.  
<#> Repeat, while *n* < *spreadLen*.  
<#> Let *exists* be the result of calling the *[[HasProperty]]* internal method of *spreadObj* with *Tostring(n)*.  
<#> If *exists* is true then,  
<#> Let *v* be the result of calling the *[[Get]]* internal method of *spreadObj* passing *Tostring(n)* as the argument.  
<#> Call the *[[DefineOwnProperty]]* internal method of *array* with arguments *Tostring(ToUint32(index))*, the Property Descriptor { *[[Value]]*: *v*, *[[Writable]]*: true, *[[Enumerable]]*: true, *[[Configurable]]*: true }, and false.  
<#> Let *n*=*n*+1.  
<#> Let *index*=*index*+1.  
<#> Return *array*.  
The production *ElementList : ElementList*,  
*Elision<sub>opt</sub>* *AssignmentExpression* is evaluated as follows:  
<#> Let *array* be the result of evaluating *ElementList*.  
<#> Let *pad* be the result of evaluating *Elision*; if present, use the numeric value zero.  
<#> Let *initResult* be the result of evaluating *AssignmentExpression*.  
<#> Let *initValue* be *GetValue(initResult)*.  
<#> Let *len* be the result of calling the *[[Get]]* internal method of *array* with argument "length".  
<#> Call the *[[DefineOwnProperty]]* internal method of *array* with arguments

PropertyDefinition :  
IdentifierName  
 PropertyName : AssignmentExpression  
MethodDefinition

PropertyName :  
IdentifierName  
 StringLiteral  
 NumericLiteral

NOTE MethodDefinition is defined in 13.3.

#### 11.1.5.1 Static Semantics

##### Early Errors

ObjectLiteral : { PropertyDefinitionList }  
 and  
ObjectLiteral : { PropertyDefinitionList , }

- It is a Syntax Error if PropertyNameList of PropertyDefinitionList contains any duplicate entries, unless one of the following conditions are true for each duplicate entry:
  1. The source code corresponding to PropertyDefinitionList is not strict code and all occurrences in the list of the duplicated entry were obtained from productions of the form PropertyDefinition : PropertyName : AssignmentExpression.
  2. The duplicated entry occurs exactly twice in the list and one occurrence was obtained from a get accessor property definition and the other occurrence was obtained from a set accessor property definition.

##### Static Semantics: Contains

With parameter symbol.

PropertyDefinition : MethodDefinition

NOTE Static semantic rules that depend upon substructure generally do not look into function definitions.

##### Static Semantics: PropName

PropertyDefinition : IdentifierName

1. Return String Value of IdentifierName.

PropertyDefinition : PropertyName : AssignmentExpression

1. Return PropName of PropertyName.

PropertyDefinition : MethodDefinition

1. Return PropName of MethodDefinition.

PropertyName : IdentifierName

1. Return PropName of IdentifierName.

PropertyName : StringLiteral

1. Return the SV of the StringLiteral.

**Deleted:** PropertyAssignment

**Deleted:**

PropertyName ( FormalParameterList ) { FunctionBody }  
 \* PropertyName ( FormalParameterList ) { FunctionBody } ..  
 get PropertyName ( ) { FunctionBody } ..  
 set PropertyName ( PropertySetParameterList ) {  
 FunctionBody }

**Formatted:** SyntaxRule

**Deleted:** PropertySetParameterList ¶  
 BindingIdentifier  
 BindingPattern ¶

**Deleted:** Identifier ¶  
 Semantics ¶

**Deleted:** PropertyNameAndValueList :  
 PropertyNameAndValueList , PropertyAssignment ¶

**Deleted:** <#>It is a Syntax Error if this production is contained in strict code, PropertyAssignment is the production PropertyAssignment : PropertyName : AssignmentExpression, and PropertyDefinitionList(PropName of PropertyAssignment) of PropertyNameAndValueList is not the empty List.¶

<#>It is a Syntax Error if PropertyAssignment is the production PropertyAssignment : IdentifierName and PropertyDefinitionList(PropName of PropertyAssignment) PropertyNameAndValueList is not the empty List.¶  
 <#>It is a Syntax Error if PropertyAssignment is the production PropertyAssignment : PropertyName ( FormalParameterList ) { FunctionBody } and PropertyDefinitionList(PropName of PropertyAssignment) PropertyNameAndValueList is not the empty List.¶  
 <#>It is a Syntax Error if PropertyAssignment is the production PropertyAssignment : \* PropertyName ( FormalParameterList ) { FunctionBody } and PropertyDefinitionList(PropName of PropertyAssignment) PropertyNameAndValueList is not the empty List.¶  
 <#>It is a Syntax Error if PropertyAssignment is the production PropertyAssignment : get PropertyName ( ) { FunctionBody } and PropertyDefinitionList(PropName of PropertyAssignment) PropertyNameAndValueList is not the empty List.¶  
 and PropertyDefinitionList(PropName of PropertyAssignment) of PropertyNameAndValueList includes a production of the form PropertyAssignment : PropertyName : AssignmentExpression.¶  
 <#>It is a Syntax Error if PropertyAssignment is the production .  
PropertyAssignment : set PropertyName ( PropertySetParameterList ) { FunctionBody } and PropertyDefinitionList(PropName of PropertyAssignment) of PropertyNameAndValueList includes a production of the form PropertyAssignment : PropertyName : AssignmentExpression.¶  
 <#>It is a Syntax Error if PropertyAssignment is the production .

**Formatted:** Font: (Default) Arial, Not Bold

**Deleted:** Static Semantics: ExpectedArgumentCount

PropertySetParameterList : BindingIdentifier ¶  
 <#>Return 1.¶  
 PropertySetParameterList : BindingPattern ¶

**Deleted:** PropertyAssignment

**Deleted:** PropertyAssignment

**Deleted:** PropertyAssignment : PropertyName ( FormalParameterList ) { FunctionBody } ¶  
 <#>Return PropName of PropertyName.¶

**Deleted:** (

**Deleted:** )

*PropertyName : NumericLiteral*

1. Let *nbr* be the result of forming the value of the *NumericLiteral*.
2. Return *ToString(nbr)*.

#### Static Semantics: *PropertyNameList*

*PropertyDefinitionList : PropertyDefinition*

1. Return a new List containing *PropName* of *PropertyDefinition*.

*PropertyDefinitionList : PropertyDefinitionList , PropertyDefinition*

1. Let *list* be *PropertyNameList* of *PropertyDefinitionList*.
1. Append *PropName* of *PropertyDefinition* to the end of *list*.
1. Return *list*.

#### 11.1.5.2 Runtime Semantics

##### Runtime Semantics: Evaluation

*ObjectLiteral : { }*

1. Return a new object created as if by the expression `new Object()` where **Object** is the standard built-in constructor with that name.

*ObjectLiteral :*

{ *PropertyDefinitionList* }  
 { *PropertyDefinitionList* , }.

1. Let *obj* be the result of the abstraction operation *ObjectCreate* (15.2).
2. Let *success* be the result of performing *Property Definition Evaluation* of *PropertyDefinitionList* with argument *obj*.
3. If *success* is an abrupt completion, return *success*.
4. Return *obj*.

##### Runtime Semantics: *Property Definition Evaluation*

With parameter *object*.

*PropertyDefinitionList : PropertyDefinition*

1. Return the result of Performing *Property Definition Evaluation* of *PropertyDefinition* with argument *object*.

*PropertyDefinitionList : PropertyDefinitionList , PropertyDefinition*

1. Let *success* be the result of performing *Property Definition Evaluation* of *PropertyDefinitionList* with argument *object*.
2. If *success* is an abrupt completion, return *success*.
3. Return the result of performing *Property Definition Evaluation* of *PropertyDefinition* with argument *object*.

*PropertyDefinition : IdentifierName*

1. Let *propName* be *PropName(IdentifierName)*.
2. Let *exprValue* be the result of performing *Identifier Resolution* as specified in 10.3.1 using *IdentifierName*.
3. Let *propValue* be *GetValue(exprValue)*.
4. If *propValue* is an abrupt completion, return *propValue*.
5. Let *desc* be the *Property Descriptor* {[[Value]]: *propValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** *PropertyNameAndValueList*

**Deleted:** *PropertyNameAndValueList*

**Deleted:** *ObjectLiteral* :  
 { *PropertyNameAndValueList* } and  
*ObjectLiteral* : { *PropertyNameAndValueList* , } a  
 evaluated as follows:

**Deleted:** creating a new object as if by the expression `new Object()` where **Object** is standard built-in constructor with that name

**Deleted:** P

**Formatted:** Font: Italic

**Deleted:** D

**Deleted:** *PropertyNameAndValueList*

**Deleted:** the result of evaluating  
*PropertyNameAndValueList*

**Deleted:** The production

**Deleted:** *PropertyNameAndValueList*

**Deleted:** *PropertyAssignment*

**Deleted:** is evaluated as follows:

**Deleted:** Let *obj* be the result of creating a new object as if by the expression `new Object()` where **Object** is the standard built-in constructor with that name.  
 ↪ Let *propld* be the result of evaluating *PropertyAssignment*.

**Deleted:** Dvaluation

**Deleted:** *PropertyAssignment*

**Deleted:** Return *obj*.

**Deleted:** The production .

**Deleted:** *PropertyNameAndValueList*

**Deleted:** *PropertyNameAndValueList*

**Deleted:** *PropertyAssignment*

**Deleted:**

**Deleted:** ↪ Let *obj* be the result of evaluating

**Deleted:** ↪ Let *previous* be the result of calling

**Deleted:** P

**Deleted:** Dvaluation

**Deleted:** *PropertyNameAndValueList*

**Deleted:** Call the [[DefineOwnProperty]] internal

**Deleted:** P

**Deleted:** Dvaluation

**Deleted:** *PropertyAssignment*

**Deleted:** Return *obj*.

**Deleted:** ↪

**Deleted:** The production

**Deleted:** *PropertyAssignment*

**Deleted:** is evaluated as follows:

**Deleted:** as the *Identifier*

6. Return the result of calling the `[[DefineOwnProperty]]` internal method of `object` with arguments `propName`, `desc`, and `false`.

**Deleted:** Call

**PropertyDefinition :** `PropertyName : AssignmentExpression`

1. Let `propName` be `PropName(PropertyName)`.
2. Let `exprValue` be the result of evaluating `AssignmentExpression`.
3. Let `propValue` be `GetValue(exprValue)`.
4. If `propValue` is an abrupt completion, return `propValue`.
5. Let `desc` be the Property Descriptor `{[[Value]]: propValue, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`.
6. Return the result of calling the `[[DefineOwnProperty]]` internal method of `object` with arguments `propName`, `desc`, and `false`.

**Deleted:** <#>Return Property Identifier (`propName, desc`)

**Deleted:** The production

**Deleted:** `PropertyAssignment`

**Deleted:** is evaluated as follows:

**Deleted:** the result of evaluating

**Deleted:** Call

**Deleted:** Return Property Identifier (`propName, desc`)

**Comment [AWB627]:** Need a actual reference

**PropertyDefinition :** `MethodDefinition`

1. Return the result of Property Definition Evaluation of `MethodDefinition` with argument `object`.

## 11.1.6 Function Expressions

### Runtime Semantics: Evaluation

**PrimaryExpression :** `FunctionExpression`

1. Return the result of evaluating `FunctionExpression`.

## 11.1.7 Generator Comprehensions

### Syntax

**GeneratorComprehension :**

`( Expression ComprehensionForList )`  
`( Expression ComprehensionForList if ( Expression ) )`

**ComprehensionForList :**

`ComprehensionFor`  
`ComprehensionForList ComprehensionFor`

**ComprehensionFor :**

`for LeftHandSideExpression of Expression`

## 11.1.8 The Grouping Operator

### Runtime Semantics: Evaluation

**PrimaryExpression :** `( Expression )`

1. Return the result of evaluating `Expression`. This may be of type Reference.

**NOTE** This algorithm does not apply `GetValue` to the result of evaluating `Expression`. The principal motivation for this is so that operators such as `delete` and `typeof` may be applied to parenthesised expressions.

**Deleted:** `PropertyAssignment : PropertyName ( FormalParameterList ) { FunctionBody }`  
 <#>Let `propName` be `PropName(PropertyName)`.  
 <#>Let `closure` be the result of creating a new Function object as specified in 13.1.21 using a `FormalParameterList` : empty production as the parameter list and body specified by `FunctionBody`. Pass in the LexicalEnvironment of the running execution context as the `Scope`. Pass in `true` as the `Strict` flag. Pass `object` as the optional `homeObject` argument and `propName` as the optional `methodName` argument.  
 <#>Let `desc` be the Property Descriptor `{[[Get]]: closure, [[Enumerable]]: true, [[Configurable]]: true}`.  
 <#>Return the result of calling `Call` the `[[DefineOwnProperty]]` internal method of `object` with arguments `propName, desc, a false`.

**PropertyAssignment :** \* `PropertyName ( FormalParameterList ) { FunctionBody }`  
 <#>Let `propName` be `PropName(PropertyName)`.  
 <#>Let `closure` be the result of creating a new Generator object as specified in 13.2.xxx using a `FormalParameterList` : [empty] production as the parameter list and body specified by `FunctionBody`. Pass in the LexicalEnvironment of the running execution context as the `Scope`. Pass in `true` as the `Strict` flag. Pass `object` as the optional `homeObject` argument and `propName` as the optional `methodName` argument.  
 <#>Let `desc` be the Property Descriptor `{[[Get]]: closure, [[Enumerable]]: true, [[Configurable]]: true}`.  
 <#>Return the result of calling the `[[DefineOwnProperty]]` internal method of `object` with arguments `propName, desc, a false`.

The production `PropertyAssignment : get PropertyName ( ) { FunctionBody }` is evaluated as follows:  
 <#>Let `propName` be `PropName(the result of evaluating PropertyName)`.  
 <#>Let `closure` be the result of creating a new Function object as specified in 13.2 with using a `FormalParameterList` : [empty] production an emptyas the formal parameter list and body specified by `FunctionBody`. Pass in the LexicalEnvironment of the running execution context as the `Scope`. Pass in `true` as the `Strict` flag if the `PropertyAssignment` is contained in strict code or if its `FunctionBody` is strict code. Pass `object` as the optional `homeObject` argument and `propName` as the optional `methodName` argument.  
 <#>Let `desc` be the Property Descriptor `{[[Get]]: closure, [[Enumerable]]: true, [[Configurable]]: true}`.

**Deleted:** (

**Deleted:** )

**Deleted:** The production `PropertyName : IdentifierName` is evaluated as follows:  
 <#>Let `closure` be the result of creating a new Function object as specified in 13.2 with using a `FormalParameterList` : [empty] production an emptyas the formal parameter list and body specified by `FunctionBody`. Pass in the LexicalEnvironment of the running execution context as the `Scope`. Pass in `true` as the `Strict` flag if the `PropertyAssignment` is contained in strict code or if its `FunctionBody` is strict code. Pass `object` as the optional `homeObject` argument and `propName` as the optional `methodName` argument.

**Deleted:** 6

**Deleted:** 7

**Deleted:** The production

**Deleted:** is evaluated as follows:

## 11.2 Left-Hand-Side Expressions

### Syntax

*MemberExpression :*

*PrimaryExpression*  
*MemberExpression [ Expression ]*  
*MemberExpression . IdentifierName*  
*MemberExpression <| TriangleLiteral*  
*super [ Expression ]*  
*super . IdentifierName*  
*new MemberExpression Arguments*

*NewExpression :*

*MemberExpression*  
*new NewExpression*

*CallExpression :*

*MemberExpression Arguments*  
*super Arguments*  
*CallExpression Arguments*  
*CallExpression [ Expression ]*  
*CallExpression . IdentifierName*  
*CallExpression <| TriangleLiteral*

*Arguments :*

*()*  
*( ArgumentList )*

*ArgumentList :*

*AssignmentExpression*  
*... AssignmentExpression*  
*ArgumentList , AssignmentExpression*  
*ArgumentList , ... AssignmentExpression*

*TriangleLiteral :*

*SealedArrayLiteral*  
*SealedObjectLiteral*  
*FunctionExpression*  
*ArrowFunction*  
*ValueLiteral*

*LeftHandSideExpression :*

*NewExpression*  
*CallExpression*

### Static Semantics

**Static Semantics: IsValidAssignmentPattern**

*LeftHandSideExpression : NewExpression*

1. If the source code corresponding to *NewExpression* cannot be parsed as *AssignmentPattern*, then
  - a. If the source code corresponding to *NewExpression* cannot be parsed as *ObjectLiteral*, then return **false**.
2. If the source code corresponding to *NewExpression* cannot be parsed using *AssignmentPattern* (11.13.1) as the goal symbol, return **true**.
3. Return **false**.

**Deleted:** *FunctionExpression*

**Deleted:** *MemberExpression <| SealedArrayLiteral .*

*MemberExpression <| SealedObjectLiteral*  
*MemberExpression <| FunctionExpression*  
*MemberExpression <| ValueLiteral*

### [LeftHandSideExpression : CallExpression](#)

1. Return false.

#### 11.2.1 Property Accessors

Properties are accessed by name, using either the dot notation:

*MemberExpression . IdentifierName*  
*CallExpression . IdentifierName*

or the bracket notation:

*MemberExpression [ Expression ]*  
*CallExpression [ Expression ]*

The dot notation is explained by the following syntactic conversion:

*MemberExpression . IdentifierName*

is identical in its behaviour to

*MemberExpression [ <identifier-name-string> ]*

and similarly

*CallExpression . IdentifierName*

is identical in its behaviour to

*CallExpression [ <identifier-name-string> ]*

where *<identifier-name-string>* is a string literal containing the same sequence of characters after processing of Unicode escape sequences as the *IdentifierName*.

### [Runtime Semantics: Evaluation](#)

#### [MemberExpression : MemberExpression \[ Expression \]](#)

1. Let *baseReference* be the result of evaluating *MemberExpression*.
2. Let *baseValue* be *GetValue(baseReference)*.
3. If *baseValue* is an abrupt completion, return *baseValue*.
4. Let *propertyNameReference* be the result of evaluating *Expression*.
5. Let *propertyNameValue* be *GetValue(propertyNameReference)*.
6. If *propertyNameValue* is an abrupt completion, return *propertyNameValue*.
7. ReturnIfAbrupt(CheckObjectCoercible(*baseValue*)).
8. Let *propertyNameString* be *ToString(propertyNameValue)*.
9. If the code matched by the syntactic production that is being evaluated is strict mode code, let *strict* be **true**, else let *strict* be **false**.
10. Return a value of type Reference whose base value is *baseValue* and whose referenced name is *propertyNameString*, and whose strict mode flag is *strict*.

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** Call

**Deleted:** If the result of

**Deleted:**

**Deleted:** is an abrupt completion, return that result

**Deleted:** contained in

**Deleted:** The production

**Deleted:** is

**Deleted:** ,

#### [CallExpression : CallExpression \[ Expression \]](#)

Is evaluated in exactly the same manner as [MemberExpression : MemberExpression \[ Expression \]](#), except that the contained *CallExpression* is evaluated in step 1.

#### 11.2.2 The new Operator

### [Runtime Semantics: Evaluation](#)

#### [NewExpression : new NewExpression](#)

1. Let *ref* be the result of evaluating *NewExpression*.

**Deleted:** The production

**Deleted:** is evaluated as follows:

2. Let *constructor* be GetValue(*ref*).
- 3. If *constructor* is an abrupt completion, return *constructor*.**
4. If Type(*constructor*) is not Object, throw a **TypeError** exception.
5. If *constructor* does not implement the [[Construct]] internal method, throw a **TypeError** exception.
6. Return the result of calling the [[Construct]] internal method on *constructor*, providing no arguments (that is, an empty list of arguments).

*MemberExpression : new MemberExpression Arguments*

1. Let *ref* be the result of evaluating *MemberExpression*.
2. Let *constructor* be GetValue(*ref*).
- 3. If *constructor* is an abrupt completion, return *constructor*.**
4. Let *argList* be the result of evaluating *Arguments*, producing an internal list of argument values (11.2.4).
- 5. If *argList* is an abrupt completion, return *argList*.**
6. If Type(*constructor*) is not Object, throw a **TypeError** exception.
7. If *constructor* does not implement the [[Construct]] internal method, throw a **TypeError** exception.
8. Return the result of calling the [[Construct]] internal method on *constructor*, providing the list *argList* as the argument values.

### 11.2.3 Function Calls

#### Runtime Semantics: Evaluation

*CallExpression : MemberExpression Arguments*

1. Let *ref* be the result of evaluating *MemberExpression*.
2. Return the result of the abstract operation *Evaluate Call* with arguments *ref* and *Arguments*.

*CallExpression : CallExpression Arguments*

1. Let *ref* be the result of evaluating *CallExpression*.
2. Return the result of the abstract operation *Evaluate Call* with arguments *ref* and *Arguments*.

The abstract operation *Evaluate Call* takes as arguments a **Reference**, *ref*, and a syntactic grammar production *arguments*. It performs the following steps:

3. Let *ref* be the result of evaluating *MemberExpression*.
4. Let *func* be GetValue(*ref*).
- 5. If *func* is an abrupt completion, return *func*.**
6. Let *argList* be the result of evaluating *Arguments*, producing an internal list of argument values (see 11.2.4).
- 7. If *argList* is an abrupt completion, return *argList*.**
8. If Type(*func*) is not Object, throw a **TypeError** exception.
9. If IsCallable(*func*) is **false**, throw a **TypeError** exception.
10. If Type(*ref*) is Reference, then
  - a. If IsPropertyReference(*ref*) is **true**, then
    - i. Let *thisValue* be *GetThisValue*(*ref*).
  - b. Else, the base of *ref* is an Environment Record
    - i. Let *thisValue* be the result of calling the *WithBaseObject* concrete method of *GetBase*(*ref*).
11. Else, Type(*ref*) is not Reference.
  - a. Let *thisValue* be **undefined**.
12. Return the result of calling the [[Call]] internal method on *func*, providing *thisValue* as the **this** value and providing the list *argList* as the argument values.

**NOTE** The returned result will never be of type Reference if *func* is a native ECMAScript object. Whether calling a host object can return a value of type Reference is implementation-dependent. If a value of type Reference is returned, it must be a non-strict Property Reference.

### 11.2.4 The *super* Keyword

#### Static Semantics

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Comment [AWB728]:** TODO: tail calls.

Jan 19 meeting notes: Tentative decision is to support tail calls in strict mode only.

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** GetBase

**Deleted:** ImplicitThisValue

**Deleted:** The This production *CallExpression : CallExpression Arguments* is evaluated in exactly the same manner, as *CallExpression : MemberExpression Arguments* except that the contained *CallExpression* is evaluated in step

**Deleted:** 3

**Formatted:** Font: (Default) Arial, Not Bold

**Formatted:** Font: Courier New, 11 pt

**Formatted:** Font: Bold

**Formatted:** Normal

**Formatted:** Font: Helvetica, Bold

## Static Semantics: Early Errors

MemberExpression :

super [ Expression ]  
super . IdentifierName

- It is a Syntax Error if the source code parsed with this production is global code that is not eval code.
- It is a Syntax Error if the source code parsed with this production is eval code and the source code is not being processed by a direct call to eval that is contained in function code.

CallExpression : super Arguments

- It is a Syntax Error if the source code parsed with this production is global code that is not eval code.
- It is a Syntax Error if the source code parsed with this production is eval code and the source code is not being processed by a direct call to eval that is contained in function code.

## Runtime Semantics: Evaluation

MemberExpression : super [ Expression ]

1. Let `env` be the result of performing the `GetThisEnvironment` abstract operation.
2. If the result of calling the `HasSuperBinding` concrete method of `env` is `false`, then throw `ReferenceError`.
3. Let `actualThis` be the result of calling the `GetThisBinding` concrete method of `env`.
4. Let `baseValue` be the result of calling the `GetHomeObject` concrete method of `env`.
5. Let `propertyNameReference` be the result of evaluating `Expression`.
6. Let `propertyNameValue` be `GetValue(propertyNameReference)`.
7. `ReturnIfAbrupt(CheckObjectCoercible(baseValue))`.
8. Let `propertyKey` be `ToPropertyKey(propertyNameValue)`.
9. If the code matched by the syntactic production that is being evaluated is strict mode code, let `strict` be `true`, else let `strict` be `false`.
10. Return a value of type Reference that is a Super Reference whose base value is `baseValue`, whose referenced name is `propertyKey`, whose `thisValue` is `actualThis`, and whose strict mode flag is `strict`.

MemberExpression : super . IdentifierName

1. Let `env` be the result of performing the `GetThisEnvironment` abstract operation.
2. If the result of calling the `HasSuperBinding` concrete method of `env` is `false`, then throw `ReferenceError`.
3. Let `actualThis` be the result of calling the `GetThisBinding` concrete method of `env`.
4. Let `baseValue` be the result of calling the `GetHomeObject` concrete method of `env`.
5. `ReturnIfAbrupt(CheckObjectCoercible(baseValue))`.
6. Let `propertyKey` be String Value of `IdentifierName`.
7. If the code matched by the syntactic production that is being evaluated is strict mode code, let `strict` be `true`, else let `strict` be `false`.
8. Return a value of type Reference that is a Super Reference whose base value is `baseValue`, whose referenced name is `propertyKey`, whose `thisValue`.

CallExpression : super Arguments

1. Let `env` be the result of performing the `GetThisEnvironment` abstract operation.
2. If the result of calling the `HasSuperBinding` concrete method of `env` is `false`, then throw `ReferenceError`.
3. Let `actualThis` be the result of calling the `GetThisBinding` concrete method of `env`.
4. Let `baseValue` be the result of calling the `GetHomeObject` concrete method of `env`.
5. `ReturnIfAbrupt(CheckObjectCoercible(baseValue))`.
6. Let `propertyKey` be the result of calling the `GetMethodNames` concrete method of `env`.
7. If the code matched by the syntactic production that is being evaluated is strict mode code, let `strict` be `true`, else let `strict` be `false`.
8. Let `ref` be a value of type Reference that is a Super Reference whose base value is `baseValue`, whose referenced name is `propertyKey`, whose `thisValue`.
9. Return the result of the abstract operation `Evaluate Call` with arguments `ref` and `Arguments`.

**Deleted:** code

**Formatted:** Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

**Deleted:** that is also global code

**Deleted:** <#>It is a Syntax Error if the source code parsed with this production is not extended code.¶

**Formatted:** Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

**Deleted:** code

**Deleted:** that is also global code

**Deleted:** <#>It is a Syntax Error if the source code parsed with this production is not extended code.¶

**Deleted:** <#>Let `baseReference` be the result of evaluating `MemberExpression`.¶  
 <#>Let `baseValue` be `GetValue(baseReference)`.¶

**Deleted:** Call `CheckObjectCoercible(baseValue)`.

**Deleted:** `NameString`

**Deleted:** String

**Deleted:** and

**Deleted:** `NameString`

## 11.2.5 Argument Lists

The evaluation of an argument list produces a List of values (see 8.7).

### Runtime Semantics

#### Runtime Semantics: Evaluation

Arguments : ( )

1. Return an empty List.

Arguments : ( ArgumentList )

1. Return the result of evaluating *ArgumentList*.

ArgumentList : AssignmentExpression

1. Let *ref* be the result of evaluating *AssignmentExpression*.
2. Let *arg* be *GetValue(ref)*.
3. If *arg* is an abrupt completion, return *arg*.
4. Return a List whose sole item is *arg*.

ArgumentList : ... AssignmentExpression

1. Let *list* be an empty List.
2. Let *spreadRef* be the result of evaluating *AssignmentExpression*.
3. Let *spreadValue* be *GetValue(spreadRef)*.
4. Let *spreadObj* be *ToObject(spreadValue)*.
5. If *spreadObj* is an abrupt completion, return *spreadObj*.
6. Let *lenVal* be the result of calling the **[[Get]]** internal method of *spreadObj* with argument "**length**".
7. Let *spreadLen* be *ToInt32(lenVal)*.
8. ReturnIfAbrupt(*spreadLen*).
9. Let *n* = 0.
10. Repeat, while *n* < *spreadLen*
  - a. Let *nextArg* be the result of calling the **[[Get]]** internal method of *spreadObj* passing *ToString(n)* as the argument.
  - b. ReturnIfAbrupt(*nextArg*).
  - c. Append *nextArg* as the last element of *list*.
  - d. Let *n* = *n*+1.
11. Return *list*.

ArgumentList : ArgumentList , AssignmentExpression

1. Let *precedingArgs* be the result of evaluating *ArgumentList*.
2. If *precedingArgs* is an abrupt completion, return *precedingArgs*.
3. Let *ref* be the result of evaluating *AssignmentExpression*.
4. Let *arg* be *GetValue(ref)*.
5. If *arg* is an abrupt completion, return *arg*.
6. Return a List whose length is one greater than the length of *precedingArgs* and whose items are the items of *precedingArgs*, in order, followed at the end by *arg* which is the last item of the new list.

ArgumentList : ArgumentList , ... AssignmentExpression

1. Let *precedingArgs* be an empty List.
2. Let *spreadRef* be the result of evaluating *AssignmentExpression*.
3. Let *spreadValue* be *GetValue(spreadRef)*.
4. Let *spreadObj* be *ToObject(spreadValue)*.
5. If *spreadObj* is an abrupt completion, return *spreadObj*.

**Deleted:** 4

**Deleted:** 8

**Deleted:** 11.2.45.1 Static Semantics  
**Static Semantics:** Early Errors

The production *Arguments* : ( ) is evaluated follows:  
↳ Return an empty List.

The production *Arguments* : ( *ArgumentList* ) evaluated as follows:  
↳ Return the result of evaluating *ArgumentList*.

The production *ArgumentList* : *AssignmentExpression* is evaluated as follows:  
↳ Let *ref* be the result of evaluating *AssignmentExpression*.

↳ Let *arg* be *GetValue(ref)*.  
↳ Return a List whose sole item is *arg*.

The static semantics of the production *ArgumentList* : ... *AssignmentExpression* are:  
↳ It is a Syntax Error if the sourcecode parsed with this production is not extended code.

The static semantics of the production *ArgumentList* : *ArgumentList* , ... *AssignmentExpression* are:  
↳ It is a Syntax Error if the sourcecode parsed with this production is not extended code.

**11.2.45.2**

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Comment [AW29]:** Note that the value the spread operator is applied to is coerced to an Object.

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Comment [AW30]:** Note that the value the spread operator is applied to is coerced to an Object.

6. Let `lenVal` be the result of calling the `[[Get]]` internal method of `spreadObj` with argument `"length"`.
7. Let `spreadLen` be `ToUint32(lenVal)`.
8. `ReturnIfAbrupt(spreadLen)`.
9. Let `n = 0`.
10. Repeat, while `n < spreadLen`
  - a. Let `nextArg` be the result of calling the `[[Get]]` internal method of `spreadObj` passing `ToString(n)` as the argument.
  - b. `ReturnIfAbrupt(nextArg)`.
  - c. Append `nextArg` as the last element of `precedingArgs`.
  - d. Let `n = n+1`.
11. `Return precedingArgs`.

### 11.3 Postfix Expressions

#### Syntax

*PostfixExpression :*

- LeftHandSideExpression*
- LeftHandSideExpression* [no LineTerminator here] `++`
- LeftHandSideExpression* [no LineTerminator here] `--`

#### Static Semantics

##### Static Semantics: Early Errors

*PostfixExpression :*

- LeftHandSideExpression* [no LineTerminator here] `++`
- LeftHandSideExpression* [no LineTerminator here] `--`

- It is a Syntax Error if the *PostfixExpression* is contained in strict code and *LeftHandSideExpression* is the Identifier `eval` or the Identifier `arguments`.
- It is a Syntax Error if the *LeftHandSideExpression* is *PrimaryExpression : ( Expression )* and *Expression* derived a production that if used in place of *LeftHandSideExpression* would produce a Syntax Error according to these rules. This rule is recursively applied.

##### 11.3.1 Postfix Increment Operator

##### Runtime Semantics: Evaluation

*PostfixExpression : LeftHandSideExpression* [no LineTerminator here] `++`

1. Let `lhs` be the result of evaluating *LeftHandSideExpression*.
2. Let `oldValue` be `ToNumber(GetValue(lhs))`.
3. If `oldValue` is an abrupt completion, return `oldValue`.
4. Let `newValue` be the result of adding the value `1` to `oldValue`, using the same rules as for the `+` operator (see 11.6.3).
5. If `PutValue(lhs, newValue)` is an abrupt completion, return that Completion Record.
6. Return `oldValue`.

##### 11.3.2 Postfix Decrement Operator

##### Runtime Semantics: Evaluation

*PostfixExpression : LeftHandSideExpression* [no LineTerminator here] `--`

1. Let `lhs` be the result of evaluating *LeftHandSideExpression*.
2. Let `oldValue` be `ToNumber(GetValue(lhs))`.
3. Let `newValue` be the result of subtracting the value `1` from `oldValue`, using the same rules as for the `-` operator (11.6.3).

**Formatted:** Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

**Deleted:** ~~<#>It is a Syntax Error if the AssignmentExpression is contained in extended code and the LeftHandSideExpression is a Literal or a FunctionExpression.~~  
~~<#>It is a Syntax Error if the AssignmentExpression is contained in extended code and the LeftHandSideExpression is an Identifier that does not statically resolve to a declarative environment record binding or if the resolved binding is an immutable binding.~~

**Deleted:** that

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** ~~<#>Throw a `SyntaxError` exception if the following conditions are all true:  
<#>`Type(lhs)` Reference is `true`  
<#>`IsStrictReference(lhs)` is `true`  
<#>`Type(GetBase(lhs))` Environment Record  
<#>`GetReferencedName(lhs)` is either `"eval"` or `"arguments"`~~

**Deleted:** Call

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** ~~<#>Throw a `SyntaxError` exception if the following conditions are all true:  
<#>`Type(lhs)` Reference is `true`  
<#>`IsStrictReference(lhs)` is `true`  
<#>`Type(GetBase(lhs))` Environment Record  
<#>`GetReferencedName(lhs)` is either `"eval"` or `"arguments"`~~

4. If Call PutValue(*lhs*, *newValue*) is an abrupt completion, return that Completion Record.
5. Return *oldValue*.

## 11.4 Unary Operators

### Syntax

*UnaryExpression* :  
*PostfixExpression*  
**delete** *UnaryExpression*  
**void** *UnaryExpression*  
**typeof** *UnaryExpression*  
**++** *UnaryExpression*  
**--** *UnaryExpression*  
**+** *UnaryExpression*  
**-** *UnaryExpression*  
**~** *UnaryExpression*  
**!** *UnaryExpression*

### Static Semantics

#### Static Semantics: Early Errors

*UnaryExpression* :  
**delete** *UnaryExpression*  
**++** *UnaryExpression*  
**--** *UnaryExpression*

- It is a Syntax Error if the *UnaryExpression* is contained in strict code and the derived *UnaryExpression* is the Identifier **eval** or the Identifier **arguments**.
- It is a Syntax Error if the derived *UnaryExpression* is *PrimaryExpression* : **( Expression )** and *Expression* derived a production that if used in place of *LeftHandSideExpression* would produce a Syntax Error according to these rules. This rule is recursively applied.

**Formatted:** Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

**Deleted:** <#>It is a Syntax Error if the *UnaryExpression* is contained in extended code and the derived *UnaryExpression* is a *Literal* or *FunctionExpression*.¶

<#>It is a Syntax Error if the *UnaryExpression* contained in extended code and the derived *UnaryExpression* is an *Identifier* that does not statically resolve to a declarative environment record binding or if the resolved binding is an immutable binding.¶

#### 11.4.1 The delete Operator

#### Static Semantics: Early Errors

*UnaryExpression* : **delete** *UnaryExpression*

- It is a Syntax Error if the *UnaryExpression* is contained in strict code and the *UnaryExpression* derives an *Identifier* that statically resolves to a environment record.

**Formatted:** Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

#### Runtime Semantics: Evaluation

*UnaryExpression* : **delete** *UnaryExpression*

1. Let *ref* be the result of evaluating *UnaryExpression*.
2. If *ref* is an abrupt completion, return *ref*.
3. If Type(*ref*) is not Reference, return **true**.
4. If IsUnresolvableReference(*ref*) is **true**, then
  - a. If IsStrictReference(*ref*) is **true**, throw a **SyntaxError** exception.
  - b. Else, return **true**.
5. If IsPropertyReference(*ref*) is **true**, then
  - a. If IsSuperReference(*ref*), then throw a **ReferenceError** exception.
  - b. Return the result of calling the **[[Delete]]** internal method on **ToObject**(**GetBase**(*ref*)) providing **GetReferencedName**(*ref*) and **IsStrictReference**(*ref*) as the arguments.
6. Else, *ref* is a Reference to an Environment Record binding, so
  - a. Let *bindings* be **GetBase**(*ref*).

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** <#>If IsStrictReference(*ref*) is **true**, throw a **SyntaxError** exception.¶

- b. Return the result of calling the `DeleteBinding` concrete method of `bindings`, providing `GetReferencedName(ref)` as the argument.

**NOTE** When a `delete` operator occurs within strict mode code, a `SyntaxError` exception is thrown if its `UnaryExpression` is a direct reference to a variable, function argument, or function name. In addition, if a `delete` operator occurs within strict mode code and the property to be deleted has the attribute `{ [[Configurable]]: false }`, a `TypeError` exception is thrown.

#### 11.4.2 The `void` Operator

##### Runtime Semantics: Evaluation

*UnaryExpression : void UnaryExpression*

1. Let `expr` be the result of evaluating `UnaryExpression`.
2. Call `GetValue(expr)`.
3. **If *ref* is an abrupt completion, return *ref*.**
4. Return `undefined`.

**NOTE** `GetValue` must be called even though its value is not used because it may have observable side-effects.

#### 11.4.3 The `typeof` Operator

##### Runtime Semantics: Evaluation

*UnaryExpression : typeof UnaryExpression*

1. Let `val` be the result of evaluating `UnaryExpression`.
2. If `Type(val)` is Reference, then
  - a. If `IsUnresolvableReference(val)` is `true`, return `"undefined"`.
  - b. Let `val` be `GetValue(val)`.
3. **If *val* is an abrupt completion, return *val*.**
4. Return a String determined by `Type(val)` according to Table 21.

Table 21 — `typeof` Operator Results

Type of <i>val</i>	Result
Undefined	<code>"undefined"</code>
Null	<code>"object"</code>
Boolean	<code>"boolean"</code>
Number	<code>"number"</code>
String	<code>"string"</code>
Object (native and does not implement <code>[[Call]]</code> )	<code>"object"</code>
Object (native or host and does implement <code>[[Call]]</code> )	<code>"function"</code>
Object (host and does not implement <code>[[Call]]</code> )	Implementation-defined except may not be <code>"undefined"</code> , <code>"boolean"</code> , <code>"number"</code> , or <code>"string"</code> .

#### 11.4.4 Prefix Increment Operator

##### Runtime Semantics: Evaluation

*UnaryExpression : ++ UnaryExpression*

1. Let `expr` be the result of evaluating `UnaryExpression`.

**Deleted:** The production  
**Deleted:** is evaluated as follows:  
**Formatted:** Space After: 0 pt

**Deleted:** The production  
**Deleted:** is evaluated as follows:  
**Deleted:** 20  
**Deleted:** 20

2. Let *oldValue* be ToNumber(GetValue(*expr*)).
3. If *oldValue* is an abrupt completion, return *oldValue*.
4. Let *newValue* be the result of adding the value 1 to *oldValue*, using the same rules as for the + operator (see 11.6.3).
5. If PutValue(*expr*, *newValue*) is an abrupt completion, return that Completion Record.
6. Return *newValue*.

#### 11.4.5 Prefix Decrement Operator

##### Runtime Semantics: Evaluation

*UnaryExpression* : -- *UnaryExpression*

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* be ToNumber(GetValue(*expr*)).
3. If *oldValue* is an abrupt completion, return *oldValue*.
4. Let *newValue* be the result of subtracting the value 1 from *oldValue*, using the same rules as for the - operator (see 11.6.3).
5. If PutValue(*expr*, *newValue*) is an abrupt completion, return that Completion Record.
6. Return *newValue*.

#### 11.4.6 Unary + Operator

NOTE The unary + operator converts its operand to Number type.

##### Runtime Semantics: Evaluation

*UnaryExpression* : + *UnaryExpression*

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Return ToNumber(GetValue(*expr*)).

#### 11.4.7 Unary - Operator

NOTE The unary - operator converts its operand to Number type and then negates it. Negating +0 produces -0, and negating -0 produces +0.

**Deleted:** <#> Throw a SyntaxError exception if the following conditions are all true:  
 <#> Type(*expr*) is Reference is true  
 <#> IsStrictReference(*expr*) is true  
 <#> Type(GetBase(*expr*)) is Environment Record  
 <#> GetReferencedName(*expr*) is either "eval" or "arguments"  
**Deleted:** Call

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** <#> Throw a SyntaxError exception if the following conditions are all true:  
 <#> Type(*expr*) is Reference is true  
 <#> IsStrictReference(*expr*) is true  
 <#> Type(GetBase(*expr*)) is Environment Record  
 <#> GetReferencedName(*expr*) is either "eval" or "arguments"  
**Deleted:** Call

**Deleted:** The production

**Formatted:** Font: 9 pt

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** Note that negating

**Deleted:** The production

**Deleted:** is evaluated as follows:

##### Runtime Semantics: Evaluation

*UnaryExpression* : - *UnaryExpression*

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* be ToNumber(GetValue(*expr*)).
3. If *oldValue* is an abrupt completion, return *oldValue*.
4. If *oldValue* is NaN, return NaN.
5. Return the result of negating *oldValue*; that is, compute a Number with the same magnitude but opposite sign.

#### 11.4.8 Bitwise NOT Operator ( ~ )

##### Runtime Semantics: Evaluation

*UnaryExpression* : ~ *UnaryExpression*

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* beToInt32(GetValue(*expr*)).
3. If *oldValue* is an abrupt completion, return *oldValue*.
4. Return the result of applying bitwise complement to *oldValue*. The result is a signed 32-bit integer.

**Deleted:** The production

**Deleted:** is evaluated as follows:

#### 11.4.9 Logical NOT Operator ( ! )

##### Runtime Semantics: Evaluation

*UnaryExpression* : ! *UnaryExpression*

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* be ToBoolean(GetValue(*expr*)).
- 3. If *oldValue* is an abrupt completion, return *oldValue*.**
4. If *oldValue* is true, return false.
5. Return true.

**Deleted:** The production

**Deleted:** is evaluated as follows:

#### 11.5 Multiplicative Operators

##### Syntax

*MultiplicativeExpression* :

*UnaryExpression*

*MultiplicativeExpression* \* *UnaryExpression*

*MultiplicativeExpression* / *UnaryExpression*

*MultiplicativeExpression* % *UnaryExpression*

##### Runtime Semantics: Evaluation

The production *MultiplicativeExpression* : *MultiplicativeExpression* @ *UnaryExpression*, where @ stands for one of the operators in the above definitions, is evaluated as follows:

**Deleted:** Semantics

1. Let *left* be the result of evaluating *MultiplicativeExpression*.
2. Let *leftValue* be GetValue(*left*).
- 3. If *leftValue* is an abrupt completion, return *leftValue*.**
4. Let *right* be the result of evaluating *UnaryExpression*.
5. Let *rightValue* be GetValue(*right*).
6. Let *leftNum* be ToNumber(*leftValue*).
- 7. If *leftNum* is an abrupt completion, return *leftNum*.**
8. Let *rightNum* be ToNumber(*rightValue*).
- 9. If *rightNum* is an abrupt completion, return *rightNum*.**
10. Return the result of applying the specified operation (\*, /, or %) to *leftNum* and *rightNum*. See the Notes below 11.5.1, 11.5.2, 11.5.3.

##### 11.5.1 Applying the \* Operator

The \* operator performs multiplication, producing the product of its operands. Multiplication is commutative. Multiplication is not always associative in ECMAScript, because of finite precision.

The result of a floating-point multiplication is governed by the rules of IEEE 754 binary double-precision arithmetic:

- If either operand is NaN, the result is NaN.
- The sign of the result is positive if both operands have the same sign, negative if the operands have different signs.
- Multiplication of an infinity by a zero results in NaN.
- Multiplication of an infinity by an infinity results in an infinity. The sign is determined by the rule already stated above.
- Multiplication of an infinity by a finite nonzero value results in a signed infinity. The sign is determined by the rule already stated above.
- In the remaining cases, where neither an infinity or NaN is involved, the product is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest mode. If the magnitude is too large to represent, the result is then an infinity of appropriate sign. If

the magnitude is too small to represent, the result is then a zero of appropriate sign. The ECMAScript language requires support of gradual underflow as defined by IEEE 754.

### 11.5.2 Applying the / Operator

The / operator performs division, producing the quotient of its operands. The left operand is the dividend and the right operand is the divisor. ECMAScript does not perform integer division. The operands and result of all division operations are double-precision floating-point numbers. The result of division is determined by the specification of IEEE 754 arithmetic:

- If either operand is **NaN**, the result is **NaN**.
- The sign of the result is positive if both operands have the same sign, negative if the operands have different signs.
- Division of an infinity by an infinity results in **NaN**.
- Division of an infinity by a zero results in an infinity. The sign is determined by the rule already stated above.
- Division of an infinity by a nonzero finite value results in a signed infinity. The sign is determined by the rule already stated above.
- Division of a finite value by an infinity results in zero. The sign is determined by the rule already stated above.
- Division of a zero by a zero results in **NaN**; division of zero by any other finite value results in zero, with the sign determined by the rule already stated above.
- Division of a nonzero finite value by a zero results in a signed infinity. The sign is determined by the rule already stated above.
- In the remaining cases, where neither an infinity, nor a zero, nor **NaN** is involved, the quotient is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest mode. If the magnitude is too large to represent, the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent, the operation underflows and the result is a zero of the appropriate sign. The ECMAScript language requires support of gradual underflow as defined by IEEE 754.

### 11.5.3 Applying the % Operator

The % operator yields the remainder of its operands from an implied division; the left operand is the dividend and the right operand is the divisor.

**NOTE** In C and C++, the remainder operator accepts only integral operands; in ECMAScript, it also accepts floating-point operands.

The result of a floating-point remainder operation as computed by the % operator is not the same as the “remainder” operation defined by IEEE 754. The IEEE 754 “remainder” operation computes the remainder from a rounding division, not a truncating division, and so its behaviour is not analogous to that of the usual integer remainder operator. Instead the ECMAScript language defines % on floating-point operations to behave in a manner analogous to that of the Java integer remainder operator; this may be compared with the C library function fmod.

The result of an ECMAScript floating-point remainder operation is determined by the rules of IEEE arithmetic:

- If either operand is **NaN**, the result is **NaN**.
- The sign of the result equals the sign of the dividend.
- If the dividend is an infinity, or the divisor is a zero, or both, the result is **NaN**.
- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
- If the dividend is a zero and the divisor is nonzero and finite, the result is the same as the dividend.
- In the remaining cases, where neither an infinity, nor a zero, nor **NaN** is involved, the floating-point remainder r from a dividend n and a divisor d is defined by the mathematical relation  $r = n - (d \times q)$  where q is an integer that is negative only if  $n/d$  is negative and positive only if  $n/d$  is positive, and whose magnitude is as large as possible without

exceeding the magnitude of the true mathematical quotient of n and d. r is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest mode.

## 11.6 Additive Operators

### Syntax

```
AdditiveExpression :
  MultiplicativeExpression
  AdditiveExpression + MultiplicativeExpression
  AdditiveExpression - MultiplicativeExpression
```

#### 11.6.1 The Addition operator (+)

NOTE The addition operator either performs string concatenation or numeric addition.

**Formatted:** Font: 9 pt

##### Runtime Semantics: Evaluation

AdditiveExpression : AdditiveExpression + MultiplicativeExpression

**Deleted:** The production

**Deleted:** is evaluated as follows:

1. Let *lref* be the result of evaluating AdditiveExpression.
2. Let *lval* be GetValue(*lref*).
3. If *lval* is an abrupt completion, return *lval*.
4. Let *rref* be the result of evaluating MultiplicativeExpression.
5. Let *rval* be GetValue(*rref*).
6. If *rval* is an abrupt completion, return *rval*.
7. Let *lprim* be ToPrimitive(*lval*).
8. If *lprim* is an abrupt completion, return *lprim*.
9. Let *rprim* be ToPrimitive(*rval*).
10. If *rprim* is an abrupt completion, return *rprim*.
11. If Type(*lprim*) is String or Type(*rprim*) is String, then
  - a. Return the String that is the result of concatenating ToString(*lprim*) followed by ToString(*rprim*)
12. Return the result of applying the addition operation to ToNumber(*lprim*) and ToNumber(*rprim*). See the Note below 11.6.3.

NOTE 1 No hint is provided in the calls to ToPrimitive in steps 5 and 6. All native ECMAScript objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Host objects may handle the absence of a hint in some other manner.

NOTE 2 Step 7 differs from step 3 of the comparison algorithm for the relational operators (11.8.1), by using the logical-or operation instead of the logical-and operation.

**Deleted:** 5

#### 11.6.2 The Subtraction Operator (-)

##### Runtime Semantics: Evaluation

AdditiveExpression : AdditiveExpression - MultiplicativeExpression

**Deleted:** The production

**Deleted:** is evaluated as follows:

1. Let *lref* be the result of evaluating AdditiveExpression.
2. Let *lval* be GetValue(*lref*).
3. If *lval* is an abrupt completion, return *lval*.
4. Let *rref* be the result of evaluating MultiplicativeExpression.
5. Let *rval* be GetValue(*rref*).
6. If *rval* is an abrupt completion, return *rval*.
7. Let *lnum* be ToNumber(*lval*).
8. If *lnum* is an abrupt completion, return *lnum*.
9. Let *rnum* be ToNumber(*rval*).
10. If *rnum* is an abrupt completion, return *rnum*.
11. Return the result of applying the subtraction operation to *lnum* and *rnum*. See the note below 11.6.3.

### 11.6.3 Applying the Additive Operators to Numbers

The `+` operator performs addition when applied to two operands of numeric type, producing the sum of the operands. The `-` operator performs subtraction, producing the difference of two numeric operands.

Addition is a commutative operation, but not always associative.

The result of an addition is determined using the rules of IEEE 754 binary double-precision arithmetic:

- If either operand is **NaN**, the result is **NaN**.
- The sum of two infinities of opposite sign is **NaN**.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and a finite value is equal to the infinite operand.
- The sum of two negative zeroes is **-0**. The sum of two positive zeroes, or of two zeroes of opposite sign, is **+0**.
- The sum of a zero and a nonzero finite value is equal to the nonzero operand.
- The sum of two nonzero finite values of the same magnitude and opposite sign is **+0**.
- In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, and the operands have the same sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest mode. If the magnitude is too large to represent, the operation overflows and the result is then an infinity of appropriate sign. The ECMAScript language requires support of gradual underflow as defined by IEEE 754.

The `-` operator performs subtraction when applied to two operands of numeric type, producing the difference of its operands; the left operand is the minuend and the right operand is the subtrahend. Given numeric operands  $a$  and  $b$ , it is always the case that  $a - b$  produces the same result as  $a + (-b)$ .

## 11.7 Bitwise Shift Operators

### Syntax

```
ShiftExpression :  
    AdditiveExpression  
    ShiftExpression << AdditiveExpression  
    ShiftExpression >> AdditiveExpression  
    ShiftExpression >>> AdditiveExpression
```

### 11.7.1 The Left Shift Operator (**<<**)

NOTE Performs a bitwise left shift operation on the left operand by the amount specified by the right operand.

**Formatted:** Font: 9 pt

#### Runtim Semantics: Evaluation

*ShiftExpression* : *ShiftExpression* **<<** *AdditiveExpression*

**Deleted:** The production

**Deleted:** is evaluated as follows:

1. Let *lref* be the result of evaluating *ShiftExpression*.
2. Let *lval* be *GetValue(lref)*.
3. **If lval is an abrupt completion, return lval.**
4. Let *rref* be the result of evaluating *AdditiveExpression*.
5. Let *rval* be *GetValue(rref)*.
6. **If rval is an abrupt completion, return rval.**
7. Let *lnum* be *ToInt32(lval)*.
8. **If lnum is an abrupt completion, return lnum.**
9. Let *rnum* be *ToInt32(rval)*.
10. **If rnum is an abrupt completion, return rnum.**
11. Let *shiftCount* be the result of masking out all but the least significant 5 bits of *rnum*, that is, compute *rnum* & `0x1F`.

12. Return the result of left shifting *lnum* by *shiftCount* bits. The result is a signed 32-bit integer.

### 11.7.2 The Signed Right Shift Operator ( $>>$ )

NOTE Performs a sign-filling bitwise right shift operation on the left operand by the amount specified by the right operand.

Formatted: Font: 9 pt

#### Runtime Semantics: Evaluation

*ShiftExpression* : *ShiftExpression*  $>>$  *AdditiveExpression*

Deleted: The production

Deleted: is evaluated as follows:

1. Let *lref* be the result of evaluating *ShiftExpression*.
2. Let *lval* be *GetValue(lref)*.
3. If *lval* is an abrupt completion, return *lval*.
4. Let *rref* be the result of evaluating *AdditiveExpression*.
5. Let *rval* be *GetValue(rref)*.
6. If *rval* is an abrupt completion, return *rval*.
7. Let *lnum* be *ToInt32(lval)*.
8. If *lnum* is an abrupt completion, return *lnum*.
9. Let *rnum* be *ToUint32(rval)*.
10. If *rnum* is an abrupt completion, return *rnum*.
11. Let *shiftCount* be the result of masking out all but the least significant 5 bits of *rnum*, that is, compute *rnum* & 0x1F.
12. Return the result of performing a sign-extending right shift of *lnum* by *shiftCount* bits. The most significant bit is propagated. The result is a signed 32-bit integer.

### 11.7.3 The Unsigned Right Shift Operator ( $>>>$ )

NOTE Performs a zero-filling bitwise right shift operation on the left operand by the amount specified by the right operand.

Formatted: Font: 9 pt

#### Runtime Semantics: Evaluation

*ShiftExpression* : *ShiftExpression*  $>>>$  *AdditiveExpression*

Deleted: The production

Deleted: is evaluated as follows:

1. Let *lref* be the result of evaluating *ShiftExpression*.
2. Let *lval* be *GetValue(lref)*.
3. If *lval* is an abrupt completion, return *lval*.
4. Let *rref* be the result of evaluating *AdditiveExpression*.
5. Let *rval* be *GetValue(rref)*.
6. If *rval* is an abrupt completion, return *rval*.
7. Let *lnum* be *ToUint32(lval)*.
8. If *lnum* is an abrupt completion, return *lnum*.
9. Let *rnum* be *ToUint32(rval)*.
10. If *rnum* is an abrupt completion, return *rnum*.
11. Let *shiftCount* be the result of masking out all but the least significant 5 bits of *rnum*, that is, compute *rnum* & 0x1F.
12. Return the result of performing a zero-filling right shift of *lnum* by *shiftCount* bits. Vacated bits are filled with zero. The result is an unsigned 32-bit integer.

## 11.8 Relational Operators

NOTE The result of evaluating a relational operator is always of type Boolean, reflecting whether the relationship named by the operator holds between its two operands.

Formatted: Font: 9 pt

## Syntax

*RelationalExpression* :

```

ShiftExpression
RelationalExpression < ShiftExpression
RelationalExpression > ShiftExpression
RelationalExpression <= ShiftExpression
RelationalExpression >= ShiftExpression
RelationalExpression instanceof ShiftExpression
RelationalExpression in ShiftExpression

```

*RelationalExpressionNoIn* :

```

ShiftExpression
RelationalExpressionNoIn < ShiftExpression
RelationalExpressionNoIn > ShiftExpression
RelationalExpressionNoIn <= ShiftExpression
RelationalExpressionNoIn >= ShiftExpression
RelationalExpressionNoIn instanceof ShiftExpression

```

The semantics of the *RelationalExpressionNoIn* productions are the same as the *RelationalExpression* productions except that the contained *RelationalExpressionNoIn* is used in place of the contained *RelationalExpression*.

NOTE The “*NoIn*” variants are needed to avoid confusing the *in* operator in a relational expression with the *in* operator in a *for* statement.

**Deleted:** evaluated in

**Deleted:** manner

**Deleted:** evaluated instead

**Deleted:** Semantics¶

The result of evaluating a relational operator always of type Boolean, reflecting whether the relationship named by the operator holds between its two operands.¶

The *RelationalExpressionNoIn* productions are evaluated in the same manner as the *RelationalExpression* productions except that the contained *RelationalExpressionNoIn* is evaluated instead of the contained *RelationalExpression*.

### 11.8.1 Runtime Semantics

#### Runtime Semantics: The Abstract Relational Comparison Algorithm

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

1. If  $x$  is an abrupt completion, return  $x$ .
2. If  $y$  is an abrupt completion, return  $y$ .
3. If the *LeftFirst* flag is **true**, then
  - a. Let  $px$  be the result of calling *ToPrimitive*( $x$ , hint Number).
  - b. If  $px$  is an abrupt completion, return  $px$ .
  - c. Let  $py$  be the result of calling *ToPrimitive*( $y$ , hint Number).
  - d. If  $py$  is an abrupt completion, return  $py$ .
4. Else the order of evaluation needs to be reversed to preserve left to right evaluation
  - a. Let  $py$  be the result of calling *ToPrimitive*( $y$ , hint Number).
  - b. If  $py$  is an abrupt completion, return  $py$ .
  - c. Let  $px$  be the result of calling *ToPrimitive*( $x$ , hint Number).
  - d. If  $px$  is an abrupt completion, return  $px$ .
5. If it is not the case that both *Type*( $px$ ) is String and *Type*( $py$ ) is String, then
  - a. Let  $nx$  be the result of calling *ToNumber*( $px$ ). Because  $px$  and  $py$  are primitive values evaluation order is not important.
  - b. Let  $ny$  be the result of calling *ToNumber*( $py$ ).
  - c. If  $nx$  is **NaN**, return **undefined**.
  - d. If  $ny$  is **NaN**, return **undefined**.
  - e. If  $nx$  and  $ny$  are the same Number value, return **false**.
  - f. If  $nx$  is **+0** and  $ny$  is **-0**, return **false**.
  - g. If  $nx$  is **-0** and  $ny$  is **+0**, return **false**.

- h. If  $nx$  is  $+\infty$ , return **false**.
  - i. If  $ny$  is  $+\infty$ , return **true**.
  - j. If  $ny$  is  $-\infty$ , return **false**.
  - k. If  $nx$  is  $-\infty$ , return **true**.
  - l. If the mathematical value of  $nx$  is less than the mathematical value of  $ny$  —note that these mathematical values are both finite and not both zero—return **true**. Otherwise, return **false**.
6. Else, both  $px$  and  $py$  are Strings
- a. If  $py$  is a prefix of  $px$ , return **false**. (A String value  $p$  is a prefix of String value  $q$  if  $q$  can be the result of concatenating  $p$  and some other String  $r$ . Note that any String is a prefix of itself, because  $r$  may be the empty String.)
  - b. If  $px$  is a prefix of  $py$ , return **true**.
  - c. Let  $k$  be the smallest nonnegative integer such that the character at position  $k$  within  $px$  is different from the character at position  $k$  within  $py$ . (There must be such a  $k$ , for neither String is a prefix of the other.)
  - d. Let  $m$  be the integer that is the code unit value for the character at position  $k$  within  $px$ .
  - e. Let  $n$  be the integer that is the code unit value for the character at position  $k$  within  $py$ .
  - f. If  $m < n$ , return **true**. Otherwise, return **false**.

NOTE 1 Step 3 differs from step 7 in the algorithm for the addition operator + (11.6.1) in using `and` instead of `or`.

NOTE 2 The comparison of Strings uses a simple lexicographic ordering on sequences of code unit values. There is no attempt to use the more complex, semantically oriented definitions of character or string equality and collating order defined in the Unicode specification. Therefore String values that are canonically equal according to the Unicode standard could test as unequal. In effect this algorithm assumes that both Strings are already in normalised form. Also, note that for strings containing supplementary characters, lexicographic ordering on sequences of UTF-16 code unit values differs from that on sequences of code point values.

**Formatted:** Note

## Runtime Semantics: Evaluation

*RelationalExpression : RelationalExpression < ShiftExpression*

1. Let  $lref$  be the result of evaluating *RelationalExpression*.
2. Let  $lval$  be `GetValue`( $lref$ ).
- 3. If  $lval$  is an abrupt completion, return  $lval$ .**
4. Let  $rref$  be the result of evaluating *ShiftExpression*.
5. Let  $rval$  be `GetValue`( $rref$ ).
6. Let  $r$  be the result of performing abstract relational comparison  $lval < rval$ . (see 11.8.5)
7. If  $r$  is **undefined**, return **false**. Otherwise, return  $r$ .

**Deleted:** 11.8.1 The Less-than Operator (<)¶  
The production

**Deleted:** is evaluated as follows:

*RelationalExpression : RelationalExpression > ShiftExpression*

1. Let  $lref$  be the result of evaluating *RelationalExpression*.
2. Let  $lval$  be `GetValue`( $lref$ ).
- 3. If  $lval$  is an abrupt completion, return  $lval$ .**
4. Let  $rref$  be the result of evaluating *ShiftExpression*.
5. Let  $rval$  be `GetValue`( $rref$ ).
6. Let  $r$  be the result of performing abstract relational comparison  $rval < lval$  with *LeftFirst* equal to **false**.
7. If  $r$  is **undefined**, return **false**. Otherwise, return  $r$ .

**Deleted:** 11.8.2 . The Greater-than Operator (>)¶  
The production

**Deleted:** is evaluated as follows:

*RelationalExpression : RelationalExpression <= ShiftExpression*

1. Let  $lref$  be the result of evaluating *RelationalExpression*.
2. Let  $lval$  be `GetValue`( $lref$ ).
- 3. If  $lval$  is an abrupt completion, return  $lval$ .**
4. Let  $rref$  be the result of evaluating *ShiftExpression*.
5. Let  $rval$  be `GetValue`( $rref$ ).
6. Let  $r$  be the result of performing abstract relational comparison  $rval < lval$  with *LeftFirst* equal to **false**.
7. If  $r$  is **true** or **undefined**, return **false**. Otherwise, return **true**.

**Deleted:** 11.8.3 The Less-than-or-equal Operator (<=)¶  
The production

**Deleted:** is evaluated as follows:

*RelationalExpression : RelationalExpression >= ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be *GetValue(lref)*.
3. **If *lval* is an abrupt completion, return *lval*.**
4. Let *rref* be the result of evaluating *ShiftExpression*.
5. Let *rval* be *GetValue(rref)*.
6. Let *r* be the result of performing abstract relational comparison *lval < rval*.
7. If *r* is **true** or **undefined**, return **false**. Otherwise, return **true**.

**Deleted:** 11.8.4 The Greater-than-or-equal Operator ( $\geq$ )  
The production

**Deleted:** is evaluated as follows:

**Deleted:** (see 11.8.5)

**Deleted:** is evaluated as follows:

*RelationalExpression: RelationalExpression instanceof ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be *GetValue(lref)*.
3. **If *lval* is an abrupt completion, return *lval*.**
4. Let *rref* be the result of evaluating *ShiftExpression*.
5. Let *rval* be *GetValue(rref)*.
6. **If *rval* is an abrupt completion, return *rval*.**
7. If *Type(rval)* is not **Object**, throw a **TypeError** exception.
8. If *rval* does not have a **[[HasInstance]]** internal method, throw a **TypeError** exception.
9. Return the result of calling the **[[HasInstance]]** internal method of *rval* with argument *lval*.

**Deleted:** 11.8.7 . The in operator  
The production

**Deleted:** is evaluated as follows:

*RelationalExpression : RelationalExpression in ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be *GetValue(lref)*.
3. **If *lval* is an abrupt completion, return *lval*.**
4. Let *rref* be the result of evaluating *ShiftExpression*.
5. Let *rval* be *GetValue(rref)*.
6. **If *rval* is an abrupt completion, return *rval*.**
7. If *Type(rval)* is not **Object**, throw a **TypeError** exception.
8. Return the result of calling the **[[HasProperty]]** internal method of *rval* with argument *ToString(lval)*.

**Formatted:** Note

## 11.9 Equality Operators

**NOTE** The result of evaluating an equality operator is always of type Boolean, reflecting whether the relationship named by the operator holds between its two operands.

### Syntax

*EqualityExpression :*  
*RelationalExpression*  
*EqualityExpression == RelationalExpression*  
*EqualityExpression != RelationalExpression*  
*EqualityExpression === RelationalExpression*  
*EqualityExpression !== RelationalExpression*  
*EqualityExpression [no LineTerminator here] is RelationalExpression*  
*EqualityExpression [no LineTerminator here] isnt RelationalExpression*

*EqualityExpressionNoIn :*  
*RelationalExpressionNoIn*  
*EqualityExpressionNoIn == RelationalExpressionNoIn*  
*EqualityExpressionNoIn != RelationalExpressionNoIn*  
*EqualityExpressionNoIn === RelationalExpressionNoIn*  
*EqualityExpressionNoIn !== RelationalExpressionNoIn*  
*EqualityExpression [no LineTerminator here] is RelationalExpression*  
*EqualityExpression [no LineTerminator here] isnt RelationalExpression*

The semantics of the *EqualityExpressionNoIn* productions are the same as the *EqualityExpression* productions except that the contained *EqualityExpressionNoIn* and *RelationalExpressionNoIn* are used in place of the contained *EqualityExpression* and *RelationalExpression*, respectively.

## 11.9.1 Runtime Semantics

### Runtime Semantics: The Abstract Equality Comparison Algorithm

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

1. If  $x$  is an abrupt completion, return  $x$ .
2. If  $y$  is an abrupt completion, return  $y$ .
3. If  $\text{Type}(x)$  is the same as  $\text{Type}(y)$ , then
  - a. If  $\text{Type}(x)$  is **Undefined**, return **true**.
  - b. If  $\text{Type}(x)$  is **Null**, return **true**.
  - c. If  $\text{Type}(x)$  is **Number**, then
    - i. If  $x$  is **Nan**, return **false**.
    - ii. If  $y$  is **Nan**, return **false**.
    - iii. If  $x$  is the same Number value as  $y$ , return **true**.
    - iv. If  $x$  is **+0** and  $y$  is **-0**, return **true**.
    - v. If  $x$  is **-0** and  $y$  is **+0**, return **true**.
    - vi. Return **false**.
  - d. If  $\text{Type}(x)$  is **String**, then return **true** if  $x$  and  $y$  are exactly the same sequence of characters (same length and same characters in corresponding positions). Otherwise, return **false**.
  - e. If  $\text{Type}(x)$  is **Boolean**, return **true** if  $x$  and  $y$  are both **true** or both **false**. Otherwise, return **false**.
  - f. Return **true** if  $x$  and  $y$  are the same **Object** value. Otherwise, return **false**.
4. If  $x$  is **null** and  $y$  is **undefined**, return **true**.
5. If  $x$  is **undefined** and  $y$  is **null**, return **true**.
6. If  $\text{Type}(x)$  is **Number** and  $\text{Type}(y)$  is **String**, return the result of the comparison  $x == \text{ToNumber}(y)$ .
7. If  $\text{Type}(x)$  is **String** and  $\text{Type}(y)$  is **Number**, return the result of the comparison  $\text{ToNumber}(x) == y$ .
8. If  $\text{Type}(x)$  is **Boolean**, return the result of the comparison  $\text{ToNumber}(x) == y$ .
9. If  $\text{Type}(y)$  is **Boolean**, return the result of the comparison  $x == \text{ToNumber}(y)$ .
10. If  $\text{Type}(x)$  is either **String** or **Number** and  $\text{Type}(y)$  is **Object**, return the result of the comparison  $x == \text{ToPrimitive}(y)$ .
11. If  $\text{Type}(x)$  is **Object** and  $\text{Type}(y)$  is either **String** or **Number**, return the result of the comparison  $\text{ToPrimitive}(x) == y$ .
12. Return **false**.

NOTE 1 Given the above definition of equality:

- String comparison can be forced by: `" " + a == " " + b`.
- Numeric comparison can be forced by: `+a == +b`.
- Boolean comparison can be forced by: `!a == !b`.

NOTE 2 The equality operators maintain the following invariants:

- $A != B$  is equivalent to  $!(A == B)$ .
- $A == B$  is equivalent to  $B == A$ , except in the order of evaluation of  $A$  and  $B$ .

NOTE 3 The equality operator is not always transitive. For example, there might be two distinct String objects, each representing the same String value; each String object would be considered equal to the String value by the `==` operator, but the two String objects would not be equal to each other. For Example:

- `new String("a") == "a"` and `"a" == new String("a")` are both **true**.
- `new String("a") == new String("a")` is **false**.

NOTE 4 Comparison of Strings uses a simple equality test on sequences of code unit values. There is no attempt to use the more complex, semantically oriented definitions of character or string equality and collating order defined in the

#### **Deleted:** Semantics¶

The result of evaluating an equality operator is always type Boolean, reflecting whether the relationship named by the operator holds between its two operands.¶

#### **Deleted:** evaluated in

#### **Deleted:** manner

#### **Deleted:** evaluated instead

#### **Deleted:** 1

#### **Deleted:** refer to

#### **Deleted:** object

Unicode specification. Therefore Strings values that are canonically equal according to the Unicode standard could test as unequal. In effect this algorithm assumes that both Strings are already in normalised form.

### Runtime Semantics: The Strict Equality Comparison Algorithm

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

1. If  $x$  is an abrupt completion, return  $x$ .
2. If  $y$  is an abrupt completion, return  $y$ .
3. If  $\text{Type}(x)$  is different from  $\text{Type}(y)$ , return **false**.
4. If  $\text{Type}(x)$  is **Undefined**, return **true**.
5. If  $\text{Type}(x)$  is **Null**, return **true**.
6. If  $\text{Type}(x)$  is **Number**, then
  - a. If  $x$  is **NaN**, return **false**.
  - b. If  $y$  is **NaN**, return **false**.
  - c. If  $x$  is the same Number value as  $y$ , return **true**.
  - d. If  $x$  is **+0** and  $y$  is **-0**, return **true**.
  - e. If  $x$  is **-0** and  $y$  is **+0**, return **true**.
  - f. Return **false**.
7. If  $\text{Type}(x)$  is **String**, then return **true** if  $x$  and  $y$  are exactly the same sequence of characters (same length and same characters in corresponding positions); otherwise, return **false**.
8. If  $\text{Type}(x)$  is **Boolean**, return **true** if  $x$  and  $y$  are both **true** or both **false**; otherwise, return **false**.
9. Return **true** if  $x$  and  $y$  are the same Object value. Otherwise, return **false**.

NOTE This algorithm differs from the **SameValue Algorithm** (9.12) in its treatment of signed zeroes and NaNs.

**Deleted:** refer to  
**Deleted:** object

### Runtime Semantics: Evaluation

*EqualityExpression : EqualityExpression == RelationalExpression*

1. Let  $lref$  be the result of evaluating *EqualityExpression*.
2. Let  $lval$  be  $\text{GetValue}(lref)$ .
3. If  $lval$  is an abrupt completion, return  $lval$ .
4. Let  $rref$  be the result of evaluating *RelationalExpression*.
5. Let  $rval$  be  $\text{GetValue}(rref)$ .
6. Return the result of performing abstract equality comparison algorithm  $rval == lval$ .

**Deleted:** 11.9.1 The Equals Operator ( $==$ )  
The production

**Deleted:** is evaluated as follows:

*EqualityExpression : EqualityExpression != RelationalExpression*

1. Let  $lref$  be the result of evaluating *EqualityExpression*.
2. Let  $lval$  be  $\text{GetValue}(lref)$ .
3. If  $lval$  is an abrupt completion, return  $lval$ .
4. Let  $rref$  be the result of evaluating *RelationalExpression*.
5. Let  $rval$  be  $\text{GetValue}(rref)$ .
6. If  $rval$  is an abrupt completion, return  $rval$ .
7. Let  $r$  be the result of performing abstract equality comparison algorithm  $rval == lval$ .
8. If  $r$  is **true**, return **false**. Otherwise, return **true**.

**Deleted:** (see 11.9.3).

**Deleted:** 11.9.2 . The Does-not-equals Operator ( $!=$ )  
The production

**Deleted:** is evaluated as follows:

*EqualityExpression : EqualityExpression === RelationalExpression*

1. Let  $lref$  be the result of evaluating *EqualityExpression*.
2. Let  $lval$  be  $\text{GetValue}(lref)$ .
3. If  $lval$  is an abrupt completion, return  $lval$ .
4. Let  $rref$  be the result of evaluating *RelationalExpression*.
5. Let  $rval$  be  $\text{GetValue}(rref)$ .
6. Return the result of performing the strict equality comparison algorithm  $rval === lval$ .

**Deleted:** 11.9.4 The Strict Equals Operator ( $==$ )  
The production

**Deleted:** is evaluated as follows:

*EqualityExpression : EqualityExpression !== RelationalExpression*

**Deleted:** (See 11.9.6)

**Deleted:** 11.9.5 . The Strict Does-not-equals Operator ( $!==$ )  
The production

**Deleted:** is evaluated as follows:

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be *GetValue(lref)*.
- 3. If *lval* is an abrupt completion, return *lval*.**
4. Let *rref* be the result of evaluating *RelationalExpression*.
5. Let *rval* be *GetValue(rref)*.
6. Let *r* be the result of performing strict equality comparison *algorithm rval === lval*.
7. If *r* is **true**, return **false**. Otherwise, return **true**.

**Deleted:** (See 11.9.6)

#### *EqualityExpression : EqualityExpression [no LineTerminator here] is RelationalExpression*

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be *GetValue(lref)*.
- 3. If *lval* is an abrupt completion, return *lval*.**
4. Let *rref* be the result of evaluating *RelationalExpression*.
5. Let *rval* be *GetValue(rref)*.
- 6. If *rval* is an abrupt completion, return *rval*.**
7. Return the result of performing *SameValue(rval, lval)*.

**Deleted:** v

#### *EqualityExpression : EqualityExpression [no LineTerminator here] isn't RelationalExpression*

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be *GetValue(lref)*.
- 3. If *lval* is an abrupt completion, return *lval*.**
4. Let *rref* be the result of evaluating *RelationalExpression*.
5. Let *rval* be *GetValue(rref)*.
6. Let *r* be the result of performing *SameValue(rval, lval)*.
7. If *r* is **true**, return **false**. Otherwise, return **true**.

**Deleted:** v

## 11.10 Binary Bitwise Operators

### Syntax

*BitwiseANDExpression :*  
*EqualityExpression*  
*BitwiseANDExpression & EqualityExpression*

*BitwiseANDExpressionNoIn :*  
*EqualityExpressionNoIn*  
*BitwiseANDExpressionNoIn & EqualityExpressionNoIn*

*BitwiseXORExpression :*  
*BitwiseANDExpression*  
*BitwiseXORExpression ^ BitwiseANDExpression*

*BitwiseXORExpressionNoIn :*  
*BitwiseANDExpressionNoIn*  
*BitwiseXORExpressionNoIn ^ BitwiseANDExpressionNoIn*

*BitwiseORExpression :*  
*BitwiseXORExpression*  
*BitwiseORExpression | BitwiseXORExpression*

*BitwiseORExpressionNoIn :*  
*BitwiseXORExpressionNoIn*  
*BitwiseORExpressionNoIn | BitwiseXORExpressionNoIn*

### Runtime Semantics: Evaluation

The production *A : A @ B*, where *@* is one of the bitwise operators in the productions above, is evaluated as follows:

**Deleted:** Semantics¶

1. Let *lref* be the result of evaluating *A*.
2. Let *lval* be GetValue(*lref*).
- 3. If *lval* is an abrupt completion, return *lval*.**
4. Let *rref* be the result of evaluating *B*.
5. Let *rval* be GetValue(*rref*).
- 6. If *rval* is an abrupt completion, return *rval*.**
7. Let *lnum* beToInt32(*lval*).
- 8. If *lnum* is an abrupt completion, return *lnum*.**
9. Let *rnum* beToInt32(*rval*).
- 10. If *rnum* is an abrupt completion, return *rnum*.**
11. Return the result of applying the bitwise operator @ to *lnum* and *rnum*. The result is a signed 32 bit integer.

## 11.11 Binary Logical Operators

### Syntax

*LogicalANDExpression* :  
 BitwiseORExpression  
*LogicalANDExpression* && BitwiseORExpression

*LogicalANDExpressionNoIn* :  
 BitwiseORExpressionNoIn  
*LogicalANDExpressionNoIn* && BitwiseORExpressionNoIn

*LogicalORExpression* :  
*LogicalANDExpression*  
*LogicalORExpression* || *LogicalANDExpression*

*LogicalORExpressionNoIn* :  
*LogicalANDExpressionNoIn*  
*LogicalORExpressionNoIn* || *LogicalANDExpressionNoIn*

The semantics of the *LogicalANDExpressionNoIn* and *LogicalORExpressionNoIn* productions are the same manner as the *LogicalANDExpression* and *LogicalORExpression* productions except that the contained *LogicalANDExpressionNoIn*, *BitwiseORExpressionNoIn* and *LogicalORExpressionNoIn* are used in place of the contained *LogicalANDExpression*, *BitwiseORExpression* and *LogicalORExpression*, respectively.

NOTE The value produced by a && or || operator is not necessarily of type Boolean. The value produced will always be the value of one of the two operand expressions.

### Runtime Semantics: Evaluation

*LogicalANDExpression* : *LogicalANDExpression* && BitwiseORExpression

1. Let *lref* be the result of evaluating *LogicalANDExpression*.
2. Let *lval* be ToBoolean(GetValue(*lref*)).
- 3. If *lval* is an abrupt completion, return *lval*.**
4. If *lval* is false, return *lval*.
5. Let *rref* be the result of evaluating *BitwiseORExpression*.
6. Return GetValue(*rref*).

*LogicalORExpression* : *LogicalORExpression* || *LogicalANDExpression*

1. Let *lref* be the result of evaluating *LogicalORExpression*.
2. Let *lval* be ToBoolean(GetValue(*lref*)).
- 3. If *lval* is an abrupt completion, return *lval*.**
4. If *lval* is true, return *lval*.
5. Let *rref* be the result of evaluating *LogicalANDExpression*.
6. Return GetValue(*rref*).

**Deleted:** evaluated in

**Deleted:** evaluated instead

**Deleted:** Semantics||  
The production

**Deleted:** is evaluated as follows:

**Deleted:** ToBoolean(*lval*)

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** ToBoolean(*lval*)

## 11.12 Conditional Operator ( ?: )

### Syntax

*ConditionalExpression* :

*LogicalORExpression*  
*LogicalORExpression* ? *AssignmentExpression* : *AssignmentExpression*

*ConditionalExpressionNoIn* :

*LogicalORExpressionNoIn*  
*LogicalORExpressionNoIn* ? *AssignmentExpression* : *AssignmentExpressionNoIn*

The semantics of the *ConditionalExpressionNoIn* production is the same as the *ConditionalExpression* production except that the contained *LogicalORExpressionNoIn*, *AssignmentExpression* and *AssignmentExpressionNoIn* are used in place of the contained *LogicalORExpression*, first *AssignmentExpression* and second *AssignmentExpression*, respectively.

NOTE The grammar for a *ConditionalExpression* in ECMAScript is a little bit different from that in C and Java, which each allow the second subexpression to be an Expression but restrict the third expression to be a *ConditionalExpression*. The motivation for this difference in ECMAScript is to allow an assignment expression to be governed by either arm of a conditional and to eliminate the confusing and fairly useless case of a comma expression as the centre expression.

### Runtime Semantics: Evaluation

*ConditionalExpression* : *LogicalORExpression* ? *AssignmentExpression* : *AssignmentExpression*

1. Let *lref* be the result of evaluating *LogicalORExpression*.
2. Let *lval* be *ToBoolean(GetValue(lref))*.
3. If *lval* is an abrupt completion, return *lval*.
4. If *lval* is true, then
  - a. Let *trueRef* be the result of evaluating the first *AssignmentExpression*.
  - b. Return *GetValue(trueRef)*.
5. Else
  - a. Let *falseRef* be the result of evaluating the second *AssignmentExpression*.
  - b. Return *GetValue(falseRef)*.

## 11.13 Assignment Operators

### Syntax

*AssignmentExpression* :

- ConditionalExpression*
- YieldExpression*
- ArrowFunction*

*LeftHandSideExpression* = *AssignmentExpression*  
*LeftHandSideExpression* *AssignmentOperator* *AssignmentExpression*

*AssignmentExpressionNoIn* :

- ConditionalExpressionNoIn*
- YieldExpression*
- ArrowFunction*

*LeftHandSideExpression* = *AssignmentExpressionNoIn*  
*LeftHandSideExpression* *AssignmentOperator* *AssignmentExpressionNoIn*

*AssignmentOperator* : one of

\*= /= %= += -= <=> >>= &= ^= |=

The semantics of the *AssignmentExpressionNoIn* productions are the same manner as the *AssignmentExpression* productions except that the contained *ConditionalExpressionNoIn* and *AssignmentExpressionNoIn* are used in place of the contained *ConditionalExpression* and *AssignmentExpression*, respectively.

**Deleted:** The *LogicalANDExpressionNoIn* and *LogicalORExpressionNoIn* productions are evaluated in the same manner as the *LogicalANDExpression* and *LogicalORExpression* productions except that the contained *LogicalANDExpressionNoIn*, *BitwiseORExpressionNoIn* and *LogicalORExpressionNoIn* are evaluated instead of the contained *LogicalANDExpression*, *BitwiseORExpression* and *LogicalORExpression*, respectively.  
**NOTE** The value produced by a **&&** or **||** operator is not necessarily of type Boolean. The value produced will always be the value of one of the two operand expressions.¶

**Formatted:** Not Small caps, All caps

**Deleted:** evaluated in

**Deleted:** manner

**Deleted:** evaluated instead

## Static Semantics

### Static Semantics: Early Errors

AssignmentExpression :

LeftHandSideExpression = AssignmentExpression  
LeftHandSideExpression AssignmentOperator AssignmentExpression

- It is a Syntax Error if the AssignmentExpression is contained in strict code and LeftHandSideExpression is the Identifier eval or the Identifier arguments.
- It is a Syntax Error if the LeftHandSideExpression is an Identifier that statically resolves to a declarative environment record binding and the resolved binding is an immutable binding.
- It is a Syntax Error if the LeftHandSideExpression is PrimaryExpression : ( Expression ) and Expression derived a production that would produce a Syntax Error according to these rules. This rule is recursively applied.

AssignmentExpression : LeftHandSideExpression = AssignmentExpression

- It is a Syntax Error if IsInvalidAssignmentPattern of LeftHandSideExpression is true.

## Runtime Semantics

### Runtime Semantics: Evaluation

AssignmentExpression : LeftHandSideExpression = AssignmentExpression

1. If LeftHandSideExpression is neither an ObjectLiteral nor an ArrayLiteral, then
  - a. Let lref be the result of evaluating LeftHandSideExpression.
  - b. If lref is an abrupt completion, return lref.
  - c. Let rref be the result of evaluating AssignmentExpression.
  - d. Let rval be GetValue(rref).
  - e. If the result of PutValue(lref, rval) is an abrupt completion, return that result.
  - f. Return rval.
2. Let AssignmentPattern be the parse of the source code corresponding to LeftHandSideExpression using AssignmentPattern as the goal symbol.
3. Let rref be the result of evaluating AssignmentExpression.
4. Let rval be ToObject(GetValue(rref)).
5. If rval is an abrupt completion, return rval.
6. Let status be the result of performing Destructuring Assignment Evaluation of AssignmentPattern using rval as the argument.
7. If status is an abrupt completion, return rval.
8. Return rval.

AssignmentExpression : LeftHandSideExpression AssignmentOperator AssignmentExpression

1. Let lref be the result of evaluating LeftHandSideExpression.
2. Let lval be GetValue(lref).
3. If lval is an abrupt completion, return lval.
4. Let rref be the result of evaluating AssignmentExpression.
5. Let rval be GetValue(rref).
6. If rval is an abrupt completion, return rval.
7. Let operator be the @ where AssignmentOperator is @=
8. Let r be the result of applying operator @ to lval and rval.
9. If the result of PutValue(lref, r) is an abrupt completion, return that result.
10. Return r.

NOTE When an assignment occurs within strict mode code, it is an runtime error if lref in step 1 e of the first algorithm or step 9 of the second algorithm is an unresolved reference. If it is, a ReferenceError exception is thrown. The LeftHandSide also may not be a reference to a data property with the attribute value {[[Writable]]:false}, to an accessor

**Comment [AW31]:** Note that a rest element needs to be added to ArrayLiterals

**Deleted:** Supplemental Syntax¶  
 In certain circumstances when processing the production AssignmentExpression :  
LeftHandSideExpression = AssignmentExpression the following grammar is used to refine the interpretation of LeftHandSideExpression.¶  
AssignmentPattern :¶  
ObjectAssignmentPattern  
ArrayAssignmentPattern ¶  
ObjectAssignmentPattern :¶  
{ } .  
{ AssignmentPropertyList }  
{ AssignmentPropertyList , } ¶  
ArrayAssignmentPattern :¶  
[ Elision<sub>opt</sub> AssignmentRestElement<sub>opt</sub> ]  
[ AssignmentElementList , Elision<sub>opt</sub> AssignmentRestElement<sub>opt</sub> ] ¶  
AssignmentRestElement<sub>opt</sub> J ¶

**Deleted:** <#>Semantics¶

**Formatted**

**Deleted:** <#>It is a Syntax Error if the  
**Deleted:** <#>It is a Syntax Error if the  
**Deleted:** the AssignmentExpression is contained  
**Deleted:** does not  
**Deleted:** or  
**Deleted:** if  
**Formatted**  
**Deleted:** the AssignmentExpression is contained  
**Deleted:** the  
**Deleted:** an ObjectLiteral or an ArrayLiteral a  
**Deleted:** The production  
**Deleted:** is evaluated as follows:  
**Formatted:** Font: Not Bold  
**Deleted:** <#>Throw a SyntaxError exception i  
**Deleted:** Call  
**Deleted:** <#>If this is not extended code, throw  
**Deleted:** P  
**Deleted:** Evaluate  
**Deleted:** obj parameter  
**Deleted:**  
**Deleted:** <#>Throw a SyntaxError exception i  
**Deleted:** Call  
**Deleted:** ¶  
**Deleted:** Formatted: Note  
**Deleted:** its LeftHandSide must not evaluate to  
**Deleted:** d  
**Deleted:**  
**Deleted:** must not be  
**Deleted:** an  
**Deleted:** does  
**Deleted:** upon assignment

property with the attribute value `{[[Set]]:undefined}`, nor to a non-existent property of an object whose `[[Extensible]]` internal property has the value `false`. In these cases a `TypeError` exception is thrown.

**Deleted:** ¶

**Deleted:** .1

### 11.13.1. Destructuring Assignment

#### Supplemental Syntax

In certain circumstances when processing the production `AssignmentExpression : LeftHandSideExpression = AssignmentExpression` the following grammar is used to refine the interpretation of `LeftHandSideExpression`.

##### AssignmentPattern :

ObjectAssignmentPattern  
ArrayAssignmentPattern

##### ObjectAssignmentPattern :

{ }  
{ AssignmentPropertyList }  
{ AssignmentPropertyList , }

##### ArrayAssignmentPattern :

[ Elision<sub>opt</sub> AssignmentRestElement<sub>opt</sub> ]  
[ AssignmentElementList ]  
[ AssignmentElementList , Elision<sub>opt</sub> AssignmentRestElement<sub>opt</sub> ]

##### AssignmentPropertyList :

AssignmentProperty  
AssignmentPropertyList , AssignmentProperty

##### AssignmentElementList :

Elision<sub>opt</sub> AssignmentElement  
AssignmentElementList , Elision<sub>opt</sub> AssignmentElement

##### AssignmentProperty :

Identifier  
PropertyName : LeftHandSideExpression

##### AssignmentElement :

LeftHandSideExpression

##### AssignmentRestElement :

... LeftHandSideExpression

#### Static Semantics

##### Static Semantics: Early Errors

###### AssignmentProperty : PropertyName : LeftHandSideExpression

###### AssignmentElement : LeftHandSideExpression

- It is a Syntax Error if `LeftHandSideExpression` is the Identifier `eval` or the Identifier `arguments`.
- It is a Syntax Error if `LeftHandSideExpression` is the Identifier `this` or the Identifier `super`.
- It is a Syntax Error if the `LeftHandSideExpression` is a Literal, a FunctionExpression or a ClassExpression.
- It is a Syntax Error if the `LeftHandSideExpression` is an Identifier that does not statically resolve to a declarative environment record binding or if the resolved binding is an immutable binding.
- It is a Syntax Error if `IsInvalidAssignmentPattern` of `LeftHandSideExpression` is true.
- It is a Syntax Error if the `LeftHandSideExpression` is PrimaryExpression : ( Expression ) and `Expression` derived a production that would produce a Syntax Error according to these rules. This rule is recursively applied.

**Formatted:** Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

**Comment [AWB632]:** This list need to be updated when the grammar is stable

**Comment [AW33]:** This part probably doesn't need to be here if 11.1.2 has this as a static semantic for extended code.

**Deleted:** It is a Syntax Error if the `LeftHandSideExpression` is an ObjectLiteral or an ArrayLiteral and the source code corresponding to `LeftHandSideExpression` using cannot be parsed using `AssignmentPattern` as the goal symbol.

### AssignmentRestElement : ... LeftHandSideExpression

- It is a Syntax Error if *LeftHandSideExpression* is the Identifier `eval` or the Identifier `arguments`.
- It is a Syntax Error if *LeftHandSideExpression* is the Identifier `this` or the Identifier `super`.
- It is a Syntax Error if the *LeftHandSideExpression* is a *Literal*, an *ObjectLiteral*, an *ArrayLiteral*, a *FunctionExpression*, or a *ClassExpression*.
- It is a Syntax Error if the *LeftHandSideExpression* is an *Identifier* that does not statically resolve to a declarative environment record binding or if the resolved binding an immutable binding.
- It is a Syntax Error if the *LeftHandSideExpression* is *PrimaryExpression* : *( Expression )* and *Expression* derived a production that would produce a Syntax Error according to these rules. This rule is recursively applied.

### Runtime Semantics

#### Runtime Semantics: Destructuring Assignment Evaluation

with parameter *obj*

#### AssignmentPattern : ObjectAssignmentPattern

1. Return the result of performing Destructuring Assignment Evaluation of *ObjectAssignmentPattern* with *obj* as the argument.

#### AssignmentPattern : ArrayAssignmentPattern

1. Return the result of performing Destructuring Assignment Evaluation of *ArrayAssignmentPattern* with *obj* as the argument.

#### ObjectAssignmentPattern : { }

and

#### ArrayAssignmentPattern :

[1]  
[Elision]

1. Return.

#### ObjectAssignmentPattern :

{ *AssignmentPropertyList* }  
{ *AssignmentPropertyList* , }

1. Return the result of performing Destructuring Assignment Evaluation of *AssignmentPropertyList* with *obj* as the argument.

#### AssignmentPropertyList : AssignmentProperty

1. Return the result of performing Destructuring Assignment Evaluation of *AssignmentProperty* with *obj* as the argument.

#### AssignmentPropertyList : AssignmentPropertyList , AssignmentProperty

1. Let *status* be the result of performing Destructuring Assignment Evaluation of *AssignmentPropertyList* with *obj* as the argument.
2. If *status* is an abrupt completion, return *status*.
3. Return the result of performing Destructuring Assignment Evaluation of *AssignmentProperty* with *obj* as the argument.

#### AssignmentProperty : Identifier

#### **Formatted**

**Comment [AWB634]:** This list need to be

**Deleted:** The supplemental production

**Deleted:** is evaluated with the parameter *obj*

**Deleted:** Perform

**Deleted:** using

**Deleted:** *obj* parameter

**Deleted:** ¶

**Deleted:** The supplemental production

**Deleted:**

**Deleted:** is evaluated with the parameter *obj*

**Deleted:** Perform

**Deleted:** Evaluate

**Deleted:** using

**Deleted:** *obj*

**Deleted:** parameter

**Deleted:** ¶

**Deleted:** The supplemental production

**Deleted:** ,the production

**Deleted:** The supplemental productions

**Deleted:** Perform

**Deleted:** Evaluate

**Deleted:** using

**Deleted:** *obj*

**Deleted:** parameter

**Deleted:** ¶

**Deleted:** The supplemental production

**Deleted:** is evaluated with the parameter *obj*

**Deleted:** Perform

**Deleted:** Evaluate

**Deleted:** using

**Deleted:** *obj*

**Deleted:** parameter

**Deleted:** ¶

**Deleted:** The supplemental production

**Deleted:** is evaluated with the parameter *obj*

**Deleted:** Perform

**Deleted:** Evaluate

**Deleted:** using

**Deleted:** *obj*

**Deleted:** parameter

**Deleted:** ¶

**Deleted:** The supplemental production

**Deleted:** is evaluated with the parameter *obj*

**Deleted:** Perform

**Deleted:** Evaluate

**Deleted:** using

**Deleted:** *obj*

**Deleted:** parameter

**Deleted:** The supplemental production

1. Let  $v$  be the result of calling the  $[[\text{Get}]]$  internal method of  $obj$  with the  $\text{Identifier}$  string as the argument.
2. Let  $lref$  be the result of performing Identifier Resolution(10.3.1) with the  $\text{IdentifierName}$  corresponding to  $\text{Identifier}$ .
3. Return  $\text{PutValue}(lref, v)$ .

#### AssignmentProperty : $\text{PropertyName} : \text{LeftHandSideExpression}$

1. Let  $\text{name}$  be the result of evaluating  $\text{PropertyName}$ .
2. Let  $v$  be the result of calling the  $[[\text{Get}]]$  internal method of  $obj$  with  $\text{name}$  as the argument.
3. If  $v$  is an abrupt completion, return  $v$ .
4. If  $\text{LeftHandSideExpression}$  is an  $\text{ObjectLiteral}$  or an  $\text{ArrayLiteral}$  then
  - a. Let  $\text{AssignmentPattern}$  be the parse of the source code corresponding to  $\text{LeftHandSideExpression}$  using  $\text{AssignmentPattern}$  as the goal symbol
  - b. Let  $vObj$  be  $\text{ToObject}(v)$ .
  - c. Return  $\text{IfAbrupt}(vObj)$ .
  - d. Return the result of evaluating  $\text{AssignmentPattern}$  using  $vObj$  as the  $obj$  parameter.
5. Let  $lref$  be the result of evaluating  $\text{LeftHandSideExpression}$ .
6. Call  $\text{PutValue}(lref, v)$ .

#### ArrayAssignmentPattern : $[\ Elision_{\text{opt}}, \text{AssignmentRestElement} ]$

1. If  $\text{Elision}$  is present, then let  $\text{skip}$  be the result of evaluating  $\text{Elision}$ , otherwise let  $\text{skip}$  be 0.
2. Return the result of performing Indexed Destructuring Assignment Evaluation of  $\text{AssignmentRestElement}$  with  $obj$  and  $\text{skip}$  as the arguments.

#### ArrayAssignmentPattern : $[\ AssignmentElementList \ ]$

1. Return the result of performing Indexed Destructuring Assignment Evaluation of  $\text{AssignmentElementList}$  using  $obj$  and 0 as the arguments.

#### ArrayAssignmentPattern : $[\ AssignmentElementList \ , \ Elision_{\text{opt}}, \text{AssignmentRestElement}_{\text{opt}} ]$

1. Let  $\text{lastIndex}$  be the result of performing Indexed Destructuring Assignment Evaluation of  $\text{AssignmentElementList}$  using  $obj$  and 0 as the arguments.
2. If  $\text{lastIndex}$  is an abrupt completion, return  $\text{lastIndex}$ .
3. If  $\text{Elision}$  is present, then let  $\text{skip}$  be the result of evaluating  $\text{Elision}$ , otherwise let  $\text{skip}$  be 0.
4. If  $\text{AssignmentRestElement}$  is present, then return the result of performing Indexed Destructuring Assignment Evaluation of  $\text{AssignmentRestElement}$  with  $obj$  and  $\text{lastIndex}+\text{skip}$  as the arguments.
5. Return  $\text{lastIndex}$ .

#### Runtime Semantics: Indexed Destructuring Assignment Evaluation

with parameters  $obj$  and  $index$

#### $\text{AssignmentElementList} : Elision_{\text{opt}}, \text{AssignmentElement}$

1. If  $\text{Elision}$  is present, then let  $\text{skip}$  be the result of evaluating  $\text{Elision}$ , otherwise let  $\text{skip}$  be 0.
2. Let  $\text{status}$  be the result of performing Indexed Destructuring Assignment Evaluation of  $\text{AssignmentElement}$  with  $obj$  and  $index+\text{skip}$  as the arguments.
3. If  $\text{status}$  is an abrupt completion, return  $\text{status}$ .
4. Return  $index+\text{skip}+1$ .

#### $\text{AssignmentElementList} : AssignmentElementList \ , \ Elision_{\text{opt}}, \text{AssignmentElement}$

1. Let  $\text{listNext}$  be the result of performing Indexed Destructuring Assignment Evaluation of  $\text{AssignmentElementList}$  using  $obj$  as the  $obj$  parameter and  $index$  as the  $index$  parameter
2. If  $\text{Elision}$  is present, then let  $\text{skip}$  be the result of evaluating  $\text{Elision}$ , otherwise let  $\text{skip}$  be 0.
3. If  $\text{listNext}$  is an abrupt completion, return  $\text{listNext}$ .

<b>Deleted:</b> is evaluated with the parameter $obj$ as follows
<b>Deleted:</b> using
<b>Deleted:</b> Call
<b>Deleted:</b> 1
<b>Deleted:</b> The static semantics of the supplemental production
<i>AssignmentProperty : <math>\text{PropertyName} : \text{LeftHandSideExpression}</math> and the supplemental production</i>
<i>AssignmentElement : <math>\text{LeftHandSideExpression}</math> and the supplemental production</i>
<i>AssignmentRestElement : ... <math>\text{LeftHandSideExpression}</math> are: &lt;#&gt; It is a Syntax Error if <math>\text{LeftHandSideExpression}</math> is the Identifier <b>eval</b> or the Identifier <b>arguments</b>. 1</i>
<b>Deleted:</b> is evaluated with the parameter $obj$ as follows
<b>Deleted:</b> passing...with
<b>Deleted:</b> If $vObj$ is an abrupt completion, return
<b>Deleted:</b> Evaluate...return the result of evaluating
<b>Deleted:</b> 1
<b>Deleted:</b> The supplemental production
<b>Deleted:</b>
<b>Deleted:</b> is evaluated with the parameter $obj$ as follows
<b>Deleted:</b> Perform
<b>Deleted:</b> Evaluate ... $\text{AssignmentRestElement}$ with $obj$ using
<b>Deleted:</b> 0
<b>Deleted:</b> index ... $\text{argumentsparameter}$ ...1
<b>Deleted:</b> <#>Perform Indexed Destructuring Assignment
<b>Deleted:</b> The supplemental production ..
<b>Deleted:</b> $\text{AssignmentPropertyList}$
<b>Deleted:</b>
<b>Deleted:</b> <#>is evaluated with the parameter $obj$ as
<b>Deleted:</b> index
<b>Deleted:</b> evaluating ... $\text{AssignmentElementList}$ using $obj$ as
<b>Deleted:</b> perform
<b>Deleted:</b> evaluate ... $\text{AssignmentRestElement}$ with using ...
<b>Deleted:</b> i
<b>Deleted:</b> index ... $\text{argumentsparameter}$ ...1
<b>Deleted:</b> The supplemental production
<b>Deleted:</b> is evaluated with the parameters $obj$ and $index$
<b>Deleted:</b> Perform
<b>Deleted:</b> Evaluate
<b>Deleted:</b> Let $index$ be the result of evaluating
<b>Deleted:</b> using ... $bj$ as the $obj$ parameter ...nd $index+skip$
<b>Deleted:</b> +skip
<b>Deleted:</b> 1
<b>Deleted:</b> The supplemental production
<b>Deleted:</b> is evaluated with the parameters $obj$ and $index$
<b>Deleted:</b> e... $listNext$
<b>Deleted:</b> evaluating
<b>Deleted:</b> tRes

4. Let *status* be the result of performing Indexed Destructuring Assignment Evaluation of *AssignmentElement* with *obj* and *listNext+skip* as the arguments.
5. If *status* is an abrupt completion, return *status*.
6. Return *listNext+skip+1*.

#### *AssignmentElement : LeftHandSideExpression*

1. Let *name* be *ToString(index)*.
2. Let *v* be the result of calling the [[Get]] internal method of *obj* with *name* as the argument.
3. If *v* is an abrupt completion, return *v*.
4. If *LeftHandSideExpression* is an *ObjectLiteral* or an *ArrayLiteral* then
  - a. Let *AssignmentPattern* be the parse of the source code corresponding to *LeftHandSideExpression* using *AssignmentPattern* as the goal symbol.
  - b. Let *vObj* be *ToObject(v)*.
  - c. ReturnIfAbrupt(*vObj*).
  - d. Return the result of evaluating Destructuring Assignment Evaluation of *AssignmentPattern* with *vObj* as the argument.
5. Let *lref* be the result of evaluating *LeftHandSideExpression*.
6. Return *PutValue(lref,v)*.

#### *AssignmentRestElement : ... LeftHandSideExpression*

1. Let *lref* be the result of evaluating *LeftHandSideExpression*.
2. If *lref* is an abrupt completion, return *lref*.
3. Let *lenVal* be the result of calling the [[Get]] internal method of *obj* with argument "length".
4. Let *len* be *ToUint32(lenVal)*.
5. ReturnIfAbrupt(*len*).
6. Let *A* be the result of the abstraction operation *ArrayCreate* (15.4) with argument 0.
7. Let *n=0*:
8. Repeat, while *index < len*
  - a. Let *P* be *ToString(index)*.
  - b. Let *exists* be the result of calling the [[HasProperty]] internal method of *obj* with argument *P*.
  - c. If *exists* is true, then
    - i. Let *v* be the result of calling the [[Get]] internal method of *obj* passing *ToString(index)* as the argument.
    - ii. ReturnIfAbrupt(*len*).
    - iii. Call the [[DefineOwnProperty]] internal method of *A* with arguments *ToString(n)*, *Property Descriptor* {[[Value]]: *v*, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}, and false.
  - d. Let *n = n+1*.
  - e. Let *index = index+1*.
9. Return *PutValue(lref,A)*.

## 11.14 Comma Operator ( , )

### Syntax

*Expression* :  
*AssignmentExpression*  
*Expression , AssignmentExpression*

*ExpressionNoIn* :  
*AssignmentExpressionNoIn*  
*ExpressionNoIn , AssignmentExpressionNoIn*

The semantics of the *ExpressionNoIn* production is the same manner as the *Expression* production except that the contained *ExpressionNoIn* and *AssignmentExpressionNoIn* are used in place of the contained *Expression* and *AssignmentExpression*, respectively.

<b>Deleted:</b> Let <i>index</i> be the result of evaluating
<b>Deleted:</b> Perform
<b>Deleted:</b> Evaluate
<b>Deleted:</b> using
<b>Deleted:</b> as the <i>obj</i> parameter
<b>Deleted:</b> <i>e</i>
<b>Deleted:</b> <i>nd</i>
<b>Deleted:</b> <i>index</i>
<b>Deleted:</b> parameter
<b>Deleted:</b> <i>i</i>
<b>Deleted:</b> <i>ndx</i>
<b>Deleted:</b> $\uparrow$
<b>Deleted:</b> The supplemental production
<b>Deleted:</b> is evaluated with the parameters <i>a</i> and <i>index</i> as follows: $\uparrow$
<b>Deleted:</b> passing
<b>Deleted:</b> If <i>vObj</i> is an abrupt completion, return <i>vObj</i> .
<b>Deleted:</b> Performing
<b>Deleted:</b> Evaluate
<b>Deleted:</b> using
<b>Deleted:</b> <i>obj</i> parameter
<b>Deleted:</b> $\uparrow$
<b>Deleted:</b> Return
<b>Deleted:</b> Call
<b>Deleted:</b> $\uparrow$
<b>Deleted:</b> The supplemental production
<b>Deleted:</b> is evaluated with the parameters <i>a</i> and <i>index</i> as follows: $\uparrow$
<b>Deleted:</b> <i>t</i>
<b>Deleted:</b> If <i>len</i> is an abrupt completion, return <i>lref</i> .
<b>Deleted:</b> a new array object created as if by the expression <u><i>new Array()</i></u> where <u><i>Array</i></u> is the standard built-in constructor with that name
<b>Deleted:</b> If <i>v</i> is an abrupt completion, return <i>v</i> .
<b>Deleted:</b> Call
<b>Deleted:</b> $\uparrow$

<b>Deleted:</b> evaluated in
<b>Deleted:</b> evaluated instead

## Runtime Semantics: Evaluation

Expression : Expression , AssignmentExpression

1. Let *lref* be the result of evaluating *Expression*.
2. If the result of GetValue(*lref*) is an abrupt completion, return that result.
3. Let *rref* be the result of evaluating *AssignmentExpression*.
4. Return GetValue(*rref*).

NOTE     GetValue must be called even though its value is not used because it may have observable side-effects.

**Deleted:** Semantics¶

The production

**Deleted:** is evaluated as follows:

**Deleted:** Call

**Deleted:** .

## 12 Statements and Declarations

### Syntax

Statement :

BlockStatement  
VariableStatement  
EmptyStatement  
ExpressionStatement  
IfStatement  
BreakableStatement  
ContinueStatement  
BreakStatement  
ReturnStatement  
WithStatement  
LabelledStatement  
ThrowStatement  
TryStatement  
DebuggerStatement

**Deleted:** IterationStatement

Declaration :

FunctionDeclaration  
LexicalDeclaration

**Deleted:**  
SwitchStatement

### Static Semantics

#### Static Semantics: BoundNames

The BoundNames of the Declaration productions are determined as follows:

1. Return the BoundNames of the single non-terminal symbol that is the right hand side of the production.

**Deleted:** LetDeclaration ..  
*Const*

#### Static Semantics: VarDeclaredNames

Statement :

EmptyStatement  
ExpressionStatement  
ContinueStatement  
BreakStatement  
ReturnStatement  
ThrowStatement  
DebuggerStatement

**Deleted:** A Statement can be part of a LabelledStatement which itself can be part of a LabelledStatement, and so on. The labels introduced this way are collectively referred to as the "current label set" when describing the semantics of individual statements. A LabelledStatement has no semantic meaning other than the introduction of a label set. The label set of an IterationStatement or a SwitchStatement initially contains the single element em. The label set of any other statement is initially empty. ¶

1. Return a new empty List.

**Deleted:** The VarDeclaredNames of the productions:  
*Statement : EmptyStatement ..*  
*Statement : ExpressionStatement ..*  
*Statement : ContinueStatement ..*  
*Statement : BreakStatement ..*  
*Statement : ReturnStatement ..*  
*Statement : ThrowStatement ..*  
*Statement : DebuggerStatement ..*  
is determined as follows: ¶

Statement :

BlockStatement  
VariableStatement  
IterationStatement  
WithStatement  
LabelledStatement  
SwitchStatement  
TryStatement

1. Return the VarDeclaredNames of the single non-terminal symbol that is the right hand side of the production.

Statement : VariableStatement

1. Return the BoundNames of *VariableStatement*.

**Runtime Semantics**

**Runtime Semantics: Labelled Evaluation**

With argument *labelSet*.

BreakableStatement : IterationStatement

1. Let *stmtResult* be the result performing Labelled Evaluation of *IterationStatement* wth argument *labelSet*.
2. If *stmtResult*.[[type]] is break and *stmtResult*.[[target]] is empty, then
  - a. Let *stmtResult* be NormalCompletion(*stmtResult*.[[value]])
3. Return *stmtResult*.

BreakableStatement : SwitchStatement

3. Let *stmtResult* be the result of evaluating *SwitchStatement*.
4. If *stmtResult*.[[type]] is break and *stmtResult*.[[target]] is empty, then
  - a. Let *stmtResult* be NormalCompletion(*stmtResult*.[[value]])
5. Return *stmtResult*.

**NOTE** A *BreakableStatement* is one that can be existed via an unlabelled *BreakStatement*.

**Runtime Semantics: Evaluation**

The *Statement* productions are all evaluated as follows

1. Return the result of evaluating the single non-terminal symbol that is the right hand side of the production.

The *Declaration* productions are all evaluated as follows

1. Return the result of evaluating the single non-terminal symbol that is the right hand side of the production.

**NOTE** The result of evaluating a *Statement* or *Declaration* is always a Completion value.

BreakableStatement :  
IterationStatement  
SwitchStatement

1. Let *newLabelSet* be a new empty List.
2. Return the result of performing Labelled Evaluation of this *BreakableStatement* wth argument *newLabelSet*.

**Deleted:** The VarDeclaredNames of the productions:

*Statement : BlockStatement* ..  
*Statement : IfStatement*  
*Statement : IterationStatement* ..  
*Statement : WithStatement*  
*Statement : LabelledStatement* ..  
*Statement : SwitchStatement*  
*Statement : TryStatement* ..

is determined as follows:<sup>11</sup>

**Deleted:** The VarDeclaredNames of the production

**Deleted:** is determined as follows:

**Deleted:** as

**Deleted:** as

**Comment [AWB 335]:** Need to have an evalution semantics in section 13 for *FunctionDeclaration*

## 12.1 Block

### Syntax

BlockStatement :  
Block

Block :  
{ StatementList<sub>opt</sub> }

StatementList :  
StatementListItem  
StatementList StatementListItem

StatementListItem :  
Statement  
Declaration

### Static Semantics

#### Static Semantics: Early Errors

Block : { StatementList }

- It is a Syntax Error if the LexicallyDeclaredNames of StatementList contains any duplicate entries.
- It is a Syntax Error if any element of the LexicallyDeclaredNames of StatementList also occurs in the VarDeclaredNames of StatementList.

#### Static Semantics: LexicalDeclarations

StatementList : StatementListItem

1. Return the LexicalDeclarations of StatementListItem.

StatementList : StatementList StatementListItem

1. Let declarations be LexicalDeclarations of StatementList.
2. Append to names the elements of the LexicalDeclarations of StatementListItem.
3. Return declarations.

StatementListItem : Statement

1. Return a new empty List.

StatementListItem : Declaration

1. Return return a new List containing Declaration.

#### Static Semantics: LexicallyDeclaredNames

Block : {}

1. Return a new empty List.

Block : { StatementList }

1. Return the LexicallyDeclaredNames of StatementList.

StatementList : StatementListItem

**Deleted:** The BoundNames of the Declaration production are determined as follows:  
<#>Return the BoundNames of the single non-terminal symbol that is the right hand side of the production.  
The Declaration productions are all evaluated as as follows:  
<#>Return the result of evaluating the single non-terminal symbol that is the right hand side of the production.

**Deleted:** NOTE Several widely used implementations of ECMAScript are known to support the use of FunctionDeclaration as a Statement. However there are significant and irreconcilable variations among the implementations in the semantics applied to such FunctionDeclarations. Because of these irreconcilable differences, the use of a FunctionDeclaration as a Statement results in code that is not reliably portable among implementations. It is recommended that ECMAScript implementations either disallow this usage of FunctionDeclaration or issue a warning when such a usage is encountered. Future editions of ECMAScript may define alternative portable means for declaring functions in a Statement context.

**Deleted:** <#>It is a Syntax Error if StatementList includes a StatementListItem : Declaration production whose Declaration is a Declaration : FunctionDeclaration production and the source code matching this Block production is not contained in extended code.

**Formatted:** Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

**Deleted:** The VarDeclaredNames of the production BlockStatement : Block is determined as follows:  
<#>Return the VarDeclaredNames of Block.  
The production BlockStatement : Block is evaluated as follows:  
<#>Return the result of evaluating Block.

**Deleted:** The LexicallyDeclaredNames of the production

**Deleted:** is determined as follows:

**Deleted:** Item

1. Return the LexicallyDeclaredNames of *StatementListItem*.

*StatementList* : *StatementList StatementListItem*

1. Let *names* be LexicallyDeclaredNames of *StatementList*.
2. Append to *names* the elements of the LexicallyDeclaredNames of *StatementListItem*.
3. Return *names*.

*StatementListItem* : *Statement*

1. Return a new empty List.

*StatementListItem* : *Declaration*

1. Return the BoundNames of *Declaration*.

#### **Static Semantics: VarDeclaredNames**

*BlockStatement* : *Block*

1. Return the VarDeclaredNames of *Block*.

*Block* : *{ }<sub>v</sub>*

1. Return a new empty List.

*Block* : *{ StatementList }*

1. Return the VarDeclaredNames of *StatementList*.

*StatementList* : *StatementListItem*

1. Return the VarDeclaredNames of *StatementListItem*.

*StatementList* : *StatementList StatementListItem*

1. Let *names* be VarDeclaredNames of *StatementList*.
2. Append to *names* the elements of the VarDeclaredNames of *StatementListItem*.
3. Return *names*.

*StatementListItem* : *Statement*

1. Return the VarDeclaredNames of *Statement*.

*StatementListItem* : *Declaration*

1. Return a new empty List.

#### **Runtim Semantics**

##### **Runtim Semantics: Evaluation**

*BlockStatement* : *Block*

1. Return the result of evaluating *Block*.

*Block* : *{ }<sub>v</sub>*

1. Return *NormalCompletion*(empty).

**Deleted:** The VarDeclaredNames of the production

**Deleted:** is determined as follows:

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** normal,

**Deleted:** , empty

## Block : { StatementList }

1. Let *oldEnv* be the *current execution context's LexicalEnvironment*.
2. Let *blockEnv* be the result of calling *NewDeclarativeEnvironment* passing *oldEnv* as the argument.
3. Perform *Block Declaration Instantiation* using *StatementList* and *blockEnv*.
4. Set the *current execution context's LexicalEnvironment* to *blockEnv*.
5. Let *blockValue* be the result of evaluating *StatementList*.
6. Set the *current execution context's LexicalEnvironment* to *oldEnv*.
7. Return *blockValue*.

**NOTE** No matter how control leaves the *Block* the *LexicalEnvironment* is always restored to its former state.

## StatementList : StatementListItem

1. Let *s* be the result of evaluating *StatementListItem*.
2. Return *s*.

## StatementList : StatementList StatementListItem

1. Let *s1* be the result of evaluating *StatementList*.
2. If *s1* is an abrupt completion, return *s1*.
3. Let *s* be the result of evaluating *Statement*.
4. If *s.[[type]]* is *throw*, return *s*.
5. If *s.[[value]]* is empty, let *V* = *s.[[value]]*, otherwise let *V* = *s.[[value]]*.
6. Return *Completion {[ [type] : s.[[type]], [ [value] : V, [ [target] : s.[[target]] ] ] }.*

**NOTE** Steps 4 and 5 of the above algorithm ensure that the value of a *StatementList* is the value of the last value producing *Statement* in the *StatementList*. For example, the following calls to the *eval* function all return the value 1:

```
eval("1;;;;")
eval("1;{}")
eval("1;var a;")
```

## StatementListItem : Statement

1. Return the result of evaluating *Statement*.

## StatementListItem : Declaration

1. Return the result of evaluating *Declaration*.

## **12.2 Declarations and the Variable Statement**

### **12.2.1 Let and Const Declarations**

**NOTE** A *let* and *const* declarations define variables that are scoped to the current execution context's *LexicalEnvironment*. The variables are created when their containing *Lexical Environment* is instantiated but may not be accessed in any way until the variable's *LexicalBinding* is evaluated. A variable defined by a *LexicalBinding* with an *Initialiser* is assigned the value of its *Initialiser's AssignmentExpression* when the *LetBinding* is evaluated, not when the variable is created. If a *LexicalBinding* in a *let* declaration does not have an *Initialiser* the variable is assigned the value *undefined* when the *LetBinding* is executed.

### **Syntax**

*LexicalDeclaration* :  
  *LetOrConst BindingList* ;

*LexicalDeclarationNoIn* :  
  *LetOrConst BindingListNoIn*

**Deleted:** The static semantics of the production *Block : { StatementList }* are:  
  ~~<#>It is a Syntax Error if *StatementList* includes a *StatementListItem : Declaration* production whose *Declaration* is a *Declaration : FunctionDeclaration* production and the source code matching this *Block* production is not contained in extended code.~~

~~<#>It is a Syntax Error if the LexicallyDeclaredNames of *StatementList* contains any duplicate entries.~~

~~<#>It is a Syntax Error if any element of the LexicallyDeclaredNames of *StatementList* also occurs in the VarDeclaredNames of *StatementList*.~~

~~The LexicallyDeclaredNames of the production *Block : { StatementList }* is determined as follows:~~

~~<#>Return the LexicallyDeclaredNames of *StatementList*.~~

~~The VarDeclaredNames of the production *Block : { StatementList }* is determined as follows:~~

~~<#>Return the VarDeclaredNames of *StatementList*.~~

**Deleted:** is evaluated as follows:

**Deleted:** running

**Deleted:** For each element *param*, of *formals*,

**Deleted:** running

**Deleted:** running

**Deleted:** The LexicallyDeclaredNames of the production

**Deleted:** is evaluated as follows:

**Comment [AW36]:** If evaluation of code starts

**Deleted:** ~~<#>If an exception was thrown, return (throw, *V*)~~

**Deleted:** The LexicallyDeclaredNames of the production

**Deleted:** is evaluated as follows:

**Deleted:** an exception was thrown, return (

**Deleted:** *V*, empty)

**Deleted:** where

**Deleted:** *V* is the exception. (Execution now proceeds as if

**Deleted:** value

**Deleted:** (

**Deleted:** type

**Deleted:** ).

**Deleted:** 5

**Deleted:** 6

**Deleted:** The LexicallyDeclaredNames and the

**Deleted:** is evaluated as follows:

**Deleted:** The LexicallyDeclaredNames of the production

**Deleted:** is evaluated as follows:

**Deleted:** *s*

**Deleted:** *Let*

**Deleted:** *Let*

**Deleted:** executed

**Deleted:** *LetBinding*

**Deleted:** executed

**Deleted:** *LetBinding*

**Deleted:** *Let*

**Deleted:** *let Let*

LetOrConst :

let  
const

BindingList :

LexicalBinding  
BindingList , LexicalBinding

BindingListNoIn :

LexicalBindingNoIn  
BindingListNoIn , LexicalBindingNoIn

LexicalBinding :

BindingIdentifier Initialiser<sub>opt</sub>  
BindingPattern Initialiser

LexicalBindingNoIn :

BindingIdentifier InitialiserNoIn<sub>opt</sub>  
BindingPattern InitialiserNoIn

BindingIdentifier :

Identifier

Initialiser :

= AssignmentExpression

InitialiserNoIn :

= AssignmentExpressionNoIn

The semantics of the LexicalDeclarationNoIn, BindingListNoIn, LexicalBindingNoIn and InitialiserNoIn productions are the same as the LexicalDeclaration, BindingList, LexicalBinding and Initialiser productions except that the contained BindingListNoIn, LexicalBindingNoIn, InitialiserNoIn and AssignmentExpressionNoIn are used in place of the contained BindingList, LexicalBinding, Initialiser and AssignmentExpression, respectively.

## Static Semantics

### Static Semantics: Early Errors

LexicalBinding : BindingIdentifier

- It is a Syntax Error if IsConstantDeclaration of the LexicalDeclaration containing this production is true.

BindingIdentifier : Identifier

- It is a Syntax Error if the BindingIdentifier is contained in strict code and if the Identifier is eval or arguments.

### Static Semantics: Bound Names

LexicalDeclaration : LetOrConst BindingList

- Return the Bound Names of BindingList.

BindingList : LexicalBinding

- Return the Bound Names of LexicalBinding.

BindingList : BindingList , LexicalBinding

**Deleted:** Let

**Deleted:** bnf

**Deleted:** Let

1. Let *names* be the Bound Names of *BindingList*.
2. Append to *names* the elements of the Bound Names of *LexicalBinding*.
3. Return *names*.

*LexicalBinding* : *BindingIdentifier Initialiser*<sub>opt</sub>

1. Return the Bound Names of *BindingIdentifier*.

*LexicalBinding* : *BindingPattern Initialiser*

1. Return the Bound Names of *BindingPattern*.

*BindingIdentifier* : *Identifier*

1. Return a new List containing the String Value of *Identifier*.

**Static Semantics: IsConstantDeclaration**

*LexicalDeclaration* : *LetOrConst BindingList* ;

1. Return IsConstantDeclaration of *LetOrConst*.

*LetOrConst BindingList* : **let**

1. Return **false**.

*LetOrConst BindingList* : **const**

1. Return **false**.

**Static Semantics: String Value**

*BindingIdentifier* : *Identifier*

1. Return the String Value of *Identifier*.

**Runtime Semantics**

**Runtime Semantics: Binding Initialisation**

With arguments *value* and *environment*.

**NOTE** *undefined* is passed for *environment* to indicate that a PutValue operation should be used to assign the initialisation value. This is the case for *var* statements formal parameter lists of non-strict functions. In those cases a lexical binding is hosted and preinitialized prior to evaluation of its initializer.

*BindingIdentifier* : *Identifier*

1. Return the result of performing *Binding Initialisation* for *Identifier* passing *value* and *env* as the arguments.

**Token Identifier**

1. If *environment* is not *undefined*, then

a. Return the result of calling the *InitializeBinding* concrete method of *environment* passing *Identifier* and *value* as the arguments.

2. Else
  - a. Let *lhs* be the result of evaluating *Identifier* as described in 11.1.2.
  - b. Return *PutValue(lhs, value)*.

**Deleted:** N

**Deleted:** *Let*

**Deleted:** of

**Deleted:** *Let*

**Deleted:** *Let*

**Deleted:** *Let*

**Deleted:** Runtime

**Deleted:** Initialization

**Deleted:** environment

**Deleted:** initialization

**Deleted:** the

**Deleted:** P

**Deleted:** Initialization

**Deleted:** environment

**Deleted:** C

**Deleted:** environment

**Deleted:** Call

## Runtime Semantics: Evaluation

### LexicalDeclaration : LetOrConst BindingList ;

1. Let *next* be the result of evaluating *BindingList*.
2. If *next* is an abrupt completion, return *next*.
3. Return NormalCompletion(empty).

### BindingList : LexicalBinding

1. Return the result of evaluating *LexicalBinding*.

### BindingList : BindingList , LexicalBinding

1. Let *next* be the result of evaluating *BindingList*.
2. If *next* is an abrupt completion, return *next*.
3. Return the result of evaluating *LexicalBinding*.

### LexicalBinding : BindingIdentifier

1. Let *env* be the current execution context's LexicalEnvironment.
2. Return the result of performing *BindingInitialisation* for *BindingIdentifier* passing **undefined** and *env* as the arguments.

**NOTE** A static semantic rule ensures that this form of *LexicalBinding* never occurs in a **const** declaration.

### LexicalBinding : BindingIdentifier Initialiser

2. Let *rhs* be the result of evaluating *Initialiser*.
3. Let *value* be *GetValue(rhs)*.
4. If *value* is an abrupt completion, return *rval*.
5. Let *env* be the current execution context's LexicalEnvironment.
6. Return the result of performing *BindingInitialisation* for *BindingIdentifier* passing *value* and *env* as the arguments.

### LexicalBinding : BindingPattern Initialiser

1. Let *rhs* be the result of evaluating *Initialiser*.
2. Let *rval* be *GetValue(rhs)*.
3. If *rval* is an abrupt completion, return *rval*.
4. Let *env* be the current execution context's LexicalEnvironment.
5. Return the result of performing *BindingInitialisation* for *BindingPattern* using *rval* as the *obj* parameter and *env* as the *environment* parameter.

### Initialiser : = AssignmentExpression

1. Return the result of evaluating *AssignmentExpression*.

## 12.2.2 Variable Statement

**NOTE** A **var** statement declares variables that are scoped to the current execution context's *VariableEnvironment*. **Var** variables are created when their containing *Lexical Environment* is instantiated and are initialised to **undefined** when created. Within the scope of any *VariableEnvironment* a common *Identifier* may appear in more than one *VariableDeclaration* but those declarations collective define only one variable. A variable defined by a *VariableDeclaration* with an *Initialiser* is assigned the value of its *Initialiser's AssignmentExpression* when the *VariableDeclaration* is executed, not when the variable is created.

<b>Deleted:</b> The production
<b>Deleted:</b> <i>LetDeclaration</i>
<b>Deleted:</b> <b>let</b> <i>Let</i>
<b>Deleted:</b> is evaluated as follows:
<b>Deleted:</b> Evaluate <i>Let</i>
<b>Deleted:</b> (normal,
<b>Deleted:</b> , empty)
<b>Deleted:</b> <i>Let</i>
<b>Deleted:</b> <i>Let</i>
<b>Deleted:</b> is evaluated as follows:
<b>Deleted:</b> Evaluate
<b>Deleted:</b> <i>Let</i>
<b>Deleted:</b> <i>Let</i>
<b>Deleted:</b> <i>Let</i>
<b>Deleted:</b> <i>Let</i>
<b>Deleted:</b> is evaluated as follows:
<b>Deleted:</b> Evaluate <i>Let</i>
<b>Deleted:</b> Evaluate <i>Let</i>
<b>Deleted:</b> Evaluate <i>Let</i>
<b>Deleted:</b> running
<b>Deleted:</b> Perform
<b>Deleted:</b> Initialization
<b>Deleted:</b> The production
<b>Deleted:</b> <i>Let</i>
<b>Deleted:</b> is evaluated as follows:
<b>Deleted:</b> running
<b>Deleted:</b> Perform
<b>Deleted:</b> Initialization
<b>Deleted:</b> <i>Let</i>
<b>Deleted:</b> is evaluated as follows:
<b>Deleted:</b> running
<b>Deleted:</b> Perform
<b>Deleted:</b> Initialization
<b>Deleted:</b> Evaluate
<b>Deleted:</b> <i>BindingIdentifier</i> : <i>Identifier</i> is evaluated as follows: Return the result of evaluating <i>Identifier</i> as described in 11.1.2.
<b>Deleted:</b> is evaluated as follows:
<b>Deleted:</b> The <i>LetBindingListNoIn</i> , <i>LetBindingNoIn</i> and <i>InitialiserNoIn</i> productions are evaluated in the same manner as the <i>LetBindingList</i> , <i>LetBinding</i> and <i>Initialiser</i> productions except that the contained <i>LetBindingListNoIn</i> , <i>LetBindingNoIn</i> ,
<b>Deleted:</b> 12.2.2 Const Declaration
<b>Deleted:</b> 3

## Syntax

VariableStatement :

var VariableDeclarationList ;

VariableDeclarationList :

VariableDeclaration

VariableDeclarationList , VariableDeclaration

VariableDeclarationListNoIn :

VariableDeclarationNoIn

VariableDeclarationListNoIn , VariableDeclarationNoIn

VariableDeclaration :

BindingIdentifier Initialiser<sub>opt</sub>

BindingPattern Initialiser

VariableDeclarationNoIn :

BindingIdentifier InitialiserNoIn<sub>opt</sub>

BindingPattern InitialiserNoIn

The semantics of the VariableDeclarationListNoIn, VariableDeclarationNoIn and InitialiserNoIn productions are the same as the VariableDeclarationList, VariableDeclaration and Initialiser productions except that the contained VariableDeclarationListNoIn, VariableDeclarationNoIn, InitialiserNoIn and AssignmentExpressionNoIn are used in of the contained VariableDeclarationList, VariableDeclaration, Initialiser and AssignmentExpression, respectively.

## Static Semantics

### Static Semantics: BoundNames

VariableStatement : var VariableDeclarationList ;

1. Return BoundNames of VariableDeclarationList.

VariableDeclarationList : VariableDeclaration

1. Return BoundNames of VariableDeclaration.

VariableDeclarationList : VariableDeclarationList , VariableDeclaration

1. Let names be BoundNames of VariableDeclarationList.

2. Append to names the elements of BoundNames of VariableDeclaration.

3. Return names.

VariableDeclaration : BindingIdentifier Initialiser<sub>opt</sub>

1. Return the BoundNames of BindingIdentifier.

VariableDeclaration : BindingPattern Initialiser

1. Return the BoundNames of BindingPattern.

## Runtime Semantics

### Runtime Semantics: Binding Initialisation

With arguments value and environment.

**Deleted:** ¶

**Deleted:** NOTE A var statement declares variables that are scoped to the current execution context's VariableEnvironment. Var variables are created when their containing Lexical Environment is instantiated and are initialised to undefined when created. Within the scope of any VariableEnvironment a common Identifier may appear in more than one VariableDeclaration but those declarations collectively define only one variable. A variable defined by a VariableDeclaration with an Initialiser is assigned the value of its Initialiser's AssignmentExpression when the VariableDeclaration is executed, not when the variable is created. ¶

**Semantics¶**

**Deleted:** Static Semantics: Early Errors¶

VariableDeclaration : BindingPattern Initialiser ¶

<#> It is a Syntax Error if the source code matching this production is not contained in extended code. ¶

**Deleted:** The BoundNames of the production

**Deleted:** is determined as follows:

**NOTE** `undefined` is passed for `environment` to indicate that a `PutValue` operation should be used to assign the initialisation value. This is the case for `var` statements formal parameter lists of non-strict functions. In those cases a lexical binding is hosted and preinitialized prior to evaluation of its initialiser.

#### VariableDeclaration : BindingIdentifier

1. Return the result of performing Binding Initialisation for `BindingIdentifier` passing `value` and `undefined` as the arguments.

#### VariableDeclaration : BindingIdentifier Initialiser

1. Return the result of performing Binding Initialisation for `BindingIdentifier` passing `value` and `undefined` as the arguments.

#### VariableDeclaration : BindingPattern Initialiser

1. Return the result of performing Binding Initialisation for `BindingPattern` passing `value` and `undefined` as the arguments.

### Runtime Semantics: Evaluation

#### VariableStatement : var VariableDeclarationList ;

1. Let `next` be the result of evaluating `VariableDeclarationList`.
2. If `next` is an abrupt completion, return `next`.
3. Return NormalCompletion(`empty`).

#### VariableDeclarationList : VariableDeclaration

1. Return the result of evaluating `VariableDeclaration`.

#### VariableDeclarationList : VariableDeclarationList , VariableDeclaration

1. Let `next` be the result of evaluating `VariableDeclarationList`.
2. If `next` is an abrupt completion, return `next`.
3. Return the result of evaluating `VariableDeclaration`.

#### VariableDeclaration : BindingIdentifier

1. Return NormalCompletion(`empty`).

#### VariableDeclaration : BindingIdentifier Initialiser

1. Let `rhs` be the result of evaluating `Initialiser`.
2. Let `value` be `GetValue(rhs)`.
3. If `value` is an abrupt completion, return `value`.
4. Return the result of performing Binding Initialisation for `BindingIdentifier` passing `value` and `undefined` as the arguments.

**NOTE** If a `VariableDeclaration` is nested within a `with` statement and the `Identifier` in the `VariableDeclaration` is the same as a property name of the binding object of the `with` statement's object environment record, then step 3 will assign `value` to the property instead of to the `VariableEnvironment` binding of the `Identifier`.

#### VariableDeclaration : BindingPattern Initialiser

2. Let `rhs` be the result of evaluating `Initialiser`.
3. Let `rval` be `GetValue(rhs)`.
4. If `rval` is an abrupt completion, return `rval`.
5. Return the result of performing Binding Initialisation for `BindingPattern` passing `rval` and `undefined` as arguments.

**Deleted:** the  
**Deleted:** The production  
**Deleted:** is evaluated as follows:  
**Deleted:** Evaluate  
**Deleted:** (normal,  
**Deleted:** y, empty)

**Deleted:** The BoundNames of the production  
`VariableDeclarationList : VariableDeclaration` is determined as follows:  
`<#>Return BoundNames of VariableDeclaration`  
The production

**Deleted:** is evaluated as follows:  
**Deleted:** Evaluate

**Deleted:** The BoundNames of the production  
`VariableDeclarationList : VariableDeclaration`  
`VariableDeclaration` is determined as follows:  
`<#>Let names be BoundNames of VariableDeclarationList.`  
`<#>Append to names the elements of BoundNames of VariableDeclaration.`  
`<#>Return names.`  
The production

**Deleted:** is evaluated as follows:

**Deleted:** Evaluate  
**Deleted:** Evaluate  
**Deleted:** The BoundNames of the production  
`VariableDeclaration : BindingIdentifier Initialiser` is determined as follows: ¶  
`<#>Return the BoundNames of BindingIdentifier`  
The production

**Deleted:** is evaluated as follows:  
**Deleted:** the String value of `BindingIdentifier`  
**Deleted:** The production

**Deleted:** is evaluated as follows:  
**Deleted:** Perform  
**Deleted:** Initialization

**Deleted:** NOTE 1 The String value of a `VariableDeclaration` is used in the evaluation of for-in statements (12.6.4).¶  
**Deleted:** 2  
**Deleted:** 4

**Deleted:** The static semantics of the production `VariableDeclaration : BindingPattern Initialiser` are: ¶  
**Deleted:** is evaluated as follows:  
**Deleted:** ToObject()  
**Deleted:** )

**Deleted:** Perform  
**Deleted:** Initialization  
**Deleted:** Evaluate  
**Deleted:** using  
**Deleted:** as the `obj` parameter  
**Deleted:** the `environment` parameter

## 12.2.4 Destructuring Binding Patterns

### Syntax

#### BindingPattern :

ObjectBindingPattern  
ArrayBindingPattern

#### ObjectBindingPattern :

{ }  
{ BindingPropertyList }  
{ BindingPropertyList , }

#### ArrayBindingPattern :

[ Elision<sub>opt</sub> BindingRestElement<sub>opt</sub> ]  
[ BindingElementList ]  
[ BindingElementList , Elision<sub>opt</sub> BindingRestElement<sub>opt</sub> ]

#### BindingPropertyList :

BindingProperty  
BindingPropertyList , BindingProperty

#### BindingElementList :

Elision<sub>opt</sub> BindingElement  
BindingElementList , Elision<sub>opt</sub> BindingElement

#### BindingProperty :

SingleNameBinding  
PropertyName : BindingElement

#### BindingElement :

SingleNameBinding  
BindingPattern Initialiser<sub>opt</sub>

#### SingleNameBinding :

BindingIdentifier Initialiser<sub>opt</sub>

#### BindingRestElement :

... BindingIdentifier

### Static Semantics

#### Static Semantics: Early Errors

##### BindingPattern : ObjectBindingPattern

- It is a Syntax Error if the Bound Names of ObjectBindingPattern contains the string “eval” or the string “arguments”.

##### BindingPattern : ObjectBindingPattern

- It is a Syntax Error if the Bound Names of ArrayBindingPattern contains the string “eval” or the string “arguments”.

#### Static Semantics: Bound Names

##### BindingPattern : ObjectBindingPattern

1. Return the Bound Names of ObjectBindingPattern.

**Deleted:** The VariableDeclarationListNoIn, VariableDeclarationNoIn and InitialiserNoIn productions are evaluated in the same manner as the VariableDeclarationList, VariableDeclaration and Initialiser productions except that the contained VariableDeclarationListNoIn, VariableDeclarationNoIn, InitialiserNoIn and AssignmentExpressionNoIn are evaluated instead of the contained VariableDeclarationList, VariableDeclaration, Initialiser and AssignmentExpression respectively.¶

**Deleted:** VariableStatement :¶  
var VariableDeclarationList ;¶  
VariableDeclarationList :¶  
VariableDeclaration  
VariableDeclarationList , VariableDeclaration¶  
VariableDeclarationListNoIn , VariableDeclarationNoIn¶  
VariableDeclarationNoIn .  
VariableDeclarationListNoIn , VariableDeclarationNoIn¶  
VariableDeclaration :¶  
Identifier Initialiser<sub>opt</sub>¶  
VariableDeclarationNoIn :¶  
Identifier InitialiserNoIn<sub>opt</sub>¶  
Initialiser :¶  
= AssignmentExpression¶  
InitialiserNoIn :¶  
= AssignmentExpressionNoIn¶

**Comment [AW37]:** The destructuring wiki page argues for allowing empty binding patterns. I'm not really convinced.

**Comment [AW38]:** See above comment about empty binding patterns.

**Comment [AW39]:** See above comment about empty binding patterns.

**Deleted:** Identifier

**Deleted:** PropertyName : SingleNameBinding

**Deleted:** PropertyName : BindingPattern Initialiser<sub>opt</sub>

**Deleted:** Identifier .

**Deleted:** Identifier

**Formatted:** Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

**Formatted:** Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

**Deleted:** Semantics¶

**Deleted:** A variable statement declares variables that are created as defined in 10.5. Variables are initialised undefined when created. A variable with an Initialiser assigned the value of its AssignmentExpression when the VariableStatement is executed, not when the variable is created.¶

**Semantics¶**

The production VariableStatement : var  
VariableDeclarationList ; is evaluated as follows:¶

<#>Evaluate VariableDeclarationList.¶

<#>Return (normal, empty, empty).¶

The production

VariableDeclarationList : VariableDeclaration is evaluated as follows:¶

<#>Evaluate VariableDeclaration.¶

The production VariableDeclarationList :

VariableDeclarationList , VariableDeclaration is evaluated as follows:¶

**Deleted:** The BoundNames of the production

**Deleted:** is determined as follows:

*BindingPattern : ArrayBindingPattern*

1. Return BoundNames of *ArrayBindingPattern*.

*ObjectBindingPattern: { }*

1. Return an empty List.

*ObjectBindingPattern :*

```
{ BindingPropertyList }
{ BindingPropertyList , }
```

1. Return the BoundNames of *BindingPropertyList*.

*ArrayBindingPattern : [ Elision<sub>opt</sub> ]*

1. Return an empty List.

*ArrayBindingPattern : [ Elision<sub>opt</sub> BindingRestElement ]*

1. Return the BoundNames of *BindingElementList*.

*ArrayBindingPattern : [ BindingElementList ]*

1. Return the BoundNames of *BindingElementList*.

*ArrayBindingPattern : [ BindingElementList , Elision<sub>opt</sub> ]*

2. Return the BoundNames of *BindingElementList*.

*ArrayBindingPattern : [ BindingElementList , Elision<sub>opt</sub> BindingRestElement ]*

1. Let *names* be BoundNames of *BindingElementList*.

2. Append to *names* the elements of BoundNames of *BindingRestElement*.

3. Return *names*.

*BindingPropertyList : BindingProperty*

1. Return BoundNames of *BindingProperty*.

*BindingPropertyList : BindingPropertyList , BindingProperty*

1. Let *names* be BoundNames of *BindingPropertyList*.

2. Append to *names* the elements of BoundNames of *BindingProperty*.

3. Return *names*.

*BindingElementList : Elision<sub>opt</sub> BindingElement*

1. Return BoundNames of *BindingElement*.

*BindingElementList : BindingElementList , Elision<sub>opt</sub> BindingElement*

1. Let *names* be BoundNames of *BindingElementList*.

2. Append to *names* the elements of BoundNames of *BindingElement*.

3. Return *names*.

*BindingProperty : SingleNameBinding*

1. Return the BoundNames of *SingleNameBinding*.

**Comment [AW40]:** The destructuring wiki page argues for allowing empty binding patterns. I'm not really convinced.

BindingProperty : PropertyName : BindingElement

1. Return the BoundNames of *BindingElement*.

SingleNameBinding : BindingIdentifier Initialiser<sub>opt</sub>

1. Return the BoundNames of *BindingIdentifier*.

BindingElement : SingleNameBinding

1. Return the BoundNames of *SingleNameBinding*.

BindingElement : BindingPattern Initialiser<sub>opt</sub>

1. Return the BoundNames of *BindingPattern*.

BindingRestElement : ... BindingIdentifier

1. Return the BoundNames of *BindingIdentifier*.

#### **Static Semantics: HasInitialiser**

BindingElement : SingleNameBinding

1. Return HasInitializer of *SingleNameBinding*.

BindingElement : BindingPattern

1. Return **false**.

BindingElement : BindingPattern Initialiser

1. Return **true**.

SingleNameBinding : BindingIdentifier Initialiser

1. Return **false**.

SingleNameBinding : BindingIdentifier Initialiser

1. Return **true**.

#### **Runtime Semantics**

##### **Runtime Semantics: Binding Initialisation**

With parameters *value* and *environment*.

**NOTE** When **undefined** is passed for *environment* it indicates that a PutValue operation should be used to assign the **initialisation** value. This is the case for formal parameter lists of non-strict functions. In that case the formal parameter bindings are preinitialized in order to deal with the possibility of multiple parameters with the same name.

BindingPattern : ObjectBindingPattern

1. If *value* is neither of **null** or **undefined**, then  
  a. Let *obj* be ToObject(*value*).

2. Else, let *obj* be **undefined**.

3. **Return the result of performing BindingInitialisation for ObjectBindingPattern using *obj* and *environment* as arguments.**

**Comment [AWB41]:** Note, as currently written destructuring array assignment doesn't take into account the length property of the source object but destructuring binding does. We need to decide which semantic we want to consistently apply to both.

**Deleted:** Initialization

**Deleted:** environment

**Deleted:** initialization

**Deleted:** the

**Deleted:** The production

**Deleted:** is evaluated with parameters *value* and *env* as follows:  
Evaluate

**Deleted:** Perform

**Deleted:** Initialization

**Deleted:** *value* as the

**Deleted:** parameter

**Deleted:** *env* as the

**Deleted:** parameter

### BindingPattern : ArrayBindingPattern

1. If value is neither of **null** or **undefined**, then
  - a. Let array be ToObject(value).
2. Else,
  - a. Let array be **undefined**.
3. Return the result of performing Indexed Binding Initialisation for ArrayBindingPattern using array, 0, and environment as arguments.

### ObjectBindingPattern: { }

1. Return.

### ObjectBindingPattern :

{ BindingPropertyList }  
{ BindingPropertyList , }

1. Return the result of performing Binding Initialisation for BindingPropertyList using value and environment as arguments.

### BindingPropertyList : BindingProperty

1. Return the result of performing Binding Initialisation for BindingProperty using value and environment as arguments.

### BindingPropertyList : BindingPropertyList , BindingProperty

1. Let next be the result of performing Binding Initialisation for BindingPropertyList using value and environment as arguments.
2. ReturnIfAbrupt(next).
3. Return the result of performing Binding Initialisation for BindingProperty using value and environment as arguments.

### BindingProperty : SingleNameBinding

1. Let name be the string that is the only element of BoundNames of SingleNameBinding.
2. Return the result of performing Keyed Binding Initialisation for SingleNameBinding using value, environment, and name as the arguments.

### BindingProperty : PropertyName : BindingElement

1. Let P be the PropName of PropertyName.
2. Return the result of performing Keyed Binding Initialisation for BindingElement using value, environment, and P as arguments.

## Runtime Semantics: Indexed Binding Initialisation

With parameters array, nextIndex, and environment.

**NOTE** When **undefined** is passed for environment it indicates that a PutValue operation should be used to assign the initialisation value. This is the case for formal parameter lists of non-strict functions. In that case the formal parameter bindings are preinitialized in order to deal with the possibility of multiple parameters with the same name.

### ArrayBindingPattern : [ Elision<sub>opt</sub> ]

1. Return.

### ArrayBindingPattern: [ Elision<sub>opt</sub> BindingRestElement ]

<b>Deleted:</b> The BoundNames of the production <u>BindingPattern : ArrayBindingPattern</u> is determined as follows: ↪ Return BoundNames of <u>ArrayBindingPattern</u> . The production
<b>Deleted:</b> ↪ Let <u>lenValue</u> be the result of calling the [[Get]] internal method of <u>array</u> with argument "length". ↪ Let <u>arrayLength</u> be <u>ToUint32</u> ( <u>lenValue</u> ). <b>Deleted:</b> ↪ Let <u>arrayLength</u> be 0. <b>Deleted:</b> is evaluated with parameters <u>value</u> and <u>env</u> as follows: <b>Deleted:</b> Perform ...ndexed Binding Initializati... <b>Deleted:</b> Evaluate ...rrayBindingPattern using <b>Deleted:</b> <u>arrayLength</u> , <u>undefined</u>
<b>Formatted</b> <b>Deleted:</b> as the <u>obj</u> parameter ...nd <u>env</u> as the <b>Deleted:</b> The BoundNames of the production <u>ObjectBindingPattern: { }</u> is determined as follows: ↪ Return an empty List. The production ...jectBindingPattern: { } is <b>Deleted:</b> with parameters <u>value</u> and <u>env</u> as follows: If <u>value</u> is neither of <b>null</b> or <b>undefined</b> , then perf... ToObject( <u>value</u> ) and discard the result
<b>Comment [AW42]:</b> The destructuring wiki page argues for allowing empty binding patte... I'm not really convinced.
<b>Deleted:</b> The BoundNames of the productions <u>ObjectBindingPattern: { BindingPropertyList }</u> and <u>ObjectBindingPattern :</u> <b>Deleted:</b> Return the BoundNames of <b>Deleted:</b> Perform ...nding Initialization <b>Deleted:</b> Evaluate ...indingPropertyList using <b>Deleted:</b> The BoundNames of the production <b>Deleted:</b> with parameters <u>obj</u> and <u>env</u> as <b>Deleted:</b> Perform ...nding Initialization <b>Deleted:</b> Evaluate ...indingProperty using <b>Deleted:</b> The BoundNames of the production <b>Deleted:</b> with parameters <u>obj</u> and <u>env</u> as <b>Deleted:</b> Perform ...nding Initialization <b>Deleted:</b> Evaluate ...indingPropertyList using <b>Deleted:</b> If <u>next</u> is an abrupt completion, return <b>Deleted:</b> Perform ...nding Initialization <b>Deleted:</b> Evaluate ...indingProperty using <b>Deleted:</b> Perform ...keyed Binding Initializatio... <b>Deleted:</b> Perform ...keyed Binding Initializatio... <b>Deleted:</b> Evaluate <b>Deleted:</b> Initialization <b>Deleted:</b> <u>arrayLength</u> , <b>Deleted:</b> initialization <b>Deleted:</b> the

1. If *Elision* is present, then let *nextIndex* be the result of evaluating *Elision*, otherwise let *nextIndex* be 0.
2. Return the result of performing Indexed Binding Initialisation for *BindingRestElement* using *array*, *nextIndex*, and *environment* as arguments.

ArrayBindingPattern: [ BindingElementList ]

1. Return the result of performing Indexed Binding Initialisation for *BindingElementList* using *array*, *nextIndex*, and *environment* as arguments.

ArrayBindingPattern: [ BindingElementList , Elision<sub>opt</sub> ]

1. Return the result of performing Indexed Binding Initialisation for *BindingElementList* using *array*, *nextIndex*, and *environment* as arguments.

ArrayBindingPattern: [ BindingElementList , Elision<sub>opt</sub> BindingRestElement ]

1. Let *index* be the result of performing Indexed Binding Initialisation for *BindingElementList* using *array*, *nextIndex*, and *environment* as arguments.
2. ReturnIfAbrupt(*index*).
3. If *Elision* is present, then let *skip* be the result of evaluating *Elision*, otherwise let *skip* be 0.
4. Return the result of performing Indexed Binding Initialisation for *BindingRestElement* using *array*, *index+skip*, and *environment* as arguments.

BindingElementList : Elision<sub>opt</sub> BindingElement

1. If *Elision* is present, then let *skip* be the result of evaluating *Elision*, otherwise let *skip* be 0.
2. Let *next* be the result of performing Indexed Binding Initialisation for *BindingElement* using *array*, *nextIndex+skip*, and *environment* as arguments.
3. ReturnIfAbrupt(*next*).
4. Return *index+skip+1*.

BindingElementList : BindingElementList , Elision<sub>opt</sub> BindingElement

1. Let *listNext* be the result of performing Indexed Binding Initialisation for *BindingElementList* using *array*, *nextIndex*, and *environment* as arguments.
2. ReturnIfAbrupt(*listNext*).
3. If *Elision* is present, then let *skip* be the result of evaluating *Elision*, otherwise let *skip* be 0.
4. Let *next* be the result of performing Indexed Binding Initialisation for *BindingElement* using *array*, *listNext+skip*, and *environment* as arguments.
5. ReturnIfAbrupt(*listNext*).
6. Return *listNext+skip+1*.

BindingElement : SingleNameBinding

1. Return the result of performing Keyed Binding Initialisation for *SingleNameBinding* using *array*, *environment*, and *ToString(nextIndex)* as the arguments.

BindingElement : BindingPattern Initialiser<sub>opt</sub>

1. Let *P* be *ToString(nextIndex)*.
2. Let *exists* be the result of calling the *[[HasProperty]]* internal method of *array* with argument *P*.
3. If *exists* is *true*, then
  - a. Let *v* be the result of calling the *[[Get]]* internal method of *array* with argument *P*.
4. Else
  - a. If *Initialiser<sub>opt</sub>* is present, then
    - i. Let *v* be the result of evaluating *Initialiser*.
  - b. Else,
    - i. Let *v* be *undefined*.
5. ReturnIfAbrupt(*v*).

<b>Deleted:</b> Perform ...ndexed Binding Initialization
<b>Deleted:</b> <i>arrayLength</i> ,
<b>Deleted:</b> Perform ...ndexed Binding Initialization
<b>Deleted:</b> Perform ...ndexed Binding Initialization
<b>Deleted:</b> <i>0</i> ,
<b>Deleted:</b> Initialization
<b>Deleted:</b> ,
<b>Deleted:</b> <i>0</i>
<b>Deleted:</b> If <i>index</i> is an abrupt completion, return <i>index</i> .
<b>Deleted:</b> Perform ...ndexed Binding Initialization
<b>Deleted:</b> <i>arrayLength</i> ,
<b>Deleted:</b> The BoundNames of the production <i>BindingElementList : Elision<sub>opt</sub> BindingElement</i> is
<b>Deleted:</b> with the parameters <i>obj</i> , <i>env</i> , and <i>index</i> as
<b>Deleted:</b> Perform ...ndexed Binding Initialization
<b>Deleted:</b> Evaluate ...ndingElement using <i>obj...rray</i> as the
<b>Deleted:</b> <i>arrayLength</i> ,
<b>Deleted:</b> <i>env</i> as the <i>env</i> parameter, and <i>index+skip</i> ...s the
<b>Deleted:</b> If <i>next</i> is an abrupt completion, return <i>next</i> .
<b>Deleted:</b> The BoundNames of the production
<b>Deleted:</b> is evaluated with the parameters <i>obj</i> , <i>env</i> , and
<b>Deleted:</b> Initialization
<b>Deleted:</b> <i>obj</i>
<b>Deleted:</b> <i>arrayLength</i> ,
<b>Deleted:</b> as the <i>obj</i> parameter, <i>env</i> as the <i>env</i> parameter, and
<b>Deleted:</b> If <i>next</i> is an abrupt completion, return <i>next</i>
<b>Deleted:</b> Evaluate
<b>Deleted:</b> Perform ...ndexed Binding Initialization
<b>Deleted:</b> <i>obj...rray</i> as the <i>obj</i> parameter
<b>Deleted:</b> <i>arrayLength</i> ,
<b>Deleted:</b> <i>env</i> as the <i>env</i> parameter
<b>Deleted:</b> ,
<b>Deleted:</b> and ...istNext+skip , and <i>environment</i> as the <i>index</i>
<b>Deleted:</b> If <i>listNext</i> is an abrupt completion, return <i>listNext</i>
<b>Deleted:</b> ¶
<b>Deleted:</b> The BoundNames of the productions
<b>Deleted:</b> <#>If <i>nextIndex</i> ≥ <i>arrayLength</i> , then¶
<b>Deleted:</b> with the parameters <i>obj</i> , <i>env</i> and <i>index</i> as
<b>Deleted:</b> Perform ...keyed Binding Initialization
<b>Deleted:</b> Evaluate
<b>Deleted:</b> <i>P</i> , <i>obj...rray</i> , <i>environment</i> , and <i>ToString( env</i>
<b>Deleted:</b> The BoundNames of the production
<b>Deleted:</b> <#>If <i>nextIndex</i> ≥ <i>arrayLength</i> , then¶
<b>Deleted:</b> with the parameters <i>obj</i> , <i>env</i> and <i>index</i> as
<b>Deleted:</b> <i>obj...rray</i> with argument <i>P</i> passing <i>P</i> as the
<b>Deleted:</b> If <i>v</i> is an abrupt completion, return <i>v</i> .

6. Return the result of performing Binding Initialisation for BindingPattern passing v and environment as arguments.

**Deleted:** Perform

**Deleted:** Initialization

**Deleted:** Evaluate

**Deleted:** v

**Deleted:** env

**Deleted:** a new array object created as if by the expression `new Array()` where `Array` is the standard built-in constructor with that name

**Deleted:** If arrayLength is an abrupt completion return arrayLength.

#### BindingRestElement : ... BindingIdentifier

1. Let A be the result of the abstraction operation `ArrayCreate` (15.4) with argument 0.
2. Let lenVal be the result of calling the `[[Get]]` internal method of array with argument `"length"`.
3. Let arrayLength be `ToUint32(lenVal)`.
4. ReturnIfAbrupt(arrayLength).
5. Let n=0.
6. Let index = nextIndex.
7. Repeat, while index < arrayLength
  - a. Let P be `ToString(index)`.
  - b. Let exists be the result of calling the `[[HasProperty]]` internal method of array with argument P.
  - c. If exists is true, then
    - i. Let v be the result of calling the `[[Get]]` internal method of array passing P as the argument.
    - ii. ReturnIfAbrupt(v).
    - iii. Call the `[[DefineOwnProperty]]` internal method of A with arguments `ToString(n)`, Property Descriptor `{[[Value]]: v, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, and false.
  - d. Let n = n+1.
  - e. Let index = index+1.
8. Return the result of performing Binding Initialisation for BindingIdentifier using A and environment as arguments.

**Deleted:** If v is an abrupt completion, return v

**Formatted:** Font: Italic

**Deleted:** `.[[value]]`

**Deleted:** Perform

**Deleted:** Initialization

**Deleted:** array

**Deleted:** Initialization

**Deleted:** initialization

**Deleted:** the

#### Runtime Semantics: Keyed Binding Initialisation

With parameters obj, environment, and propertyName.

**NOTE** When undefined is passed for environment it indicates that a `PutValue` operation should be used to assign the initialisation value. This is the case for formal parameter lists of non-strict functions. In that case the formal parameter bindings are preinitialized in order to deal with the possibility of multiple parameters with the same name.

#### BindingElement: SingleNameBinding

1. Return the result of performing Keyed Binding Initialisation for SingleNameBinding using obj, environment, and propertyName as the arguments.

**Deleted:** Perform

**Deleted:** Initialization

#### BindingElement: BindingPattern Initialiser<sub>opt</sub>

1. Let exists be the result of calling the `[[HasProperty]]` internal method of obj with argument propertyName.
2. If exists is true, then
  - a. Let v be the result of calling the `[[Get]]` internal method of obj with argument propertyName.
3. Else
  - a. If Initialiser<sub>opt</sub> is present, then
    - i. Let v be the result of evaluating Initialiser.
  - b. Else
    - i. Let v be undefined.
4. ReturnIfAbrupt(v).
5. Return the result of performing Binding Initialisation for BindingPattern passing v and environment as arguments.

**Deleted:** If v is an abrupt completion, return v.

**Deleted:** Perform

**Deleted:** Initialization

#### SingleNameBinding : BindingIdentifier Initialiser<sub>opt</sub>

1. Let exists be the result of calling the `[[HasProperty]]` internal method of obj with argument propertyName.
2. If exists is true, then
  - a. Let v be the result of calling the `[[Get]]` internal method of obj passing propertyName as the argument.
3. Else
  - a. If Initialiser<sub>opt</sub> is present, then
    - i. Let v be the result of evaluating Initialiser.

b. Else,  
i. Let  $v$  be **undefined**.

4. **ReturnIfAbrupt**( $v$ ).  
5. Return the result of performing **BindingInitialisation** for **BindingIdentifier** passing  $v$  and **environment** as arguments.

## 12.3 Empty Statement

### Syntax

**EmptyStatement** :  
;

### Runtime Semantics

#### Runtime Semantics: Evaluation

**EmptyStatement** : ;

1. Return **NormalCompletion**(empty).

## 12.4 Expression Statement

### Syntax

**ExpressionStatement** :  
[lookahead  $\notin \{\{, \text{function}\}\}] \text{Expression} ;$

**NOTE** An **ExpressionStatement** cannot start with an opening curly brace because that might make it ambiguous with a **Block**. Also, an **ExpressionStatement** cannot start with the **function** keyword because that might make it ambiguous with a **FunctionDeclaration**.

### Runtime Semantics

#### Runtime Semantics: Evaluation

**ExpressionStatement** : [lookahead  $\notin \{\{, \text{function}\}\}] \text{Expression} ;$

1. Let  $exprRef$  be the result of evaluating **Expression**.
2. Let  $value$  be **GetValue**( $exprRef$ ).
3. If  $value$  is an abrupt completion, return  $value$ .
4. Return **NormalCompletion**( $value$ ).

## 12.5 The if Statement

### Syntax

**IfStatement** :  
if ( **Expression** ) **Statement** else **Statement**  
if ( **Expression** ) **Statement**

Each **else** for which the choice of associated **if** is ambiguous shall be associated with the nearest possible **if** that would otherwise have no corresponding **else**.

### Static Semantics: VarDeclaredNames

**IfStatement** : if ( **Expression** ) **Statement** else **Statement**

1. Let  $names$  be **VarDeclaredNames** of the first **Statement**.
2. Append to  $names$  the elements of the **VarDeclaredNames** of the second **Statement**.

**Deleted:** If  $v$  is an abrupt completion, return  $v$ .

**Deleted:** Perform

**Deleted:** Initialization

**Deleted:** environment

**Deleted:** The BoundNames of the production  
**BindingRestElement** : ... **BindingIdentifier** is determined as follows:  
↳ Return the BoundNames of **BindingIdentifier**.

**Deleted:** The BoundNames of the production  
**BindingRestElement** : ... **Identifier** is determined as follows:  
↳ Return BoundNames of **BindingElement**.

**Deleted:** The production **BindingRestElement** : ...  
**BindingIdentifier** is evaluated with the parameters **obj**, **env** and **index** as follows:  
↳ Let  $lenVal$  be the result of calling the **[[Get]]** internal method of **obj** with argument "**length**".  
↳ Let  $len$  be **ToUint32**( $lenVal$ ).  
↳ Let  $A$  be a new array object created as if by the expression **new Array()** where **Array** is the standard built-in constructor with that name.  
↳ Let  $n=0$ .  
↳ Repeat, while  $index < len$ :  
  ↳ Let  $P$  be **ToString**( $index$ ).  
  ↳ Let  $exists$  be the result of calling the **[[HasProperty]]** internal method of **obj** with argument  $P$ .  
  ↳ If  $exists$  is **true**, then:  
    ↳ Let  $v$  be the result of calling the **[[Get]]** internal method of **obj** passing  $P$  as the argument.  
    ↳ Call the **[[DefineOwnProperty]]** internal method of  $A$  with arguments **ToString**( $n$ ), Property Descriptor **{[[Value]]: v, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}**, and **false**.  
  ↳ Let  $n = n+1$ .  
  ↳ Let  $index = index+1$ .  
↳ Perform Binding Initialization for **BindingIdentifier** using  $v$  as the **value** and **env** as the **enviornment**.  
  ↳

**Deleted:** Semantics||  
The production

**Deleted:** is evaluated as follows:

**Deleted:** (normal,

**Deleted:** , empty).

**Deleted:** Semantics||  
The production

**Deleted:** is evaluated as follows:

**Deleted:** (normal, **GetValue**( $exprRef$ ))

**Deleted:** , empty

**Deleted:** Semantics||  
The **VarDeclaredNames** of the production

**Deleted:** is determined as follows:

### 3. Return names.

IfStatement : if ( Expression ) Statement

1. Return the VarDeclaredNames of Statement.

### Runtime Semantics

#### Runtime Semantics: Evaluation

IfStatement : if ( Expression ) Statement else Statement

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be *ToBoolean(GetValue(exprRef))*.
3. If *exprValue* is an abrupt completion, return *exprValue*.
4. If *exprValue* is true, then
  - a. Return the result of evaluating the first *Statement*.
5. Else,
  - a. Return the result of evaluating the second *Statement*.

IfStatement : if ( Expression ) Statement

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be *ToBoolean(GetValue(exprRef))*.
3. If *exprValue* is an abrupt completion, return *exprValue*.
4. If *exprValue* is false, return *NormalCompletion(undefined)*.
5. Return the result of evaluating *Statement*.

## 12.6 Iteration Statements

### Syntax

IterationStatement :

```

do Statement while ( Expression )
while ( Expression ) Statement
for (ExpressionNoInopt; Expressionopt ; Expressionopt ) Statement
for ( var VariableDeclarationListNoIn ; Expressionopt ; Expressionopt ) Statement
for ( LexicalDeclarationNoIn ; Expressionopt ; Expressionopt ) Statement
for ( LeftHandSideExpression in Expression ) Statement
for ( var ForBindingopt in Expression ) Statement
for ( ForDeclaration in Expression ) Statement
for ( LeftHandSideExpression of Expression ) Statement
for ( var ForBinding of Expression ) Statement
for ( ForDeclaration of Expression ) Statement

```

ForDeclaration :

LetOrConst ForBinding

NOTE 1 ForBinding is defined in 11.1.4.2.

NOTE 2 A semicolon is not required after a do-while statement.

The abstract operation *LoopContinues* with arguments *completion* and *labelSet* is defined by the following step:

1. If *completion.[[type]]* is normal, then return true.
2. If *completion.[[type]]* is not continue, then return false.
3. If *completion.[[target]]* is empty, then return true.
4. If *completion.[[target]]* is an element of *labelSet*, then return true.
5. Return false.

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** *ToBoolean(GetValue(exprRef))*

**Deleted:** The VarDeclaredNames of the production *IfStatement : if ( Expression ) Statement* is determined as follows:  
 ↳ Return the VarDeclaredNames of Statement.  
 ↳ The production

**Deleted:** is evaluated as follows:

**Comment [AWB643]:** Breaking change from ES5: completion reform

**Deleted:** *ToBoolean(GetValue(exprRef))*

**Deleted:** (normal, empty)

**Deleted:** , empty).

**Deleted:** ;

**Comment [AWB644]:** This is breaking change from ES5 which allowed a *VariableDeclarationNoIn* to appear here. See discuss thread "lexical for-in/of loose end"

**Deleted:** *BindingIdentifier*

**Deleted:** *VariableDeclarationNoIn*

**Deleted:** *ForVarDeclaration*

**Deleted:** *ForVarDeclaration : BindingPattern VariableDeclarationNoIn*

**Comment [AWB645]:** Note that this is technically a breaking change from ES5, however it is made to match web reality.

**Deleted:** *ForBinding : BindingIdentifier . BindingPattern*

**Formatted:** Font: 9 pt

**Deleted:** *loopContinues*

## 12.6.1 The do-while Statement

### Static Semantics: VarDeclaredNames

IterationStatement : do Statement while ( Expression ) ;

1. Return the VarDeclaredNames of *Statement*.

### Runtime Semantics

#### Runtime Semantics: Labelled Evaluation

With argument *labelSet*.

IterationStatement : do Statement while ( Expression ) ;

1. Let *V* = undefined.
2. Repeat
  - a. Let *stmt* be the result of evaluating *Statement*.
  - b. If *stmt.[[value]]* is not empty, let *V* = *stmt.[[value]]*.
  - c. If *stmt* is an abrupt completion and LoopContinues (stmt,labelSet) is false, return *stmt*.
  - d. Let *exprRef* be the result of evaluating *Expression*.
  - e. Let *exprValue* be ToBoolean(GetValue(exprRef)).
  - f. If *exprValue* is false, Return NormalCompletion(V).
  - g. Else if *exprValue* is a Completion Record, then
    - i. Assert: *exprValue* is an abrupt completion.
    - ii. If LoopContinues (exprValue,labelSet) is false, return *exprValue*.

## 12.6.2 The while Statement

### Static Semantics: VarDeclaredNames

IterationStatement : while ( Expression ) Statement

1. Return the VarDeclaredNames of *Statement*.

### Runtime Semantics

#### Runtime Semantics: Labelled Evaluation

With argument *labelSet*.

IterationStatement : while ( Expression ) Statement

1. Let *V* = undefined.
2. Repeat
  - a. Let *exprRef* be the result of evaluating *Expression*.
  - b. Let *exprValue* be ToBoolean(GetValue(exprRef)).
  - c. If *exprValue* is false, return NormalCompletion(V).
  - d. Else if *exprValue* is a Completion Record, then
    - i. Assert: *exprValue* is an abrupt completion.
    - ii. If LoopContinues (exprValue,labelSet) is false, return *exprValue*.
  - e. Let *stmt* be the result of evaluating *Statement*.
  - f. If *stmt.[[value]]* is not empty, let *V* = *stmt.[[value]]*.
  - g. If LoopContinues (stmt,labelSet) is false, return *stmt*.

## 12.6.3 The for Statement

### Static Semantics: VarDeclaredNames

**Deleted:** The VarDeclaredNames of the production  
**Deleted:** is determined as follows:

**Deleted:** Runtime Semantics: Evaluation¶

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Comment [AWB646]:** Breaking change: completion reform

**Deleted:** empty

**Deleted:** <#>Let *iterating* be true.¶

**Deleted:** ,

**Deleted:** while *iterating* is true

**Deleted:** <#>If *stmt.type* is not continue or|| *stmt.target* is in the current label set, then¶  
 <#>If *stmt.type* is break and *stmt.target* is in the current label set, return (normal, *V*, empty).¶

**Deleted:** loopContinues

**Deleted:** ToBoolean(GetValue(exprRef))

**Deleted:** set *iterating* to false

**Comment [AWB647]:** Break/continue/return in the expression works normally (future for do {} or block label expressions)

**Deleted:** loopContinues

**Deleted:** <#>Return (normal, *V*, empty);¶

**Deleted:** The VarDeclaredNames of the production

**Deleted:** *Statement* is determined as follows:

**Deleted:** Runtime Semantics: Evaluation¶

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Comment [AWB648]:** ES5 breaking change: completion reform

**Deleted:** empty

**Deleted:** ToBoolean(GetValue(exprRef))

**Deleted:** (normal,

**Deleted:** , empty

**Comment [AWB649]:** Break/continue/return in the expression works normally (future for do {} or block label expressions)

**Deleted:** loopContinues

**Deleted:** <#>If *stmt.type* is not continue or|| *stmt.target* is in the current label set, then¶  
 <#>If *stmt.type* is break and *stmt.target* is in the current label set, then¶  
 <#>Return (normal, *V*, empty).¶

**Deleted:** *stmt* is an abrupt completion

**Deleted:** loopContinues

**Comment [AWB 350]:** The lexical scoping of for iteration variables still needs to be taken care of

### for (ExpressionNoIn<sub>opt</sub> ; Expression<sub>opt</sub> ; Expression<sub>opt</sub>) Statement

1. Return the VarDeclaredNames of Statement.

### IterationStatement : for ( var VariableDeclarationListNoIn ; Expression<sub>opt</sub> ; Expression<sub>opt</sub>) Statement

1. Let names be BoundNames of VariableDeclarationListNoIn.
2. Append to names the elements of the VarDeclaredNames of Statement.
3. Return names.

### for (LexicalDeclarationNoIn ; Expression<sub>opt</sub> ; Expression<sub>opt</sub>) Statement

1. Return the VarDeclaredNames of Statement.

## Runtime Semantics

### Runtime Semantics: Labelled Evaluation

With argument *labelSet*.

### IterationStatement : for (ExpressionNoIn<sub>opt</sub> ; Expression<sub>opt</sub> ; Expression<sub>opt</sub>) Statement

1. If ExpressionNoIn is present, then.
  - a. Let *exprRef* be the result of evaluating ExpressionNoIn.
    - b. Let *exprValue* be GetValue(*exprRef*). (This value is not used but the call may have side-effects.)
    - c. If LoopContinues(*exprValue.labelSet*) is false, return *exprValue*.
2. Return the result of performing For Body Evaluation with the first Expression as the *incrementExpr* argument, the second Expression as the *testExpr* argument and with *labelSet*.

The abstraction operation For Body Evaluation with arguments *testExpr*, *incrementExpr*, and *labelSet* is performed as follows:

1. Let *V* = undefined.
2. Repeat
  - a. If *testExpr* is not empty, then
    - i. Let *testExprRef* be the result of evaluating *testExpr*.
    - ii. Let *testExprValue* be ToBoolean(GetValue(*testExprRef*)).
    - iii. If *testExprValue* is false, return NormalCompletion(V).
    - iv. Else LoopContinues(*testExprValue.labelSet*) is false, return *testExprValue*.
  - b. Let *stmt* be the result of evaluating Statement.
  - c. If *stmt.[value]* is not empty, let *V* = *stmt.[value]*.
  - d. If *stmt* LoopContinues (*stmt.labelSet*) is false, return *stmt*.
  - e. If *incrementExpr* is not empty, then
    - i. Let *incExprRef* be the result of evaluating *incrementExpr*.
    - ii. Let *incExprValue* be GetValue(*incExprRef*).
    - iii. If LoopContinues(*incExprValue.labelSet*) is false, return *incExprValue*.

### IterationStatement : for ( var VariableDeclarationListNoIn ; Expression<sub>opt</sub> ; Expression<sub>opt</sub>) Statement

1. Let *varDcl* be the result of evaluating VariableDeclarationListNoIn.
2. If LoopContinues(*varDcl.labelSet*) is false, return *varDcl*.
3. Return the result of performing For Body Evaluation with the first Expression as the *incrementExpr* argument, the second Expression as the *testExpr* argument and with *labelSet*.

### IterationStatement : for ( LexicalDeclarationNoIn ; Expression<sub>opt</sub> ; Expression<sub>opt</sub>) Statement

1. Let *oldEnv* be the current execution context's LexicalEnvironment.
2. Let *loopEnv* be the result of calling NewDeclarativeEnvironment passing *oldEnv* as the argument.
3. Let *isConst* be the result of performing IsConstantDeclaration of *d*.

**Deleted:** The VarDeclaredNames of the production

**Deleted:** is determined as follows:

**Deleted: Runtime Semantics: Evaluation**

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** Call

**Deleted:** loopContinues

**Comment [AWB651]:** ES5 breaking change Completion reform

**Deleted:** empty

**Deleted:** the first Expression

**Deleted:** present

**Deleted:** the first

**Deleted:** Expression

**Deleted:** ToBoolean(GetValue(*testExprRef*))

**Deleted:** (normal,

**Deleted:** , empty

**Deleted:** loopContinues

**Deleted:** <#>If *stmt.type* is break and *stmt.target* is in the current label set, return (normal, *V*, empty).  
<#>If *stmt.type* is not continue || *stmt.target* is not in the current label set, then||

**Deleted:** is an abrupt completion

**Deleted:** loopContinues

**Deleted:** the second

**Deleted:** Expression

**Deleted:** present

**Deleted:** the second

**Deleted:** Expression

**Deleted:** Call

**Deleted:** (This value is not used.)

**Deleted:** loopContinues

**Deleted:** loopContinues

**Comment [AWB652]:** A final decision has yet been reached on the scoping semantics used for this form of for statement. This version uses "loop scoping" a single set of let/const bindings are used for all iterations of the loop. This is the simplest of the semantics under consideration.

**Deleted:** <#>Evaluate VariableDeclarationListNoIn.¶  
<#>Let *V* = empty.¶  
<#>Repeat¶

<#>If the first Expression is present, then¶  
<#>Let *testExprRef* be the result of evaluating the first Expression.¶  
<#>If ToBoolean(GetValue(*testExprRef*)) is false then return (normal, *V*, empty).¶

**Deleted:** running

**Deleted:** the

4. For each element *dn* of the *BoundNames* of *LexicalDeclarationNoIn* do
  - a. If *isConst* is **true**, then
    - i. Call *loopEnv*'s *CreateImmutableBinding* concrete method passing *dn* as the argument.
  - b. Else,
    - i. Call *loopEnv*'s *CreateMutableBinding* concrete method passing *dn* and **false** as the arguments.
5. Set the *current execution context's LexicalEnvironment* to *loopEnv*.
6. Let *forDcl* be the result of evaluating *LexicalDeclarationNoIn*.
7. If *LoopContinues(varDcl,labelSet)* is **false**, then
  - a. Set the *current execution context's LexicalEnvironment* to *oldEnv*.
  - b. Return *varDcl*.
8. Let *bodyResult* be the result of performing For Body Evaluation with the first *Expression* as the *incrementExpr* argument, the second *Expression* as the *testExpr* argument and with *labelSet*.
9. Set the *current execution context's LexicalEnvironment* to *oldEnv*.
10. Return *bodyResult*.

**Deleted:** running

**Deleted:** loopContinues

**Deleted:** running

**Deleted:** running

#### 12.6.4 The `for-in` and `for-of` Statements

##### Static Semantics

###### Static Semantics: Early Errors

###### IterationStatement :

**for** (*LeftHandSideExpression* **in** *Expression*) *Statement*  
**for** (*LeftHandSideExpression* **of** *Expression*) *Statement*

- It is a Syntax Error if the *IterationStatement* is contained in strict code and *LeftHandSideExpression* is the *Identifier eval* or the *Identifier arguments*.
- It is a Syntax Error if the *LeftHandSideExpression* is an *Identifier* that statically resolves to a declarative environment record binding and the resolved binding is an immutable binding.
- It is a Syntax Error if the *LeftHandSideExpression* is *PrimaryExpression : ( Expression )* and *Expression* derived a production that would produce a Syntax Error according to these rules. This rule is recursively applied.
- It is a Syntax Error if *IsInvalidAssignmentPattern* of *LeftHandSideExpression* is **true**.

**Formatted:** Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

###### IterationStatement :

**for** (*ForDeclaration* **in** *Expression*) *Statement*  
**for** (*ForDeclaration* **of** *Expression*) *Statement*

- It is a Syntax Error if any element of the *LexicallyDeclaredNames* of *ForDeclaration* also occurs in the *Var Declared Names* of *Statement*.

**Deleted:** It is a Syntax Error if the *LeftHandSideExpression* is an *ObjectLiteral* or an *ArrayLiteral* and the source code corresponding to *LeftHandSideExpression* cannot be parsed using *AssignmentPattern* as the goal symbol.

**Deleted:**

**Deleted:**

###### Static Semantics: Bound Names

###### ForDeclaration : *LetOrConst ForBinding*

1. Return the Bound Names of *ForBinding*.

###### Static Semantics: Var Declared Names

###### IterationStatement : **for** (*LeftHandSideExpression* **in** *Expression*) *Statement*

1. Return the Var Declared Names of *Statement*.

###### IterationStatement : **for** (*var ForBinding* **in** *Expression*) *Statement*

1. Let *names* be the Bound Names of *ForBinding*.
2. Append to *names* the elements of the Var Declared Names of *Statement*.
3. Return *names*.

**Deleted:** *ForVarDeclaration : BindingPattern* ¶  
 ¶Return the Bound Names of *BindingPattern*.¶  
*ForVarDeclaration : VariableDeclarationNoIn* ¶  
 ¶Return the Bound Names of *VariableDeclarationNoIn*.¶

**Deleted:** *ForBinding : BindingIdentifier* ¶  
 ¶Return the Bound Names of *BindingIdentifier*.¶  
*ForDeclaration : BindingPattern* ¶  
 ¶Return the Bound Names of *BindingPattern*.¶

**Deleted:** The *VarDeclaredNames* of the production

**Deleted:** is determined as follows:

**Deleted:** *VariableDeclarationNoIn*

**Deleted:** *BindingIdentifier*

**Deleted:** *Variable*

**Deleted:** *BindingIdentifier*

**Deleted:** *DeclarationNoIn*

IterationStatement : `for` ( `ForDeclaration` `in` `Expression` ) `Statement`

1. Return the Var Declared Names of *Statement*.

IterationStatement : `for` ( `LeftHandSideExpression` `of` `Expression` ) `Statement`

1. Return the Var Declared Names of *Statement*.

IterationStatement : `for` ( `var` `ForBinding` `of` `Expression` ) `Statement`

1. Let *names* be the Bound Names of `ForBinding`.
2. Append to *names* the elements of the Var Declared Names of *Statement*.
3. Return *names*

IterationStatement : `for` ( `ForDeclaration` `of` `Expression` ) `Statement`

1. Return the Var Declared Names of *Statement*.

**Runtime Semantics**

**Runtime Semantics: Binding Instantiation**

With arguments *value* and *environment*.

ForDeclaration : `LetOrConst` `ForBinding`

1. For each element *name* of the Bound Names of `ForBinding` do
  - a. If `IsConstantDeclaration` of *LetOrConst* is `false`, then
    - i. Call *environment*'s `CreateMutableBinding` concrete method with argument *name*.
  - b. Else,
    - i. Call *environment*'s `CreateImmutableBinding` concrete method with argument *name*.
2. Perform Binding Initialisation for `ForBinding` passing *value* and *environment* as the arguments.

**Runtime Semantics: Labelled Evaluation**

With argument *labelSet*.

IterationStatement : `for` ( `LeftHandSideExpression` `in` `Expression` ) `Statement`

1. Let *keyResult* be the result of performing For In/Of Expression Evalution with *Statement*, `enumerate`, and *labelSet*.
2. `ReturnIfAbrupt`(*keyResult*).
3. Return the result of performing For In/Of Body Evalution with *LeftHandSideExpression*, *Statement*, *keyResult*, *assignment*, and *labelSet*.

IterationStatement : `for` ( `var` `ForBinding` `in` `Expression` ) `Statement`

1. Let *keyResult* be the result of performing For In/Of Expression Evalution with *Statement*, `enumerate`, and *labelSet*.
2. `ReturnIfAbrupt`(*keyResult*).
3. Return the result of performing For In/Of Body Evalution with `ForBinding`, *Statement*, *keyResult*, *varBinding*, and *labelSet*.

IterationStatement : `for` ( `ForDeclaration` `in` `Expression` ) `Statement`

1. Let *keyResult* be the result of performing For In/Of Expression Evalution with *Statement*, `enumerate`, and *labelSet*.
2. `ReturnIfAbrupt`(*keyResult*).

**Deleted:** `ForVarDeclaration`

**Deleted:** `ForVarDeclaration`

**Deleted:** Runtime Semantics: Binding Initialisation¶

With arguments *value* and *environment*.¶  
NOTE `undefined` is passed for *environment* indicate that a `PutValue` operation should be used to assign the initialisation value. This is the case for `var` statements formal parameter lists of non-strict functions. In those cases a lexical binding is hosted and preinitialised prior to evaluation of its initialiser.

`ForVarDeclaration : BindingPattern` ¶  
¶> Perform Binding Initialisation for `BindingPattern` passing *value* and *env* as the arguments.¶

`ForVarDeclaration : VariableDeclarationNoIn` ¶  
¶> Perform Binding Initialisation for `VariableDeclarationNoIn` passing *value* and *env* as the arguments.¶

`ForBinding : BindingIdentifier` ¶  
¶> Perform Binding Initialisation for `BindingIdentifier` passing *value* and *env* as the arguments.¶

`ForBinding : BindingPattern` ¶  
¶> Perform Binding Initialisation for `BindingPattern` passing *value* and *env* as the arguments.¶

**Deleted:** Runtime Semantics: Evaluation¶

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** .[[value]]

**Deleted:** `BindingIdentifier`

**Deleted:** `BindingIdentifier`

**Deleted:** .[[value]]

3. Return the result of performing For In/Of Body Evaluation with *ForDeclaration*, *Statement*, *keyResult*, *lexicalBinding*, and *labelSet*.

**Deleted:** .[[value]]  
**Deleted:** ForDeclaration

#### *IterationStatement : for ( LeftHandSideExpression of Expression ) Statement*

1. Let *keyResult* be the result of performing For In/Of Expression Evaluation with *Statement*, *iterate*, and *labelSet*.
2. ReturnIfAbrupt(*keyResult*).
3. Return the result of performing For In/Of Body Evaluation with *LeftHandSideExpression*, *Statement*, *keyResult*, *assignment*, and *labelSet*.

**Deleted:** .[[value]]  
**Deleted:** BindingIdentifier

#### *IterationStatement : for ( var ForBinding of Expression ) Statement*

1. Let *keyResult* be the result of performing For In/Of Expression Evaluation with *Expression*, *iterate*, and *labelSet*.
2. ReturnIfAbrupt(*keyResult*).
3. Return the result of performing For In/Of Body Evaluation with *ForBinding*, *Statement*, *keyResult*, *varBinding*, and *labelSet*.

**Deleted:** BindingIdentifier  
**Deleted:** .[[value]]

#### *IterationStatement : for ( ForDeclaration of Expression ) Statement*

1. Let *keyResult* be the result of performing For In/Of Expression Evaluation with *Expression*, *iterate*, and *labelSet*.
2. ReturnIfAbrupt(*keyResult*).
3. Return the result of performing For In/Of Body Evaluation with *ForDeclaration*, *Statement*, *keyResult*, *lexicalBinding*, and *labelSet*.

**Deleted:** Statement  
**Deleted:** .[[value]]  
**Deleted:** ForDeclaration

The abstraction operation For In/Of Expression Evaluation with arguments *expr*, *iterationKind*, and *labelSet* is performed as follows:

1. Let *exprRef* be the result of evaluating the *production* that is *expr*.
2. Let *exprValue* be GetValue(*exprRef*).
3. If *exprValue* is an abrupt completion,
  - a. If *LoopContinues(exprValue.labelSet)* is *false*, then return *exprValue*.
  - b. Else, return Completion{type: *break*, value: *undefined*, target: empty}.
4. If *exprValue.[value]* is *null* or *undefined*, return Completion{type: *break*, value: *undefined*, target: empty}.
5. Let *obj* be ToObject(*exprValue*).
6. If *iterationKind* is *enumerate*, then
  - a. Let *keys* be the result of calling the *[[Enumerate]]* internal method of *obj* with arguments *true* and *true*.
7. Else,
  - a. Assert *iterationKind* is *iterate*.
  - b. Let *keys* be the result of calling the *[[Iterate]]* internal method of *obj*.
8. If *keys* is an abrupt completion, then
  - a. If *LoopContinues(exprValue.labelSet)* is *false*, then return *exprValue*.
  - b. Assert: *keys.[[type]]* is *continue*.
  - c. Return Completion{type: *break*, value: *undefined*, target: empty}.
9. Return *keys*.

**Deleted:** Expression  
**Deleted:** then return  
**Deleted:** loopContinues  
**Comment [AWB653]:** Note a continue in the initialization expression is just like a break  
**Comment [AWB654]:** Completion value reform  
**Comment [AWB655]:** Completion value reform  
**Deleted:**  
**Deleted:** (normal, empty, empty)

The abstraction operation For In/Of Body Evaluation with arguments *lhs*, *stmt*, *keys*, *lhsKind*, and *labelSet* is performed as follows:

1. Let *oldEnv* be the *current* execution context's LexicalEnvironment.
2. Let *V*= *undefined*.
3. Repeat
  - a. Let *next* be the result of performing Invoke with arguments "next", *keys*, and an empty arguments List.
  - b. If *[IteratorComplete](next)* is *true*, then return NormalCompletion(*V*).

**Comment [AWB656]:** Break/continue in the expression works normally (future for do {} or block lambda expressions)  
**Deleted:** LoopContinues  
**Comment [AWB657]:** Note a continue in the initialization expression is just like a break  
**Comment [AWB658]:** Completion value reform  
**Deleted:** running  
**Comment [AWB659]:** Completion value reform  
**Deleted:** empty  
**Comment [AWB760]:** TODO: need to define

- c. If `LoopContinues(next,labelSet)` is `false`, then return `next`.
- d. If `next` is an abrupt completion, then let `status` be `next`.
- e. Else,
  - i. Assert `next.[[type]]` is `normal`.
  - ii. Let `nextValue` be `next.[[value]]`.
  - iii. If `lhsKind` is `assignment`, then
    - 1. Assert: `lhs` is a `LeftHandSideExpression`.
    - 2. If `lhs` is neither an `ObjectLiteral` nor an `ArrayLiteral` then
      - a. Let `lhsRef` be the result of evaluating `lhs` (it may be evaluated repeatedly).
      - b. Let `status` be the result of performing `PutValue(lhsRef, nextValue)`.
    - 3. Else
      - a. Let `AssignmentPattern` be the parse of the source code corresponding to `lhs` using `AssignmentPattern` as the goal symbol.
      - b. Let `rval` be `ToObject(nextValue)`.
      - c. If `rval` is an abrupt completion, then let `status` be `rval`.
      - d. Else, let `status` be the result of performing Destructuring Assignment Evaluation of `AssignmentPattern` using `rval` as the argument.
  - iv. Else if `lhsKind` is `varBinding`, then
    - 1. Assert: `lhs` is a `ForBinding`.
    - 2. Let `status` be the result of performing Binding Initialisation for `lhs` passing `nextValue` and `undefined` as the arguments.
  - v. Else,
    - 1. Asset `lhsKind` is `lexicalBinding`.
    - 2. Assert: `lhs` is a `ForDeclaration`.
    - 3. Let `iterationEnv` be the result of calling `NewDeclarativeEnvironment` passing `oldEnv` as the argument.
    - 4. Perform Binding Instantiation for `lhs` passing `nextValue` and `iterationEnv` as arguments.
    - 5. Let `status` be `NormalCompletion(empty)`
    - 6. Set the `current execution context's LexicalEnvironment` to `iterationEnv`.
  - vi. If `status.[[type]]` is `normal`, then
    - 1. Let `stmt` be the result of evaluating `stmt`.
    - 2. If `status.[[type]]` is `normal` and `status.[[value]]` not `empty`, then
      - a. Let `V = status.[[value]]`.
  - vii. Set the `current execution context's LexicalEnvironment` to `oldEnv`.
  - viii. If `status` is an abrupt completion and `LoopContinues(status,labelSet)` is `false`, then return `status`.

## 12.7 The `continue` Statement

### Syntax

`ContinueStatement :`  
`continue ;`  
`continue [no LineTerminator here] Identifier;`

### Static Semantics

#### Static Semantics: Early Errors

`ContinueStatement : continue ;`

- It is a Syntax Error if this production is not nested, directly or indirectly (but not crossing function boundaries), within an `IterationStatement`.

`ContinueStatement : continue [no LineTerminator here] Identifier ;`

- It is a Syntax Error if `Identifier` does not appear in the `CurrentLabelSet` of an enclosing (but not crossing function boundaries) `IterationStatement`.

**Deleted:** `loopContinues`

**Deleted:** <#>Let `P` be the name of the next property of `obj` whose `[[Enumerable]]` attribute is `true`. If there is no such property, return `(normal, empty)`.

**Deleted:** the `LeftHandSideExpression`

**Deleted:** `Call`

**Deleted:** `P`

**Comment [AWB661]:** ToDo, update this to return a completion value

**Deleted:** `ForDeclaration`

**Deleted:** `running`

**Deleted:** `stmt`

**Deleted:** `Statement`

**Deleted:** If `stmt.value` is not empty, I

**Deleted:** `stmt`

**Deleted:** `running`

**Deleted:** `loopContinues`

**Deleted:** <#>If `stmt.type` is `break` and `stmt.target` is in the current label set, return `(normal, V, empty)`.  
<#>If `stmt.type` is not `continue` or `stmt.target` is in the current label set, then  
<#>If `stmt` is an abrupt completion, return `stmt`.

**Deleted:** The `VarDeclaredNames` of the production

`for ( var VariableDeclarationNoIn in Expression ) Statement .`  
`is determined as follows: ¶`  
`<#>Let names be BoundNames of VariableDeclarationNoIn.¶`  
`<#>Append to names the elements of the VarDeclaredNames of Statement.¶`  
`<#>Return names.¶`

**Deleted:** The production

`IterationStatement : for ( var VariableDeclarationNoIn in Expression ) Statement .`  
`is evaluated as follows: ¶`  
`<#>Let varName be the result of evaluating VariableDeclarationNoIn.¶`  
`<#>Let exprRef be the result of evaluating the Expression.¶`  
`<#>Let exprValue be GetValue(exprRef).¶`  
`<#>If exprValue is null or undefined, return (normal, empty, empty).¶`  
`<#>Let obj be ToObject(exprValue).¶`  
`<#>Let V = empty.¶`  
`<#>Repeat ¶`  
`<#>Let P be the name of the next property of obj whose [[Enumerable]] attribute is true. If there is such property, return (normal, V, empty).¶`

**Deleted:** Semantics ¶

**Deleted:** A program is considered syntactically incorrect if either of the following is true: ¶

**Deleted:** The program contains a `continue` statement without the optional `Identifier`, which

**Deleted:** The program contains a `continue` statement with the optional `Identifier`, where

**Deleted:** label set

## Runtime Semantics

### Runtime Semantics: Evaluation

ContinueStatement : continue ;

1. Return Completion {[[type]]: continue, [[value]]: empty, [[target]]: empty}.

ContinueStatement : continue [no LineTerminator here] Identifier ;

1. Return Completion {[[type]]: continue, [[value]]: empty, [[target]]: Identifier}.

## 12.8 The break Statement

### Syntax

BreakStatement :  
**break** ;  
**break** [no LineTerminator here] Identifier ;

## Static Semantics

### Static Semantics: Early Errors

BreakStatement : break ;

- It is a Syntax Error if this production, not nested, directly or indirectly (but not crossing function boundaries), within an IterationStatement or a SwitchStatement.

BreakStatement : break [no LineTerminator here] Identifier ;

- It is a Syntax Error if Identifier does not appear in the CurrentLabelSet of an enclosing (but not crossing function boundaries) Statement.

## Runtime Semantics

### Runtime Semantics: Evaluation

BreakStatement : break ;

1. Return Completion {[[type]]: break, [[value]]: empty, [[target]]: empty}.

BreakStatement : break [no LineTerminator here] Identifier ;

1. Return Completion {[[type]]: break, [[value]]: empty, [[target]]: Identifier}.

## 12.9 The return Statement

### Syntax

ReturnStatement :  
**return** ;  
**return** [no LineTerminator here] Expression ;

NOTE A return statement causes a function to cease execution and return a value to the caller. If Expression is omitted, the return value is undefined. Otherwise, the return value is the value of Expression.

## Static Semantics

### Static Semantics: Early Errors

**Deleted:** A ContinueStatement without an Identifier is evaluated as follows:  
 ¶

**Deleted:** (

**Deleted:** ).

**Deleted:** A ContinueStatement with the optional Identifier is evaluated as follows:  
 ¶

**Deleted:** (

**Deleted:** ).

**Deleted:** Semantics  
 A program is considered syntactically incorrect if either the following is true:  
 ¶

**Deleted:** The program contains a **break** statement without the optional **Identifier**, which is

**Deleted:** The program contains a **break** statement with the optional **Identifier**, where

**Deleted:** label set

**Deleted:** A BreakStatement without an Identifier is evaluated as follows:  
 ¶

**Deleted:** (

**Deleted:** ).

**Deleted:** A BreakStatement with an Identifier is evaluated as follows:  
 ¶

**Deleted:** (

**Deleted:** ).

- It is a Syntax Error if a `return` statement that is not within a `FunctionBody`.

## Runtime Semantics

### Runtime Semantics: Evaluation

*ReturnStatement* : `return` ;

1. Return `Completion {[[type]]: return, [[value]]: undefined, [[target]]: empty}`.

*ReturnStatement* : `return` [no LineTerminator here] *Expression* ;

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be `GetValue(exprRef)`.
3. If *exprValue* is an abrupt completion, return *exprValue*.
4. Return `Completion {[[type]]: return, [[value]]: exprValue, [[target]]: empty}`.

## 12.10 The `with` Statement

### Syntax

*WithStatement* :  
     `with` ( *Expression* ) *Statement*

**NOTE** The `with` statement adds an object environment record for a computed object to the lexical environment of the current execution context. It then executes a statement using this augmented lexical environment. Finally, it restores the original lexical environment.

### Static Semantics

#### Static Semantics: Early Errors

*WithStatement* : `with` ( *Expression* ) *Statement* ;

- It is a Syntax Error if the code that matches this production is contained in strict code.

#### Static Semantics: VarDeclaredNames

*WithStatement* : `with` ( *Expression* ) *Statement* ;

1. Return the `VarDeclaredNames` of *Statement*.

### Runtime Semantics

#### Runtime Semantics: Evaluation

*WithStatement* : `with` ( *Expression* ) *Statement* ;

1. Let *val* be the result of evaluating *Expression*.
2. Let *obj* be `ToObject(GetValue(val))`.
3. If *obj* is an abrupt completion, return *obj*.
4. Let *oldEnv* be the `current` execution context's `LexicalEnvironment`.
5. Let *newEnv* be the result of calling `NewObjectEnvironment` passing *obj* and *oldEnv* as the arguments.
6. Set the `withEnvironment` flag of *newEnv* to `true`.
7. Set the `current` execution context's `LexicalEnvironment` to *newEnv*.
8. Let *C* be the result of evaluating *Statement*.
9. Set the `current` execution context's `Lexical Environment` to *oldEnv*.
10. Return *C*.

**Deleted:** <#>Semantics¶

**Deleted:** A `return` statement causes a function to cease execution and return a value to the caller. If *Expression* is omitted, the return value is `undefined`. Otherwise, the return value is the value of *Expression*.

**Formatted:** Bulleted + Level: 1 + Aligned at 0.25" + Indent at: 0.5"

**Deleted:** An ECMAScript program is considered syntactically incorrect if it contains

**Deleted:** A `ReturnStatement` is evaluated as follows:  
If the *Expression* is not present, r

**Deleted:** (

**Deleted:** ).

**Deleted:** (

**Deleted:** `GetValue`(

**Deleted:** *exprRef*)

**Deleted:** ).

**Deleted:** Semantics¶  
The static semantics of the production

**Deleted:** are:

**Formatted:** Bulleted + Level: 1 + Aligned at 0.25" + Indent at: 0.5"

**Deleted:** static

**Deleted:** The `VarDeclaredNames` of the production

**Deleted:** is determined as follows:

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** running

**Deleted:** provideThis

**Deleted:** running

**Deleted:** but if an exception is thrown during evaluation, let *C* be `(throw, V, empty)`, where *V* is the exception. (Execution now proceeds as if no exception were thrown.)

**Deleted:** running

**NOTE** No matter how control leaves the embedded *Statement*, whether normally or by some form of abrupt completion or exception, the *LexicalEnvironment* is always restored to its former state.

## 12.11 The `switch` Statement

### Syntax

*SwitchStatement* :  
`switch ( Expression ) CaseBlock`

*CaseBlock* :  
`{ CaseClausesopt }`  
`{ CaseClausesopt DefaultClause CaseClausesopt }`

*CaseClauses* :  
`CaseClause`  
`CaseClauses CaseClause`

*CaseClause* :  
`case Expression : StatementListopt`

*DefaultClause* :  
`default : StatementListopt`

### Static Semantics

#### Static Semantics: Early Errors

*CaseBlock* : `{ CaseClauses }`

- It is a Syntax Error if the *LexicallyDeclaredNames* of *CaseClauses* contains any duplicate entries.
- It is a Syntax Error if any element of the *LexicallyDeclaredNames* of *CaseClauses* also occurs in the *VarDeclaredNames* of *CaseClauses*.

**Deleted:** 12.10.1 Strict Mode Restrictions  
 Strict mode code may not include a *WithStatement*. The occurrence of a *WithStatement* in such a context is treated as a **SyntaxError**.

#### Static Semantics: LexicalDeclarations

*CaseBlock* : `{ }`

1. Return a new empty List.

*CaseBlock* : `{ CaseClauses }`

1. Return the *LexicalDeclarations* of *CaseClauses*.

*CaseBlock* : `{ CaseClausesopt DefaultClause CaseClausesopt }`

1. If the first *CaseClauses* is present, let *declarations* be the *LexicalDeclarations* of the first *CaseClauses*.
2. Else let *declarations* be a new empty List.
3. Append to *declarations* the elements of the *LexicallyDeclaredNames* of the *DefaultClause*.
4. If the second *CaseClauses* is not present, return *names*.
5. Else return the result of appending to *names* the elements of the *LexicallyDeclaredNames* of the second *CaseClauses*.

**Deleted:** are:

**Formatted:** Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

*CaseClauses* : *CaseClause*

1. Return the *LexicalDeclarations* of *CaseClause*.

*CaseClauses* : *CaseClauses CaseClause*

1. Let *declarations* be *LexicalDeclarations* of *CaseClauses*.

**Deleted:** *CaseClause* : `case Expression : StatementList`  
 <#>It is a Syntax Error if *StatementList* includes a *StatementListItem* : *FunctionDeclaration* production and the source code matching this *CaseClause* production is not contained in extended code.

**Deleted:** the

2. Append to *names* the elements of the LexicalDeclarations of *CaseClause*.
3. Return *declarations*.

*CaseClause* : **case** *Expression* : *StatementList*<sub>opt</sub>

1. If the *StatementList* is present, return the LexicalDeclarations of *StatementList*.
2. Else return a new empty List.

*DefaultClause* : **default** : *StatementList*<sub>opt</sub>

1. If the *StatementList* is present, return the LexicalDeclarations of *StatementList*.
2. Else return a new empty List.

#### **Static Semantics:** LexicallyDeclaredNames

*CaseBlock* : { }

1. Return a new empty List.

*CaseBlock* : { *CaseClauses* }

1. Return the LexicallyDeclaredNames of *CaseClauses*.

*CaseBlock* : { *CaseClauses*<sub>opt</sub> *DefaultClause* *CaseClauses*<sub>opt</sub> }

1. If the first *CaseClauses* is present, let *names* be the LexicallyDeclaredNames of the first *CaseClauses*.
2. Else let *names* be a new empty List.
3. Append to *names* the elements of the LexicallyDeclaredNames of the *DefaultClause*.
4. If the second *CaseClauses* is not present, return *names*.
5. Else return the result of appending to *names* the elements of the LexicallyDeclaredNames of the second *CaseClauses*.

**Deleted:** the

*CaseClauses* : *CaseClause*

1. Return the LexicallyDeclaredNames of *CaseClause*.

*CaseClauses* : *CaseClauses CaseClause*

1. Let *names* be LexicallyDeclaredNames of *CaseClauses*.
2. Append to *names* the elements of the LexicallyDeclaredNames of *CaseClause*.
3. Return *names*.

*CaseClause* : **case** *Expression* : *StatementList*<sub>opt</sub>

1. If the *StatementList* is present, return the LexicallyDeclaredNames of *StatementList*.
2. Else return a new empty List.

*DefaultClause* : **default** : *StatementList*<sub>opt</sub>

1. If the *StatementList* is present, return the LexicallyDeclaredNames of *StatementList*.
2. Else return a new empty List.

#### **Static Semantics:** VarDeclaredNames

*SwitchStatement* : **switch** ( *Expression* ) *CaseBlock*

**Deleted:** Semantics  
The VarDeclaredNames of the production

1. Return the VarDeclaredNames of *CaseBlock*.

**Deleted:** is determined as follows:

*CaseBlock* : { }

1. Return a new empty List.

CaseBlock : { CaseClauses }

1. Return the VarDeclaredNames of CaseClauses.

CaseBlock : { CaseClauses<sub>opt</sub> DefaultClause CaseClauses<sub>opt</sub> }

1. If the first CaseClauses is present, let names be the VarDeclaredNames of the first CaseClauses.
2. Else let names be a new empty List.
3. Append to names the elements of the VarDeclaredNames of the DefaultClause.
4. If the second CaseClauses is not present, return names.
5. Else return the result of appending to names the elements of the VarDeclaredNames of the second CaseClauses.

**Deleted:** the

CaseClauses : CaseClause

1. Return the VarDeclaredNames of CaseClause.

CaseClauses : CaseClauses CaseClause

1. Let names be VarDeclaredNames of CaseClauses.
2. Append to names the elements of the VarDeclaredNames of CaseClause.
3. Return names.

CaseClause : case Expression : StatementList<sub>opt</sub>

1. If the StatementList is present, return the VarDeclaredNames of StatementList.
2. Else return a new empty List.

DefaultClause : default : StatementList<sub>opt</sub>

1. If the StatementList is present, return the VarDeclaredNames of StatementList.
2. Else return a new empty List.

## Runtime Semantics

### Runtime Semantics: Case Evaluation

With argument input.

CaseBlock : { CaseClauses<sub>opt</sub> }

1. Let  $V = \text{undefined}$ .
2. Let  $A$  be the list of CaseClause items in source text order.
3. Let  $\text{searching} = \text{true}$ .
4. Repeat, while  $\text{searching}$  is true
  - a. Let  $C$  be the next CaseClause in  $A$ . If there is no such CaseClause, return NormalCompletion( $V$ ).
  - b. Let  $\text{clauseSelector}$  be the result of evaluating  $C$ .
    - c. If  $\text{clauseSelector}$  is an abrupt completion, return clauseSelector.
    - d. If  $\text{input}$  is equal to  $\text{clauseSelector}$  as defined by the Strict Equality Comparison Algorithm (11.9.1), then
      - i. Set  $\text{searching}$  to false.
      - ii. If  $C$  has a StatementList, then
        1. Evaluate  $C$ 's StatementList and let  $R$  be the result.
        2. If  $R$  is an abrupt completion, then return  $R$ .
        3. Let  $V = R[\text{value}]$ .
5. Repeat
  - a. Let  $C$  be the next CaseClause in  $A$ . If there is no such CaseClause, return NormalCompletion( $V$ ).
  - b. If  $C$  has a StatementList, then

**Deleted: Runtime Semantics: Evaluation**  
 The production SwitchStatement : switch ( Expression ) CaseBlock is evaluated as follows:  
 <#>Let  $\text{exprRef}$  be the result of evaluating Expression.  
 <#>Let  $\text{oldEnv}$  be the running execution context's LexicalEnvironment.  
 <#>Let  $\text{blockEnv}$  be the result of calling NewDeclarativeEnvironment passing  $\text{oldEnv}$  as the argument.  
 <#>Perform Block Declaration Instantiation using CaseBlock and  $\text{blockEnv}$ .  
 <#>Let  $R$  be the result of evaluating CaseBlock, passing it GetValue(exprRef) as a parameter.  
 <#>Set the running execution context's LexicalEnvironment to  $\text{oldEnv}$ .  
 <#>If  $R$ .type is break and  $R$ .target is in the current label set, return (normal,  $R$ .value, empty).  
 <#>Return  $R$ .  
 NOTE: No matter how control leaves the SwitchStatement, the LexicalEnvironment is always restored to its former state.

**Formatted: Highlight**

**Deleted:** The production

**Deleted:** is given an input parameter,  $\text{input}$ , and is evaluated as follows:

**Comment [AWB662]:** ES5 breaking change: completion reform

**Deleted:** empty

**Deleted:** (normal,

**Deleted:** , empty

**Deleted:** 6

**Deleted:** === operator

**Deleted:** (normal,

**Deleted:** , empty

- i. Evaluate  $C$ 's *StatementList* and let  $R$  be the result.
- ii. If  $R.[[value]]$  is not **empty**, then let  $V = R.[[value]]$ .
- iii. If  $R$  is an abrupt completion, then return *Completion*  $\{[[type]]: R.[[type]], [[value]]: V, [[target]]: R.[[target]]\}$ .

*CaseBlock* : { *CaseClauses*<sub>opt</sub> *DefaultClause* *CaseClauses*<sub>opt</sub> }

1. Let  $V = \text{undefined}$ .
2. Let  $A$  be the list of *CaseClause* items in the first *CaseClauses*, in source text order.
3. Let  $B$  be the list of *CaseClause* items in the second *CaseClauses*, in source text order.
4. Let *found* be **false**.
5. Repeat letting  $C$  be in order each *CaseClause* in  $A$ 
  - a. If *found* is **false**, then
    - i. Let *clauseSelector* be the result of evaluating  $C$ .
    - ii. If *clauseSelector* is an abrupt completion, then
      1. If *clauseSelector*[*value*] is empty, then return *Completion*  $\{[[type]]: clauseSelector.[[type]], [[value]]: \text{undefined}, [[target]]: clauseSelector.[[target]]\}$ .
      2. Else, return *clauseSelector*.
    - iii. If *input* is equal to *clauseSelector* as defined by the Strict Equality Comparison Algorithm (11.9.1), then set *found* to **true**.
  - b. If *found* is **true**, then
    - i. If  $C$  has a *StatementList*, then
      1. Evaluate  $C$ 's *StatementList* and let  $R$  be the result.
      2. If  $R.[[value]]$  is not **empty**, then let  $V = R.[[value]]$ .
      3. If  $R$  is an abrupt completion, then return *Completion*  $\{[[type]]: R.[[type]], [[value]]: V, [[target]]: R.[[target]]\}$ .
6. Let *foundInB* be **false**.
7. If *found* is **false**, then
  - a. Repeat, while *foundInB* is **false** and all elements of  $B$  have not been processed
    - i. Let  $C$  be the next *CaseClause* in  $B$ .
    - ii. Let *clauseSelector* be the result of evaluating  $C$ .
    - iii. If *clauseSelector* is an abrupt completion, then
      1. If *clauseSelector*[*value*] is empty, then return *Completion*  $\{[[type]]: clauseSelector.[[type]], [[value]]: \text{undefined}, [[target]]: clauseSelector.[[target]]\}$ .
      2. Else, return *clauseSelector*.
    - iv. If *input* is equal to *clauseSelector* as defined by the Strict Equality Comparison Algorithm (11.9.1), then
      1. Set *foundInB* to **true**.
      2. If  $C$  has a *StatementList*, then
        - a. Evaluate  $C$ 's *StatementList* and let  $R$  be the result.
        - b. If  $R.[[value]]$  is not **empty**, then let  $V = R.[[value]]$ .
        - c. If  $R$  is an abrupt completion, then return *Completion*  $\{[[type]]: R.[[type]], [[value]]: V, [[target]]: R.[[target]]\}$ .
  - b. If *foundInB* is **false** and the *DefaultClause* has a *StatementList*, then
    - a. Evaluate the *DefaultClause*'s *StatementList* and let  $R$  be the result.
    - b. If  $R.[[value]]$  is not **empty**, then let  $V = R.[[value]]$ .
    - c. If  $R$  is an abrupt completion, then return *Completion*  $\{[[type]]: R.[[type]], [[value]]: V, [[target]]: R.[[target]]\}$ .
9. Repeat (Note that if step 7.a.i has been performed this loop does not start at the beginning of  $B$ )
  - a. Let  $C$  be the next *CaseClause* in  $B$ . If there is no such *CaseClause*, return *NormalCompletion*.
  - b. If  $C$  has a *StatementList*, then
    - i. Evaluate  $C$ 's *StatementList* and let  $R$  be the result.
    - ii. If  $R.[[value]]$  is not **empty**, then let  $V = R.[[value]]$ .
    - iii. If  $R$  is an abrupt completion, then *Completion*  $\{[[type]]: R.[[type]], [[value]]: V, [[target]]: R.[[target]]\}$ .

#### Runtime Semantics: Evaluation

*SwitchStatement* : **switch** ( *Expression* ) *CaseBlock*

**Deleted:** ( )

**Deleted:** The production

**Deleted:** is given an input parameter, *input*, and is evaluated as follows:

**Comment [AWB663]:** ES5 breaking change completion reform

**Deleted:** empty

**Deleted:** 6

**Deleted:** === operator

**Deleted:** ( )

**Deleted:** ).

**Deleted:** 6

**Deleted:** === operator

**Deleted:** ( )

**Deleted:** ).

**Deleted:** (*R.type*, *V*, *R.target*)

**Deleted:** (normal,

**Deleted:** , empty)

**Deleted:** return (*R.type*, *V*, *R.target*)

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *switchValue* be *GetValue(exprRef)*.
3. If *switchValue* is an abrupt completion, return *switchValue*.
4. Let *oldEnv* be the *current execution context's LexicalEnvironment*.
5. Let *blockEnv* be the result of calling *NewDeclarativeEnvironment* passing *oldEnv* as the argument.
6. Perform *Block Declaration Instantiation* using *CaseBlock* and *blockEnv*.
7. Let *R* be the result of performing *Case Evaluation CaseBlock* with argument *switchValue*.
8. Set the *current execution context's LexicalEnvironment* to *oldEnv*.
9. Return *R*.

**NOTE** No matter how control leaves the *SwitchStatement* the *LexicalEnvironment* is always restored to its former state.

*CaseClause* : **case** *Expression* : *StatementList*<sub>opt</sub>

1. Let *exprRef* be the result of evaluating *Expression*.
2. Return *GetValue(exprRef)*.

**NOTE** Evaluating *CaseClause* does not execute the associated *StatementList*. It simply evaluates the *Expression* and returns the value, which the *CaseBlock* algorithm uses to determine which *StatementList* to start executing.

## 12.12 Labelled Statements

### Syntax

*LabelledStatement* :  
    *Identifier* : *Statement*

**NOTE** A *Statement* may be prefixed by a label. Labelled statements are only used in conjunction with labelled *break* and *continue* statements. ECMAScript has no *goto* statement. A *Statement* can be part of a *LabelledStatement*, which itself can be part of a *LabelledStatement*, and so on. The labels introduced this way are collectively referred to as the "current label set" when describing the semantics of individual statements. A *LabelledStatement* has no semantic meaning other than the introduction of a label to a label set. The label set of an *IterationStatement* or a *SwitchStatement* initially contains the single element *empty*. The label set of any other statement is initially empty.

### Static Semantics

#### Static Semantics: Early Errors

- It is a Syntax Error if a *LabelledStatement* is enclosed by a *LabelledStatement* with the same *Identifier* as label. This does not apply to labels appearing within the body of a *FunctionDeclaration* that is nested, directly or indirectly, within a labelled statement.

#### Static Semantics: VarDeclaredNames

*LabelledStatement* : *Identifier* : *Statement*

1. Return the *VarDeclaredNames* of *Statement*.

### Runtime Semantics

#### Runtime Semantics: Labelled Evaluation

With argument *labelSet*.

*LabelledStatement* : *Identifier* : *Statement*

1. Let *label* be the *StringValue* of *Identifier*.
2. Let *newLabelSet* be a new List containing *label* and the elements of *labelSet*.

**Deleted:** running

**Deleted:** running

**Deleted:** <#> If *R*<sub>[[type]]</sub> is *break* and *R*<sub>[[target]]</sub> is in *CurrentLabelSet*, return *NormalCompletion(R*<sub>[[value]]</sub>)<sub>¶</sub>

**Deleted:** The production

**Deleted:** is evaluated as follows:

**Deleted:** Semantics¶

**Deleted:** An ECMAScript program is considered syntactically incorrect if it contains

**Deleted:** that

**Deleted:** The *VarDeclaredNames* of the production *DefaultCase*

**Deleted:** *t* is determined as follows:

3. If Statement is either *LabelledStatement* or *BreakableStatement*, then
  - a. Let *stmtResult* be the result of performing Labelled Evaluation of *Statement* with argument *newLabelSet*.
4. Else,
  - a. Let *stmtResult* be the result of evaluating *Statement*.
5. If *stmtResult*.[[type]] is **break** and *stmtResult*.[[target]] is the same value as *label*, then
  - a. Let *result* be *NormalCompletion*(*stmtResult*.[[value]]).
6. Else, let *result* be *stmtResult*.
7. Return *result*.

#### Runtime Semantics: Evaluation

*LabelledStatement* : *Identifier* : *Statement*

3. Let *newLabelSet* be a new empty List.
4. Return the result of performing Labelled Evaluation of this *LabelledStatement* with argument *newLabelSet*.

### **12.13 The throw Statement**

#### Syntax

*ThrowStatement* :
   
    **throw** [no LineTerminator here] *Expression* ;

#### Semantics

The production *ThrowStatement* : **throw** [no LineTerminator here] *Expression* ; is evaluated as follows:

1. Let *exprRef* be the result of evaluating *Expression*.
2. Return (**throw**, *GetValue*(*exprRef*), *empty*).

### **12.14 The try Statement**

#### Syntax

*TryStatement* :
   
    **try** *Block* *Catch*
  
    **try** *Block* *Finally*
  
    **try** *Block* *Catch* *Finally*

*Catch* :
   
    **catch** ( *CatchParameter* ) *Block*

*Finally* :
   
    **finally** *Block*

*CatchParameter* :
   
    *BindingIdentifier*
  
    *BindingPattern*

**NOTE** The **try** statement encloses a block of code in which an exceptional condition can occur, such as a runtime error or a **throw** statement. The **catch** clause provides the exception-handling code. When a catch clause catches an exception, its *Identifier* is bound to that exception.

**Deleted:** The production *Identifier* : *Statement* evaluated by adding *Identifier* to the label set of *Statement* and then evaluating *Statement*. If the *LabelledStatement* itself has a non-empty label set, these labels are also added to the label set of *Statement* before evaluating it. If the result of evaluating *Statement* is (**break**, *V*, *L*) where *L* is equal to *Identifier*, the production results in (*normal*, *V*, *empty*). ¶

**Deleted:** Prior to the evaluation of a *LabelledStatement*, the contained *Statement* is regarded as possessing an empty label set, unless it is an *IterationStatement* or a *SwitchStatement*, in which case it is regarded as possessing a label set consisting of the single element, *empty*. ¶

**Deleted:** *Identifier*

**Formatted:** Note

#### Static Semantics

##### Static Semantics: Early Errors

*Catch* : **catch** ( *CatchParameter* ) *Block*

- It is a Syntax Error if any element of the BoundNames of *CatchParameter* also occurs in the LexicallyDeclaredNames of *Block*.
- It is a Syntax Error if any element of the BoundNames of *CatchParameter* also occurs in the VarDeclaredNames of *Block*.

**Formatted:** Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

**Deleted:** the code that matches this production is contained in extended code and

**Comment [AWB64]:** Note that this is a new restriction that does not exist in ES5

### Static Semantics: BoundNames

*CatchParameter*: *BindingIdentifier*

1. Return the BoundNames of *BindingIdentifier*.

*CatchParameter*: *BindingPattern*

1. Return the BoundNames of *BindingPattern*.

### Static Semantics: VarDeclaredNames

*TryStatement*: *try Block Catch*

1. Let *names* be VarDeclaredNames of *Block*.
2. Append to *names* the elements of the VarDeclaredNames of *Catch*.
3. Return *names*.

**Deleted:** Semantics¶  
The VarDeclaredNames of the production

**Deleted:** is determined as follows:

*TryStatement*: *try Block Finally*

1. Let *names* be VarDeclaredNames of *Block*.
2. Append to *names* the elements of the VarDeclaredNames of *Finally*.
3. Return *names*.

*TryStatement*: *try Block Catch Finally*

1. Let *names* be VarDeclaredNames of *Block*.
2. Append to *names* the elements of the VarDeclaredNames of *Catch*.
3. Append to *names* the elements of the VarDeclaredNames of *Finally*.
4. Return *names*.

*Catch*: *catch ( CatchParameter ) Block*

1. Return the VarDeclaredNames of *Block*.

*Finally*: *finally Block*

1. Return the VarDeclaredNames of *Block*.

### Runtime Semantics

**Runtime Semantics: BindingInitialisation**

With arguments *value* and *environment*.

NOTE *undefined* is passed for *environment* to indicate that a PutValue operation should be used to assign the initialisation value. This is the case for *var* statements formal parameter lists of non-strict functions. In those cases a lexical binding is hosted and preinitialized prior to evaluation of its initializer.

**Deleted:** Initialization

**Deleted:** environment

**Deleted:** initialization

**Deleted:** the

*CatchParameter*: *BindingIdentifier*

1. Return the result of performing *BindingInitialisation* for *BindingIdentifier* passing *value* and *environment* as the arguments.

**Deleted:** Perform

**Deleted:** Initialization

**Deleted:** environment

### CatchParameter : BindingPattern

1. Let *exceptionObj* be *ToObject(value)*.
2. Return the result of performing *BindingInitialisation* for *BindingPattern* passing *exceptionObj* and *environment* as the arguments.

### Runtime Semantics: Catch Clause Evaluation

with parameter *thrownValue*

#### Catch : catch ( *CatchParameter* ) *Block*

1. Let *oldEnv* be the *current execution context's LexicalEnvironment*.
2. Let *catchEnv* be the result of calling *NewDeclarativeEnvironment* passing *oldEnv* as the argument.
3. For each element *argName* of the *BoundNames* of *CatchParameter*, do
  - a. Call the *CreateMutableBinding* concrete method of *catchEnv* passing *argName* as the argument.
4. Let *status* be the result of performing *BindingInitialisation* for *CatchParameter* passing *thrownValue* and *catchEnv* as arguments.
5. If *status* is an abrupt completion, then return *status*.
6. Set the *current execution context's LexicalEnvironment* to *catchEnv*.
7. Let *B* be the result of evaluating *Block*.
8. Set the *current execution context's LexicalEnvironment* to *oldEnv*.
9. Return *B*.

**NOTE** No matter how control leaves the *Block* the *LexicalEnvironment* is always restored to its former state.

### Runtime Semantics: Evaluation

#### TryStatement : try *Block* *Catch*

1. Let *B* be the result of evaluating *Block*.
2. If *B*.[type] is not *throw*, return *B*.
3. Return the result of performing Catch Clause Evaluation of Catch with parameter *B*.[value].

#### TryStatement : try *Block* Finally

1. Let *B* be the result of evaluating *Block*.
2. Let *F* be the result of evaluating *Finally*.
3. If *F*.[type] is *normal*, return *B*.
4. Return *F*.

#### TryStatement : try *Block* *Catch* Finally

1. Let *B* be the result of evaluating *Block*.
2. If *B*.[type] is *throw*, then
  - a. Let *C* be the result of performing Catch Clause Evaluation of Catch with parameter *B*.value.
3. Else, *B*.[type] is not *throw*,
  - a. Let *C* be *B*.
4. Let *F* be the result of evaluating *Finally*.
5. If *F*.[type] is *normal*, return *C*.
6. Return *F*.

#### Finally : finally *Block*

1. Return the result of evaluating *Block*.

<b>Deleted:</b> Perform
<b>Deleted:</b> Initialization
<b>Deleted:</b> environment
<b>Deleted:</b> running
<b>Deleted:</b> Perform
<b>Deleted:</b> Initialization
<b>Deleted:</b> g
<b>Deleted:</b> an exception was thrown in the preceding step, return ( <i>throw</i> , <i>V</i> , <i>empty</i> ) where <i>V</i> is the exception. (Execution now proceeds as if no exception were thrown.)
<b>Deleted:</b> running
<b>Deleted:</b> running
<b>Deleted:</b> The production
<b>Deleted:</b> is evaluated as follows:
<b>Deleted:</b> evaluating
<b>Deleted:</b> The production
<b>Deleted:</b> is evaluated as follows:
<b>Deleted:</b> The production
<b>Deleted:</b> is evaluated as follows:
<b>Deleted:</b> evaluating
<b>Deleted:</b> The production <i>Catch : catch ( IdentifierCatchParameter ) Block</i> is evaluated with parameter <i>C</i> as follows: <#> Let <i>C</i> be the parameter that has been passed to this production. <#> Let <i>oldEnv</i> be the running execution context's LexicalEnvironment. <#> Let <i>catchEnv</i> be the result of calling <i>NewDeclarativeEnvironment</i> passing <i>oldEnv</i> as the argument. <#> For each element <i>argName</i> of the <i>BoundNames</i> of <i>CatchParameter</i> , do <#> Call the <i>CreateMutableBinding</i> concrete method of <i>catchEnv</i> passing <i>argName</i> the <i>Identifier</i> String value as the argument. <#> Call the <i>evaluate</i> <i>CatchParameter</i> <i>SetMutableBinding</i> using concrete method of <i>catchEnv</i> passing the <i>Identifier</i> , <i>C</i> and <i>catEnv</i> , and <i>false</i> as arguments. Note that the last argument is immaterial in this situation. <#> If an exception was thrown in the preceding step, return ( <i>throw</i> , <i>V</i> , <i>empty</i> ) where <i>V</i> is the exception. (Execution now proceeds as if no exception were thrown.) <#> Set the running execution context's LexicalEnvironment to <i>catchEnv</i> . <#> Let <i>B</i> be the result of evaluating <i>Block</i> . <#> Set the running execution context's LexicalEnvironment to <i>oldEnv</i> . <#> Return <i>B</i> . <b>NOTE</b> No matter how control leaves the <i>Block</i> the <i>LexicalEnvironment</i> is always restored to its former state. The <i>VarDeclaredNames</i> of the production <i>Finally : finally Block</i> is determined as follows: <#> Return the <i>VarDeclaredNames</i> of <i>Block</i> . The production <b>Deleted:</b> is evaluated as follows:

## 12.15 The debugger statement

### Syntax

*DebuggerStatement* :  
**debugger** ;

### Semantics

**NOTE** Evaluating the *DebuggerStatement* production may allow an implementation to cause a breakpoint when run under a debugger. If a debugger is not present or active this statement has no observable effect.

### Runtime Semantics: Evaluation

The production *DebuggerStatement* : **debugger** ; is evaluated as follows:

1. If an implementation defined debugging facility is available and enabled, then
  - a. Perform an implementation defined debugging action.
  - b. Let *result* be an implementation defined Completion value.
2. Else
  - a. Let *result* be NormalCompletion(empty).
3. Return *result*.

## 13 Functions and Generators

### 13.1 Function Definitions

#### Syntax

*FunctionDeclaration* :  
**function** *BindingIdentifier* (*FormalParameterList*) { *FunctionBody* }

**Deleted:** (normal,

**Deleted:** , empty

*FunctionExpression* :  
**function** *BindingIdentifier*<sub>opt</sub> (*FormalParameterList*) { *FunctionBody* }

**Deleted:** opt

*FormalParameterList* :  
[empty]  
*FunctionRestParameter*  
*FormalsList*  
*FormalsList*, *FunctionRestParameter*

**Deleted:** ... Identifier

**Deleted:** ... Identifier

*FormalsList* :  
*FormalParameter*  
*FormalsList*, *FormalParameter*

**Deleted:** FormalParameterList

**Deleted:** Identifier

**Deleted:** FormalParameterList

**Deleted:** Identifier

*FunctionRestParameter* :  
 ... *BindingIdentifier*

*FormalParameter* :  
*BindingElement*

**Deleted:** BindingIdentifier Initialiser<sub>opt</sub>

**Deleted:** Pattern

**Deleted:** Initialiser<sub>opt</sub>

**Deleted:** SourceElements<sub>opt</sub>

*FunctionBody* :  
*StatementList*<sub>opt</sub>

**Deleted:** NOTE . The identifier of a rest parameter is specified as an *Identifier* rather than a *BindingIdentifier* because it may be the identifier *arguments..*

#### Static Semantics

##### Static Semantics: Early Errors

FunctionDeclaration : function BindingIdentifier ( FormalParameterList ) { FunctionBody }  
 and  
FunctionExpression : function BindingIdentifier<sub>opt</sub> ( FormalParameterList ) { FunctionBody }

- It is a Syntax Error if any element of the BoundNames of FormalParameterList also occurs in the LexicallyDeclaredNames of FunctionBody.
- It is a Syntax Error if FunctionBody Contains YieldExpression.

FunctionBody : StatementList

- It is a Syntax Error if the LexicallyDeclaredNames of StatementList contains any duplicate entries.
- It is a Syntax Error if any element of the LexicallyDeclaredNames of StatementList also occurs in the VarDeclaredNames of StatementList.

FormalParameterList : FormalsList

- It is a Syntax Error if BoundNames of FormalsList contains any duplicate elements and either FormalsList Contains BindingPattern or FormalsList Contains Initialiser.
- It is a Syntax Error if BoundNames of FormalsList contains either "eval" or "arguments" and either FormalsList Contains BindingPattern or FormalsList Contains Initialiser.

FormalParameterList : FormalsList , FunctionRestParameter

- It is a Syntax Error if BoundNames of FormalsList contains any duplicate elements.
- It is a Syntax Error if BoundNames of FormalsList contains any element of the BoundNames of FunctionRestParameter.

FormalsList : FormalsList , FormalParameter

- It is a Syntax Error if the source code matching this production is strict code and BoundNames of FormalsList contains any element which is also contained in BoundNames of FormalParameter.

FunctionRestParameter : ... BindingIdentifier

- It is a Syntax Error if the String Value of the sole element of the BoundNames of BindingIdentifier is eval or arguments.

FormalParameter : BindingElement

- It is a Syntax Error if BoundNames of BindingElement contains any duplicate elements.

### Static Semantics: BoundNames

FunctionDeclaration : function BindingIdentifier ( FormalParameterList ) { FunctionBody }

1. Return the BoundNames of BindingIdentifier.

FormalParameterList : [empty]

1. Return an empty List.

FormalParameterList : FunctionRestParameter

1. Return the BoundNames of FunctionRestParameter.

**Deleted:** Semantics¶

**Deleted:** <#>It is a Syntax Error if the source code matching this production is extended code and the BoundNames of BindingIdentifier also occurs in the VarDeclaredNames of FunctionBody.  
 <>It is a Syntax Error if the source code matching this production is extended code and the BoundNames of BindingIdentifier also occurs in the LexicallyDeclaredNames of FunctionBody.

**Deleted:** <#>It is a Syntax Error if the source code matching this production is extended code and any element of the LexicallyDeclaredNames of FormalParameterList also occurs in the VarDeclaredNames of FunctionBody.

**Deleted:** the source code matching this production is extended code and

**Formatted:** Bulleted + Level: 1 + Aligned at 0.25" + Indent at: 0.5"

**Comment [AWB665]:** There is an issue that function declarations have "lexically declarative names" and in ES<=5 nested functions are allowed to have the same name as vars or formal parameters. We either need to take this a breaking change or adjust this rule to only apply to function declarations nested within blocks.

**Deleted:** the source code matching this production is extended code and

**Deleted:** BoundNames

**Formatted:** Bulleted + Level: 1 + Aligned at 0.25" + Indent at: 0.5"

**Deleted:** the source code matching this production is extended code and

**Comment [AWB666]:** Similar function declaration ES5 compat issues.

**Formatted:** Don't add space between paragraphs of the same style, Bulleted + Level 1 + Aligned at: 0.25" + Indent at: 0.5"

**Formatted:** Bulleted + Level: 1 + Aligned at 0.25" + Indent at: 0.5"

**Formatted:** Font: Bold

**Formatted:** Bulleted + Level: 1 + Aligned at 0.25" + Indent at: 0.5"

**Deleted:** the

**Deleted:** String Value

**Deleted:** Identifier

**Formatted:** Bulleted + Level: 1 + Aligned at 0.25" + Indent at: 0.5"

**Formatted:** Bulleted + Level: 1 + Aligned at 0.25" + Indent at: 0.5"

**Deleted:** <#>It is a Syntax Error if the source code parsed with this production is not extended code.  
 <>It is a Syntax Error if Identifier is "eval"¶

**Deleted:** <#>It is a Syntax Error if the derivation of BindingElement is BindingIdentifier

**Formatted:** Bulleted + Level: 1 + Aligned at 0.25" + Indent at: 0.5"

FormalParameterList : FormalsList

1. Return the BoundNames of *FormalsList*.

FormalParameterList : FormalsList , FunctionRestParameter

1. Let *names* be BoundNames of *FormalsList*.  
 2. Append to *names* the BoundNames of *FunctionRestParameter*.  
 3. Return *names*.

FormalsList : FormalsList , FormalParameter

1. Let *names* be BoundNames of *FormalsList*.  
 2. Append to *names* the elements of BoundNames of *FormalParameter*.  
 3. Return *names*.

FormalParameterList : ... BindingIdentifier

1. Return the BoundNames of *BindingIdentifier*.

FormalParameter : BindingElement

1. Return the BoundNames of *BindingElement*.

**Static Semantics: Contains**

With parameter *symbol*.

FunctionDeclaration : function BindingIdentifier ( FormalParameterList ) { FunctionBody }

1. Return false.

FunctionExpression : function BindingIdentifier<sub>opt</sub> ( FormalParameterList ) { FunctionBody }

1. Return false.

**NOTE** Static semantic rules that depend upon substructure generally do not look into function definitions.

**Static Semantics: ExpectedArgumentCount**

FormalParameterList :  
 [empty]  
FunctionRestParameter

1. Return 0.

FormalParameterList :  
FormalsList  
FormalsList , FunctionRestParameter

1. Return the ExpectedArgumentCount of *FormalsList*.

**NOTE** The ExpectedArgumentCount of a *FormalParameterList* is the number of *FormalParameters* to the left of either the rest parameter or the first *FormalParameter* with an Initialiser.

FormalsList : FormalParameter

1. If HasInitialiser of *FormalParameter* is false return 0  
 2. Return 1.

**Deleted:** String Value

**Formatted:** Note

**Comment [AW67]:** Note that formal parameters with an initialiser are allowed after the first parameter with a initializer but such parameters are considered to be optional with undefined as their default value.

*FormalsList* : *FormalsList* , *FormalParameter*

1. Let *count* be the *ExpectedArgumentCount* of *FormalsList*.
2. If *HasInitialiser* of *FormalsList* is **true** or *HasInitialiser* of *FormalParameter* is **true**, then return *count*.
3. Return *count*+1.

#### **Static Semantics: HasInitialiser**

*FormalsList* : *FormalParameter*

1. Return *HasInitialiser* of *FormalParameter*.

*FormalsList* : *FormalsList* , *FormalParameter*

1. If *HasInitialiser* of *FormalsList* is **true**, then return **true**.
2. Return *HasInitialiser* of *FormalParameter*.

*FormalParameter* : *BindingElement*

1. Return *HasInitialiser* of *BindingElement*.

#### **Static Semantics: IsConstantDeclaration**

*FunctionDeclaration* : **function** *BindingIdentifier* ( *FormalParameterList* ) { *FunctionBody* }

1. Return **false**.

#### **Static Semantics: LexicallyDeclaredNames**

*FunctionBody* : [empty]

1. Return an empty List.

*FunctionBody* : *StatementList*

1. Return the *LexicallyDeclaredNames* of *StatementList*.

#### **Static Semantics: VarDeclaredNames**

*FunctionDeclaration* : **function** *BindingIdentifier* ( *FormalParameterList* ) { *FunctionBody* }

1. Return an empty List.

*FunctionBody* : [empty]

1. Return an empty List.

*FunctionBody* : *StatementList*

1. Return the *VarDeclaredNames* of *StatementList*.

#### **Runtime Semantics**

##### **Runtime Semantics: BindingInitialisation**

With parameters *value* and *environment* and optional parameter *index*.

**Deleted:** Initialization

**Deleted:** environment

NOTE When undefined is passed for environment it indicates that a PutValue operation should be used to assign the initialisation value. This is the case for formal parameter lists of non-strict functions. In that case the formal parameter bindings are preinitialized in order to deal with the possibility of multiple parameters with the same name.

FormalParameterList : [empty]

1. Return NormalCompletion(empty).

FormalParameterList : FunctionRestParameter

1. Return the result of performing Indexed Binding Initialisation for FunctionRestParameter using value, lenValue, 0, and environment as the arguments.

FormalParameterList : FormalsList

1. Return the result of performing Indexed Binding Initialisation for FormalsList using value, lenValue, 0, and environment as the arguments.

FormalParameterList : FormalsList , FunctionRestParameter

1. Let restIndex be the result of performing Indexed Binding Initialisation for FormalsList using value, 0, and environment as the arguments.

2. ReturnIfAbrupt(restIndex).

3. Return the result of performing Indexed Binding Initialisation for FunctionRestParameter using value, lenValue, restIndex, and environment as the arguments.

### Runtime Semantics: Indexed Binding Initialisation

With parameters array, nextIndex, and environment.

FormalsList : FormalParameter

1. Let status be the result of performing Indexed Binding Initialisation for FormalParameter using array, env, and nextIndex as the arguments.

2. ReturnIfAbrupt(status).

3. Return nextIndex + 1.

FormalsList : FormalsList , FormalParameter

1. Let lastIndex be the result of performing Indexed Binding Initialisation for FormalsList using array, nextIndex, and environment as the arguments.

2. ReturnIfAbrupt(lastIndex).

3. Let status be the result of performing Indexed Binding Initialisation for FormalParameter using array, lastIndex, and environment as the arguments.

4. ReturnIfAbrupt(status).

5. Return lastIndex + 1.

FunctionRestParameter : ... BindingIdentifier

1. Asset: array is a well formed arguments object and hence it has a valid integer valued "length" property.

2. Let argumentsLength be the result of calling the [[Get]] internal method of value with argument "length".

3. Let argumentsLength be argumentsLength.[[value]].

4. Let A be the result of the abstraction operation ArrayCreate (15.4) with argument 0.

5. Let n=0.

6. Repeat, while nextIndex < argumentsLength,

a. Let P be ToString(nextIndex).

b. Assert: array is a well formed arguments object, hence it must have a property P.

c. Let v be the result of calling the [[Get]] internal method of array passing P as the argument.

**Deleted:** initialization

**Deleted:** the

**Deleted:** Perform

**Deleted:** Initialization

**Comment [AWB768]:** ?????

**Deleted:** ¶

Return

**Comment [AWB769]:** ?????

**Deleted:** <#>Asset: value is a new created arguments object and hence it has a valid integer valued "length" property  
<#>Let lenValue be the result of calling the [[Get]] internal method of value with argument "length".¶

**Deleted:** Perform

**Deleted:** Initialization

**Deleted:** <#>Return.¶

**Deleted:** Initialization

**Deleted:** lenValue,

**Deleted:** Perform

**Deleted:** Initialization

**Comment [AWB770]:** ?????

**Deleted:** .[[value]]

**Deleted:** <#>Return.¶

**Deleted:** Initialization

**Deleted:** arrayLength,

**Deleted:** Perform

**Deleted:** Initialization

**Deleted:** value

**Deleted:** i

**Deleted:** i

**Deleted:** Initialization

**Deleted:** value

**Deleted:** lenValue,

**Deleted:** Perform

**Deleted:** Initialization

**Deleted:** value

**Deleted:** lenValue,

**Deleted:** Initialization

**Deleted:** arrayLength

- d. Call the `[[DefineOwnProperty]]` internal method of *A* with arguments `ToString(n)`, `[[value]]`, `Property Descriptor` `{[[Value]]: v, [[value]], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, and `false`.
- e. Let *n* = *n*+1.
- f. Let *nextIndex* = *nextIndex* +1.
- 7. Return the result of performing `BindingInitialisation` for `BindingIdentifier` using *A* and *environment* as arguments.

**Deleted:** Perform  
**Deleted:** Initialization

*FormalParameter* : *BindingElement*

1. Return the result of performing `IndexedBindingInitialisation` for `BindingElement` using *array*, *nextIndex*, and *environment* as the arguments.

**Deleted:** Perform  
**Deleted:** Initialization  
**Deleted:** *arrayLength*,

### Runtime Semantics: Instantiate Function Declaration

*FunctionDeclaration* : `function` *BindingIdentifier* `(` *FormalParameterList* `) {` *FunctionBody* `}`

1. If the *FunctionDeclaration* is contained in strict code or if its *FunctionBody* is strict code, then let *strict* be `true`. Otherwise let *strict* be `false`.
2. Let *scope* be the `LexicalEnvironment` of the current execution context.
3. Let *F* be the result of performing the `FunctionCreate` abstract operation with arguments `Normal`, *FormalParameterList*, *FunctionBody*, *scope*, and *strict*.
4. Performing the abstract operation `MakeConstructor` with argument *F*.
5. Return *F*.

**Deleted:** The production

**Deleted:** opt

**Deleted:** is instantiated as follows during Declaration Binding instantiation (10.5):

**Deleted:** Return the result of creating a new `Function` object as specified in 13.2 with parameters specified by *FormalParameterList*<sub>opt</sub>, and body specified by *FunctionBody*. Pass in the `VariableEnvironment` of the running execution context as the `Scope`. Pass in `true` as the `Strict` flag if the *FunctionDeclaration* is contained in strict code or if its *FunctionBody* is strict code.

**Deleted:** The production

**Deleted:** opt

**Deleted:** is evaluated as follows:

**Formatted:** Outline numbered + Level: 1 + Numbering Style: 1, 2, 3, ... + Start at: 1 + Alignment: Left + Aligned at: 0" + Tab after 0.25" + Indent at: 0.25"

**Deleted:** the result of creating a new `Function` object as specified in 13.2 with parameters specified by *FormalParameterList*<sub>opt</sub> and body specified by *FunctionBody*. Pass in the `LexicalEnvironment` of the running execution context as the `Scope`. Pass in `true` as the `Strict` flag if the *FunctionExpression* is contained in strict code or if its *FunctionBody* is strict code.

**Deleted:** opt

**Deleted:** running

**Deleted:** <#>Let *closure* be the result of creating a new `Function` object as specified in 13.2 with parameters specified by *FormalParameterList*<sub>opt</sub> and body specified by *FunctionBody*. Pass in *funcEnv* as the `Scope`. Pass in `true` as the `Strict` flag if the *FunctionExpression* is contained in strict code or if its *FunctionBody* is strict code.¶

**Deleted:** immutable

### Runtime Semantics: Evaluation

*FunctionDeclaration* : `function` *BindingIdentifier* `(` *FormalParameterList* `) {` *FunctionBody* `}`

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

*FunctionExpression* : `function` `(` *FormalParameterList* `) {` *FunctionBody* `}`

1. If the *FunctionExpression* is contained in strict code or if its *FunctionBody* is strict code, then let *strict* be `true`. Otherwise let *strict* be `false`.
2. Let *scope* be the `LexicalEnvironment` of the current execution context.
3. Let *closure* be the result of performing the `FunctionCreate` abstract operation with arguments `Normal`, *FormalParameterList*, *FunctionBody*, *scope*, and *strict*.
4. Performing the abstract operation `MakeConstructor` with argument *closure*.
5. Return *closure*.

*FunctionExpression* : `function` *BindingIdentifier* `(` *FormalParameterList* `) {` *FunctionBody* `}`

1. If the *FunctionExpression* is contained in strict code or if its *FunctionBody* is strict code, then let *strict* be `true`. Otherwise let *strict* be `false`.
2. Let *funcEnv* be the result of calling `NewDeclarativeEnvironment` passing the *current* execution context's `Lexical Environment` as the argument.
3. Let *envRec* be *funcEnv*'s environment record.
4. Call the `CreateImmutableBinding` concrete method of *envRec* passing the String value of *BindingIdentifier* as the argument.
5. Let *closure* be the result of performing the `FunctionCreate` abstract operation with arguments `Normal`, *FormalParameterList*, *FunctionBody*, *funcEnv*, and *strict*.
6. Performing the abstract operation `MakeConstructor` with argument *closure*.
7. Call the `InitializeBinding` concrete method of *envRec* passing the String value of *Identifier* and *closure* as the arguments.
8. Return *closure*.

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

**NOTE** A *prototype* property is automatically created for every function, to allow for the possibility that the function will be used as a constructor.

*FunctionBody* : *StatementList*<sub>opt</sub>

1. The code of this *FunctionBody* is strict mode code if it is contained in strict mode code or if the Directive Prologue (14.1) of its *StatementList* contains a Use Strict Directive or if any of the conditions in 10.1.1 apply. If the code of this *FunctionBody* is strict mode code, *StatementList* is evaluated in the following steps as strict mode code. Otherwise, *StatementList* is evaluated in the following steps as non-strict mode code.
2. If *StatementList* is present return the result of evaluating *StatementList*.
3. Else return *NormalCompletion*(*undefined*).

## 13.2 Arrow Function Definitions

### Syntax

*ArrowFunction* :  
*ArrowParameters* => *ConciseBody*

*ArrowParameters* :  
*BindingIdentifier*  
(*ArrowFormalParameterList*)

*ArrowFormalParameterList* :  
[empty]  
*FunctionRestParameter*  
*CoverFormalsList*  
*CoverFormalsList* , *FunctionRestParameter*

*ConciseBody* :  
!lookahead ∈ { { } } *AssignmentExpression*  
{ *FunctionBody* }

*CoverFormalsList* :  
*Expression*

### Supplemental Syntax

When processing the production *CoverFormalsList* : *Expression* the *FormalsList* production is used to further restrict the source code that matches *Expression*.

*ArrowFormalParameterList* :  
*FormalParameterList*

### Static Semantics

#### Static Semantics: Early Errors

*ArrowFunction* : *ArrowParameters* => *ConciseBody*

- It is a Syntax Error if any element of the *LexicallyDeclaredNames* of *ConciseBody* also occurs in the *VarDeclaredNames* of *ConciseBody*.
- It is a Syntax Error if any element of the *BoundNames* of *ArrowParameters* also occurs in the *LexicallyDeclaredNames* of *ConciseBody*.
- It is a Syntax Error if *ConciseBody* Contains *YieldExpression*.

<b>Deleted:</b> The production
<b>Deleted:</b> <i>Source</i>
<b>Deleted:</b> <i>Elements</i>
<b>Deleted:</b> is evaluated as follows:
<b>Deleted:</b> part of a <i>FunctionDeclaration</i> or <i>FunctionExpression</i> that is
<b>Deleted:</b> <i>SourceElements</i>
<b>Deleted:</b> <i>SourceElements</i>
<b>Deleted:</b> <i>SourceElements</i>
<b>Deleted:</b> <i>Source</i>
<b>Deleted:</b> <i>Elements</i>
<b>Deleted:</b> <i>SourceElements</i>
<b>Deleted:</b> normal,
<b>Deleted:</b> , empty

ArrowParameters : BindingIdentifier

- It is a Syntax Error if the String Value of the sole element of the BoundNames of *BindingIdentifier* is **eval** or **arguments**.

ArrowFormalParameterList : CoverFormalsList , FunctionRestParameter

- It is a Syntax Error if *CoverFormalsList* contains any element which is also contained in *BoundNames* of *FunctionRestParameter*.

CoverFormalsList : Expression

- It is a Syntax Error if the source code corresponding to *Expression* cannot be parsed using *FormalsList* as the goal symbol.
- All Early Errors rules for *FormalsList* also apply to the Covered *FormalsList* of *Expression*.

Static Semantics: BoundNamesArrowParameters : BindingIdentifier

1. Return the *BoundNames* of *BindingIdentifier*.

ArrowParameters : ( ArrowFormalParameterList )

1. Return the *BoundNames* of *ArrowFormalParameterList*.

ArrowFormalParameterList : [empty]

1. Return an empty List.

ArrowFormalParameterList : FunctionRestParameter

1. Return the *BoundNames* of *FunctionRestParameter*.

ArrowFormalParameterList : CoverFormalsList

1. Return the *BoundNames* of *CoverFormalsList*.

ArrowFormalParameterList : CoverFormalsList , FunctionRestParameter

1. Let *names* be *BoundNames* of *CoverFormalsList*.
2. Append to *names* the *BoundNames* of *FunctionRestParameter*.
3. Return *names*.

CoverFormalsList : Expression

1. Return the *BoundNames* of Covered *FormalsList* of *Expression*.

Static Semantics: Contains

With parameter *symbol*.

ArrowFunction : ArrowParameters => ConciseBody

1. Return false.

NOTE Static semantic rules that depend upon substructure generally do not look into function definitions.

### Static Semantics: CoveredFormalsList

CoverFormalsList : Expression

1. Return the result of parsing the source code corresponding to *Expression* using *FormalsList* as the goal symbol.

### Static Semantics: ExpectedArgumentCount

ArrowParameters : BindingIdentifier

1. Return 1.

ArrowParameters : ( ArrowFormalParameterList )

1. Return the ExpectedArgumentCount of *ArrowFormalParameterList*.

ArrowFormalParameterList :  
  [empty]  
  FunctionRestParameter

1. Return 0.

ArrowFormalParameterList :  
  CoverFormalsList  
  CoverFormalsList , FunctionRestParameter

1. Return the ExpectedArgumentCount of CoveredFormalsList of *CoverFormalsList*.

### Static Semantics: LexicallyDeclaredNames

ConciseBody : [lookahead ∈ { { } } AssignmentExpression

1. Return an empty List.

ConciseBody : { FunctionBody }

1. Return the LexicallyDeclaredNames of *FunctionBody*.

### Static Semantics: VarDeclaredNames

ConciseBody : [lookahead ∈ { { } } AssignmentExpression

1. Return the VarDeclaredNames of *AssignmentExpression*.

ConciseBody : { FunctionBody }

1. Return the VarDeclaredNames of *FunctionBody*.

### Runtime Semantics

#### Runtime Semantics: Binding Initialisation

With parameters *value* and *environment* and optional parameter *index*.

**NOTE** When **undefined** is passed for *environment* it indicates that a PutValue operation should be used to assign the initialisation value. This is the case for formal parameter lists of non-strict functions. In that case the formal parameter bindings are preinitialized in order to deal with the possibility of multiple parameters with the same name.

ArrowParameters : BindingIdentifier

1. Return the result of performing Indexed Binding Initialisation for *BindingIdentifier* using *value*, *lenValue*, 0, and *environment* as the arguments.

ArrowParameters : ( ArrowFormalParameterList )

1. Return the result of performing Binding Initialisation for *ArrowFormalParameterList* using *value*, *environment* as the arguments. Also pass the optional argument *index* if it is present.

ArrowFormalParameterList : [empty]

1. Return NormalCompletion(empty).

ArrowFormalParameterList : FunctionRestParameter

1. Return the result of performing Indexed Binding Initialisation for *FunctionRestParameter* using *value*, *lenValue*, 0, and *environment* as the arguments.

Comment [AWB771]: ?????

ArrowFormalParameterList : CoverFormalsList

1. Return the result of performing Indexed Binding Initialisation for *CoveredCormalsList* of *CoverFormalsList* using *value*, *lenValue*, 0, and *environment* as the arguments.

Comment [AWB772]: ?????

ArrowFormalParameterList : CoverFormalsList , FunctionRestParameter

1. Let *restIndex* be the result of performing Indexed Binding Initialisation for *CoveredCormalsList* of *CoverFormalsList* using *value*, 0, and *environment* as the arguments.
2. ReturnIfAbrupt(*restIndex*).
3. Return the result of performing Indexed Binding Initialisation for *FunctionRestParameter* using *value*, *lenValue*, *restIndex*, and *environment* as the arguments.

Runtime Semantics: Evaluation

ArrowFunction : ArrowParameters => ConciseBody

1. Let *strict* be *true*.
2. Let *scope* be the LexicalEnvironment of the current execution context.
3. Let *closure* be the result of performing the *FunctionCreate* abstract operation with arguments *Arrow*, *ArrowParameterList*, *ConciseBody*, *scope*, and *strict*.
4. Return *closure*.

Comment [AWB773]: Confirm: concise methods are always strict code.

ConciseBody : !lookahead ∈ { { } } AssignmentExpression

1. The code of this *ArrowBody* is strict mode code if it is contained in strict mode code or if any of the conditions in 10.1.1 apply. If the code of this *ArrowBody* is strict mode code, *AssignmentExpression* is evaluated in the following steps as strict mode code. Otherwise, *AssignmentExpression* is evaluated in the following steps as non-strict mode code.
  2. Let *exprRef* be the result of evaluating *AssignmentExpression*.
  3. Let *exprValue* be *GetValue(exprRef)*.
  4. If *exprValue* is an abrupt completion, return *exprValue*.
  5. Return Completion {[ [type] ]: return, {[ [value] ]: *exprValue*, {[ [target] ]: empty}}.

Comment [AWB774]: TODO, need to resc whether or not ArrowBodies are always strict

ConciseBody : { FunctionBody }

1. Return the result of evaluating *FunctionBody*.

### 13.3 Method Definitions

#### Syntax

MethodDefinition :

```
PropertyName ( FormalParameterList ) ConciseBody
* PropertyName ( FormalParameterList ) ConciseBody
get PropertyName ( ) ConciseBody
set PropertyName ( PropertySetParameterList ) ConciseBody
```

PropertySetParameterList :

```
BindingIdentifier
BindingPattern
```

**Deleted:** {  
**Deleted:** FunctionBody }  
**Deleted:** {  
**Deleted:** FunctionBody }

#### Static Semantics

##### Early Errors

MethodDefinition : PropertyName ( FormalParameterList ) ConciseBody

and

MethodDefinition : \* PropertyName ( FormalParameterList ) ConciseBody

- It is a Syntax Error if any element of the LexicallyDeclaredNames of FormalParameterList also occurs in the VarDeclaredNames of ConciseBody.
- It is a Syntax Error if any element of the BoundNames of FormalParameterList also occurs in the LexicallyDeclaredNames of ConciseBody.

MethodDefinition : PropertyName ( FormalParameterList ) ConciseBody

- It is a Syntax Error if ConciseBody Contains YieldExpression.

MethodDefinition : get PropertyName ( ) ConciseBody

- It is a Syntax Error if ConciseBody Contains YieldExpression.

MethodDefinition : set PropertyName ( PropertySetParameterList ) ConciseBody

- It is a Syntax Error if any element of the BoundNames of PropertySetParameterList also occurs in the LexicallyDeclaredNames of ConciseBody.
- It is a Syntax Error if ConciseBody Contains YieldExpression.

**Formatted:** Don't add space between paragraphs of the same style  
**Formatted:** Add space between paragraphs of the same style

PropertySetParameterList : BindingPattern

- It is a Syntax Error if BoundNames of BindingPattern contains any duplicate elements.

#### Static Semantics: BoundNames

PropertySetParameterList : BindingIdentifier

1. Return BoundNames of BindingIdentifier.

PropertySetParameterList: BindingPattern

1. Return BoundNames of BindingPattern.

#### Static Semantics: ExpectedArgumentCount

PropertySetParameterList : BindingIdentifier

1. Return 1.

#### PropertySetParameterList : BindingPattern

1. Return 1.

#### Static Semantics: PropName

##### MethodDefinition : PropertyName ( FormalParameterList ) ConciseBody

1. Return PropName of *PropertyName*.

##### MethodDefinition : \* PropertyName ( FormalParameterList ) ConciseBody

1. Return PropName of *PropertyName*.

##### MethodDefinition : get PropertyName ( ) ConciseBody

1. Return PropName of *PropertyName*.

##### MethodDefinition : set PropertyName ( PropertySetParameterList ) ConciseBody

1. Return PropName of *PropertyName*.

#### Runtime Semantics

##### Runtime Semantics: Property Definition Evaluation

With parameter *object*.

##### MethodDefinition : PropertyName ( FormalParameterList ) ConciseBody

1. Let *propName* be PropName of *PropertyName*.
2. Let *strict* be **true**.
3. Let *scope* be the current execution context's LexicalEnvironment.
4. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments *Method*, *FormalParameterList*, *ConciseBody*, *scope*, and *strict* and with *object* as the *homeObject* optional argument and *propName* as the *methodName* optional argument.
5. Let *desc* be the Property Descriptor{[[Value]]: *closure*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true**}.
6. Return the result of calling the [[DefineOwnProperty]] internal method of *object* with arguments *propName*, *desc*, and **false**.

**Comment [AWB775]:** Confirm: concise methods are always strict code.

##### MethodDefinition : \* PropertyName ( FormalParameterList ) ConciseBody

1. Let *propName* be PropName of *PropertyName*.
2. Let *strict* be **true**.
3. Let *scope* be the current execution context's LexicalEnvironment.
4. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments *Method*, *FormalParameterList*, *ConciseBody*, *scope*, and *strict* and with *object* as the *homeObject* optional argument and *propName* as the *methodName* optional argument.
5. Let *desc* be the Property Descriptor{[[Value]]: *closure*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true**}.
6. Return the result of calling the [[DefineOwnProperty]] internal method of *object* with arguments *propName*, *desc*, and **false**.

**Comment [AWB777]:** Confirm: concise methods are always strict code.

##### MethodDefinition : get PropertyName ( ) ConciseBody

1. Let *propName* be PropName of *PropertyName*.

**Comment [AWB778]:** TODO: need to replace with creation of a generator object

**Comment [AWB779]:** Confirm: methods are non-writable, but configurable.

**Deleted:** { *FunctionBody* }

2. If this *MethodDefinition* is contained in strict code or if *ConciseBody* is strict code, then let *strict* be **true**. Otherwise, let *strict* be **false**.
3. Let *scope* be the current execution context's *LexicalEnvironment*.
4. Let *formalParameterList* be the production *FormalParameterList*: *empty*
5. Let *closure* be the result of performing the *FunctionCreate* abstract operation with arguments *Method*, *formalParameterList*, *ConciseBody*, *scope*, and *strict* and with *object* as the *homeObject* optional argument and *propName* as the *methodName* optional argument.
6. Let *desc* be the Property Descriptor{[[Get]]: *closure*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}
7. Return the result of calling the [[DefineOwnProperty]] internal method of *object* with arguments *propName*, *desc*, and **false**.

*MethodDefinition* : **set** *PropertyName* ( *PropertySetParameterList* ) *ConciseBody*

**Deleted:** { *FunctionBody* }

1. Let *propName* be *PropName* of *PropertyName*.
2. If this *MethodDefinition* is contained in strict code or if *ConciseBody* is strict code, then let *strict* be **true**. Otherwise, let *strict* be **false**.
3. Let *scope* be the current execution context's *LexicalEnvironment*.
4. Let *closure* be the result of performing the *FunctionCreate* abstract operation with arguments *Method*, *PropoertySetParameterList*, *ConciseBody*, *scope*, and *strict* and with *object* as the *homeObject* optional argument and *propName* as the *methodName* optional argument.
5. Let *desc* be the Property Descriptor{[[Set]]: *closure*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}
6. Return the result of calling the [[DefineOwnProperty]] internal method of *object* with arguments *propName*, *desc*, and **false**.

**NOTE** The strictness and property attributes for **set** and **get** *MethodDefinition* productions differ from the other *MethodDefinition* productions in order to maintain compatibility with previous editions of this specification.

## 13.4 Generator Definitions

### Syntax

*GeneratorDeclaration* :  
**function** \* *BindingIdentifier* ( *FormalParameterList* ) { *FunctionBody* }

*GeneratorExpression* :  
**function** \* *BindingIdentifier<sub>opt</sub>* ( *FormalParameterList* ) { *FunctionBody* }

### Static Semantics

#### Static Semantics: Early Errors

*FunctionDeclaration* : **function** \* *BindingIdentifier* ( *FormalParameterList* ) { *FunctionBody* }  
 and

*FunctionExpression* : **function** \* *BindingIdentifier<sub>opt</sub>* ( *FormalParameterList* ) { *FunctionBody* }

**Formatted:** Font: Not Italic

**Formatted:** Font: Not Italic

- It is a Syntax Error if any element of the *BoundNames* of *FormalParameterList* also occurs in the *LexicallyDeclaredNames* of *FunctionBody*.

**Formatted:** Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

**Comment [AWB680]:** There is an issue that function declarations have "lexically declared names" and in ES<=5 nested functions are allowed to have the same name as vars or formal parameters. We either need to take this a breaking change or adjust this rule to only apply to function declarations nested within blocks.

#### Static Semantics: Contains

With parameter *symbol*.

*FunctionDeclaration* : **function** \* *BindingIdentifier* ( *FormalParameterList* ) { *FunctionBody* }

1. Return **false**.

*FunctionExpression* : **function** \* *BindingIdentifier<sub>opt</sub>* ( *FormalParameterList* ) { *FunctionBody* }

**Formatted:** Outline numbered + Level: 1 + Numbering Style: 1, 2, 3, ... + Start at: 1 + Alignment: Left + Aligned at: 0" + Tab after: 0.25" + Indent at: 0.25"

1. Return false.

**NOTE** Static semantic rules that depend upon substructure generally do not look into function definitions.

### **13.5 Creating Function Objects and Constructors**

The abstract operation `FunctionCreate` requires the arguments: `kind` which is one of (Normal, Method, Arrow), an parameter list specified by `FormalParameterList`, a body specified by `FunctionBody`, a Lexical Environment specified by `Scope`, a Boolean flag `Strict`, and optionally, an object `functionPrototype`, an object `homeObject` and a string `methodName`. `FunctionCreate` performs the following steps:

1. Create a new native ECMAScript object and let `F` be that object.
2. Set all the internal methods, except for `[[GetProperty]]`, of `F` as described in 8.12.
3. Add the `[[NativeBrand]]` internal property with value `NativeFunction` to `F`.
4. If the `functionPrototype` argument was not provided, then
  - a. Let `functionPrototype` be the intrinsic object `%FunctionPrototype%`.
5. Set the `[[Prototype]]` internal property of `F` to `functionPrototype`.
6. Set the `[[GetProperty]]` internal property of `F` as described in 15.3.5.4.
7. Set the `[[Call]]` internal property of `F` as described in 13.5.1.
8. Set the `[[Scope]]` internal property of `F` to the value of `Scope`.
9. Set the `[[FormalParameters]]` internal property of `F` to `FormalParameterList`.
10. Set the `[[Code]]` internal property of `F` to `FunctionBody`.
11. Set the `[[Extensible]]` internal property of `F` to `true`.
12. Set the `[[Realm]]` internal property of `F` to the current execution context's `Realm`.
13. If the `homeObject` argument was provided, set the `[[Home]]` internal property of `F` to `homeObject`.
14. If the `methodName` argument was provided, set the `[[MethodName]]` internal property of `F` to `methodName`.
15. Let `len` be the `ExpectedArgumentCount` of `FormalParameterList`.
16. Call the `[[DefineOwnProperty]]` internal method of `F` with arguments `"length"`, Property Descriptor `{[[Value]]: len, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}`, and `false`.
17. If `kind` is Normal and `Strict` is true, then
  - a. Let `thrower` be the `[[ThrowTypeError]]` function Object (13.5.3).
  - b. Call the `[[DefineOwnProperty]]` internal method of `F` with arguments `"caller"`, PropertyDescriptor `{[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: false}`, and `false`.
  - c. Call the `[[DefineOwnProperty]]` internal method of `F` with arguments `"arguments"`, PropertyDescriptor `{[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: false}`, and `false`.
18. Set the `[[Strict]]` internal property of `F` to `Strict`.
19. Return `F`.

The abstract operation `MakeConstructor` requires a `Function` argument `F` and optionally, a Boolean `writablePrototype` and an object `prototype`. It converts `F` into a constructor by performs the following steps:

1. If the `prototype` argument was not provided, then
  - a. Let `prototype` be the result of the abstraction operation `ObjectCreate` (15.2).
2. If the `writablePrototype` argument was not provided, then
  - a. Let `writablePrototype` be `true`.
3. Set the `[[Construct]]` internal property of `F` as described in 13.5.2.
4. Set the `[[HasInstance]]` internal property of `F` as described in 15.3.5.3.
5. Set the `[[Home]]` internal property of `F` to `prototype`.
6. Set the `[[MethodName]]` internal property of `F` to `"constructor"`.
7. Call the `[[DefineOwnProperty]]` internal method of `prototype` with arguments `"constructor"`, Property Descriptor `{[[Value]]: F, [[Writable]]: writablePrototype, [[Enumerable]]: false, [[Configurable]]: writablePrototype}`, and `false`.
8. Call the `[[DefineOwnProperty]]` internal method of `F` with arguments `"prototype"`, Property Descriptor `{[[Value]]: prototype, [[Writable]]: writablePrototype, [[Enumerable]]: false, [[Configurable]]: false}`, and `false`.
9. Return.

**Deleted:** 13.1 Strict Mode Restrictions  
 It is a `SyntaxError` if any `Identifier` value occurs more than once within a `FormalParameterList` of a strict mode `FunctionDeclaration` or `FunctionExpression`.  
 It is a `SyntaxError` if the `Identifier "eval"` or the `Identifier "arguments"` occurs within a `FormalParameterList` of a strict mode `FunctionDeclaration` or `FunctionExpression`.  
 It is a `SyntaxError` if the `Identifier "eval"` or the `Identifier "arguments"` occurs as the `Identifier` of a strict mode `FunctionDeclaration` or `FunctionExpression`.  
**Deleted:** 2

**Deleted:** 1.1

**Deleted:** A Function object is constructed as follows Given given

**Deleted:** optional

**Deleted:** and

**Deleted:** a Function object is constructed as follows

**Deleted:** Set

**Deleted:** Class

**Deleted:** IsFunction

**Deleted:** of

**Deleted:** to "Function"

**Deleted:** the

**Deleted:** standard built-in Function prototype object as specified in 15.3.3.1

**Deleted:** 2

**Deleted:** <#> Set the `[[Construct]]` internal property of `F` as described in 13.2.2.  
 <#> Set the `[[HasInstance]]` internal property of `F` as described in 15.3.5.3.

**Deleted:** <#> Let `names` be a List containing, in right textual order, the Strings corresponding to the identifiers of `FormalParameterList`. If no parameters are specified, let `names` be the empty list.

**Deleted:** `names`

**Formatted:** Font: Italic

**Deleted:** number of formal parameters specified

**Deleted:** .

**Deleted:** If no parameters are specified, let `len` 0

**Deleted:** 2

**Deleted:** <#> Let `proto` be the result of creating

**Deleted:** {

**Deleted:** true

**Deleted:** true

**Deleted:** proto

**Deleted:** {

**Deleted:** true

**Deleted:** <#> If `Strict` is true, then

**Deleted:** `F`

### 13.5.1 [[Call]]

When the [[Call]] internal method for a Function object  $F$  is called with a  $thisArgument$  and a list of arguments, the following steps are taken:

1. Let  $callerContext$  be the current execution context.
2. Suspend  $callerContext$ .
3. Let  $calleeContext$  be a new execution context.
4. Set  $calleeContext$ 's PreviousContext to  $callerContext$ .
5. Set  $calleeContext$ 's Realm to the value of  $F$ 's [[Realm]] internal property.
6. Let  $thisMode$  be the value of  $F$ 's [[ThisMode]] internal property.
7. If  $thisMode$  is lexical, then
  - a. Let  $localEnv$  be the result of calling NewDeclarativeEnvironment passing the value of the [[Scope]] internal property of  $F$  as the argument.
8. Else,
  - a. If  $thisMode$  is strict, set  $thisValue$  to  $thisArgument$ .
  - b. Else if  $thisArgument$  is null or undefined, set  $thisValue$  to %GlobalObject%.
  - c. Else if Type( $thisArg$ ) is not Object, set the  $thisValue$  to ToObject( $thisArgument$ ).
  - d. Else set the  $thisValue$  to  $thisArgument$ .
  - e. Let  $localEnv$  be the result of calling NewMethodEnvironment passing  $F$  and  $thisValue$  as the arguments.
9. Set the LexicalEnvironment of  $calleeContext$  to  $localEnv$ .
10. Set the VariableEnvironment of  $calleeContext$  to  $localEnv$ .
11. Make  $calleeContext$  the current execution context.
12. Let  $status$  be the result of performing Function Declaration Binding Instantiation using the function  $F$ ,  $argumentsList$ , and  $localEnv$  as described in 10.5.3.
13. If  $status$  is an abrupt completion, then
  - a. Restore  $callerContext$  as the current execution context.
  - b. Return  $status$ .
14. Let  $result$  be the result of evaluating the  $FunctionBody$  that is the value of  $F$ 's [[Code]] internal property.
15. Restore  $callerContext$  as the current execution context.
16. If  $result.type$  is return then return  $\text{NormalCompletion}(result.value)$ .
17. Return  $result$ .

### 13.5.2 [[Construct]]

When the [[Construct]] internal method for a Function object  $F$  is called with a possibly empty list of arguments, the following steps are taken:

1. Let  $proto$  be the value of calling the [[Get]] internal property of  $F$  with argument "prototype".
2. ReturnIfAbrupt( $proto$ ).
3. If Type( $proto$ ) is Object, let  $obj$  be the result of the abstraction operation ObjectCreate with argument  $proto$ .
4. If Type( $proto$ ) is not Object, let  $obj$  be the result of the abstraction operation ObjectCreate.
5. Let  $result$  be the result of calling the [[Call]] internal property of  $F$ , providing  $obj$  as the  $this$  value and providing the argument list passed into [[Construct]] as  $args$ .
6. ReturnIfAbrupt( $result$ ).
7. If Type( $result$ ) is Object then return  $result$ .
8. Return  $\text{NormalCompletion}(obj)$ .

### 13.5.3 The [[ThrowTypeError]] Function Object

The [[ThrowTypeError]] object is a unique function object that is defined once as follows:

1. Create a new native ECMAScript object and let  $F$  be that object.
2. Set all the internal methods of  $F$  as described in 8.12.
3. Add the [[NativeBrand]] internal property with value NativeFunction to  $F$ .
4. Set the [[Prototype]] internal property of  $F$  to the standard built-in Function prototype object as specified in 15.3.3.1.
5. Set the [[Call]] internal property of  $F$  as described in 13.2.1.
6. Set the [[Scope]] internal property of  $F$  to the Global Environment.
7. Set the [[FormalParameters]] internal property of  $F$  to the  $\text{FormalParameterList : } \{\text{empty}\}$  production.

**Deleted:** NOTE A `prototype` property is automatically created for every function, to allow for the possibility that the function will be used as a constructor. ¶

**Deleted:** 2

**Deleted:** 1

**Deleted:** .1

**Deleted:** `value`

**Deleted:** Let  $funcCtx$  status be the result of establishing a new execution context for function code using the value of  $F$ 's [[FormalParameters]] internal property, the passed argument List  $args$ , and the `this` value as described in 10.4.3.

**Comment [AWB 381]:** May need to update section number

**Comment [AWB82]:** Exceptions could occur during evaluation of parameter default value expressions.

**Deleted:** If an exception was thrown in the previous step, return ( $\text{throw}$ ,  $V$ , empty) where  $V$  is the exception. (Execution now proceeds as if no exception were thrown.)

**Deleted:** <#> If  $F$  does not have a [[Code]] internal property or if its value is an empty `FunctionBody`, then ¶ <#> Let  $result$  be ( $\text{normal}$ ,  $\text{NormalCompletion}(\text{undefined})$ ). ¶ <#> Else, ¶

**Deleted:** Exit the execution context  $funcCtx$ , restoring the previous execution context.

**Deleted:** <#> If  $result.type$  is throw then throw  $result.value$ .

**Deleted:** Otherwise  $result.type$  must be normal. Return `undefined`

**Deleted:** 2

**Deleted:** 1

**Deleted:** .21

**Deleted:** <#> Let  $obj$  be a newly created native ECMAScript object. ¶ <#> Set all the internal methods of  $obj$  as specified in 8.12. ¶ <#> Set the [[Class]] internal property of  $obj$  to "Object".

**Deleted:** `set`

**Deleted:** the [[Prototype]] internal property of

**Deleted:** to `proto`

**Deleted:** `set`

**Deleted:** the [[Prototype]] internal property of

**Deleted:** to the standard built-in Object prototype object as

**Deleted:** `proto`

**Deleted:** `[[value]]`

**Deleted:** 2

**Deleted:** 1

**Deleted:** .1

**Deleted:** `Set`

**Deleted:** `Class`

**Deleted:** `IsFunction`

**Deleted:** `of`

**Deleted:** to "Function"

**Deleted:** an empty List

8. Set the `[[Code]]` internal property of  $F$  to be a `FunctionBody` that unconditionally throws a `TypeError` exception and performs no other action.
9. Call the `[[DefineOwnProperty]]` internal method of  $F$  with arguments `"length"`, Property Descriptor `{[[Value]]: 0, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}`, and `false`.
10. Set the `[[Extensible]]` internal property of  $F$  to `false`.
11. Let `[[ThrowTypeError]]` be  $F$ .

## 14 Program

### Syntax

*Program :*

*ProgramBody<sub>opt</sub>*

*ProgramBody :*

*StatementList*

### Runtime Semantics

#### Runtime Semantics: Program Evaluation

With argument *realm*.

*Program :* *ProgramBody<sub>opt</sub>* ▾

1. The code of this *Program* is strict mode code if the Directive Prologue (14.1) of its *ProgramBody* contains a Use Strict Directive or if any of the conditions of 10.1.1 apply. If the code of this *Program* is strict mode code, *ProgramBody* is evaluated in the following steps as strict mode code. Otherwise *ProgramBody* is evaluated in the following steps as non-strict mode code.
2. If *ProgramBody* is not present, return *NormalCompletion*(empty).
3. Let *progCxt* be a new execution context.
4. Set the *progCxt*'s *PreviousContext* to empty.
5. Set the *progCxt*'s *Realm* to *realm*.
6. Set the *progCxt*'s *VariableEnvironment* to *realm*'s Global Environment.
7. Set the *progCxt*'s *LexicalEnvironment* to *realm*'s Global Environment.
8. If *ProgramBody* is present, then
  - a. Perform Global Binding Instantiation as described in 10.5.x using the global code.
9. Let *result* be the result of evaluating *ProgramBody*.
10. The current execution context to empty.
11. Return *result*.

**NOTE** The processes for initiating the evaluation of a *Program* and for dealing with the result of such an evaluation are defined by an ECMAScript implementation and not by this specification.

The production *ProgramBody : StatementList* is evaluated as follows:

1. Return the result of evaluating *StatementList*.

#### 14.1 Directive Prologues and the Use Strict Directive

A Directive Prologue is the longest sequence of *ExpressionStatement* productions occurring as the initial *StatementListItem* productions of a *ProgramBody* or *FunctionBody* and where each *ExpressionStatement* in the sequence consists entirely of a *StringLiteral* token followed a semicolon. The semicolon may appear explicitly or may be inserted by automatic semicolon insertion. A Directive Prologue may be an empty sequence.

A Use Strict Directive is an *ExpressionStatement* in a Directive Prologue whose *StringLiteral* is either the exact character sequences `"use strict"` or `'use strict'`. A Use Strict Directive may not contain an *EscapeSequence* or *LineContinuation*.

<b>Deleted:</b> 13.2 Generator Definitions
<b>Syntax</b>
<i>GeneratorDeclaration</i> : Function * BindingIdentifier FormalParameterList ) { FunctionBody } GeneratorExpression : Function * BindingIdentifier <sub>opt</sub> FormalParameterList ) { FunctionBody }
<b>Deleted:</b> <i>SourceElements<sub>opt</sub></i>
<b>Deleted:</b> <i>SourceElements</i>
<b>Deleted:</b> <i>SourceElement</i> . <i>SourceElements SourceElement</i>
<b>Deleted:</b> <i>SourceElement</i> : Statement FunctionDeclaration
<b>Deleted:</b> <i>Semantics</i> The production
<b>Deleted:</b> <i>SourceElements<sub>opt</sub></i>
<b>Deleted:</b> is evaluated as follows:
<b>Deleted:</b> <i>SourceElements</i>
<b>Deleted:</b> <i>SourceElements</i>
<b>Deleted:</b> <i>SourceElements</i>
<b>Deleted:</b> <i>SourceElements</i>
<b>Deleted:</b> (normal, Deleted: Deleted: , empty Deleted: for global code as described in 10.4.1 <b>Comment [AWB783]:</b> fix
<b>Deleted:</b> <i>SourceElements</i>
<b>Comment [AWB684]:</b> This is where we can put something about what to do with unhandled exceptions
<b>Deleted:</b> Exit the execution context <i>progCxt</i> .
<b>Deleted:</b> <i>SourceElements</i>
<b>Deleted:</b> <i>SourceElements SourceElement</i>
<b>Deleted:</b> <#>Let <i>headResult</i> be the result of evaluating <i>SourceElements</i> . <#>If <i>headResult</i> is an abrupt completion, return <i>headResult</i> . <#>Let <i>tailResult</i> be result of evaluating <i>SourceElement</i> . <#>If <i>tailResult</i> .value is empty, let <i>V</i> = <i>headResult</i> .value, otherwise let <i>V</i> = <i>tailResult</i> .value <b>Deleted:</b> ( <i>tailResult</i> .type, <i>V</i> , <i>tailResult</i> .target)
<b>Deleted:</b> The production <i>SourceElement</i> : <i>Statement</i> is evaluated as follows: <#>Return the result of evaluating <i>Statement</i> . The production <i>SourceElement</i> : <i>FunctionDeclaration</i> is evaluated as follows: <#>Return (normal, empty, empty).
<b>Deleted:</b> as the initial <i>SourceElement</i> productions of a <i>Program</i> or

A Directive Prologue may contain more than one Use Strict Directive. However, an implementation may issue a warning if this occurs.

**NOTE** The *ExpressionStatement* productions of a Directive Prologue are evaluated normally during evaluation of the containing production. Implementations may define implementation specific meanings for *ExpressionStatement* productions which are not a Use Strict Directive and which occur in a Directive Prologue. If an appropriate notification mechanism exists, an implementation should issue a warning if it encounters in a Directive Prologue an *ExpressionStatement* that is not a Use Strict Directive or which does not have a meaning defined by the implementation.

**Deleted:** *SourceElements*

## 15 Standard Built-in ECMAScript Objects

There are certain built-in objects available whenever an ECMAScript program begins execution. One, the global object, is part of the lexical environment of the executing program. Others are accessible as initial properties of the global object.

Unless specified otherwise, a built-in object has the [[NativeBrand]] internal property with value NativeFunction if that built-in object has a [[Call]] internal property. Unless specified otherwise, the [[Extensible]] internal property of a built-in object initially has the value **true**.

Many built-in objects are functions: they can be invoked with arguments. Some of them furthermore are constructors: they are functions intended for use with the **new** operator. For each built-in function, this specification describes the arguments required by that function and properties of the Function object. For each built-in constructor, this specification furthermore describes properties of the prototype object of that constructor and properties of specific object instances returned by a **new** expression that invokes that constructor.

Unless otherwise specified in the description of a particular function, if a function or constructor described in this clause is given fewer arguments than the function is specified to require, the function or constructor shall behave exactly as if it had been given sufficient additional arguments, each such argument being the **undefined** value.

Unless otherwise specified in the description of a particular function, if a function or constructor described in this clause is given more arguments than the function is specified to allow, the extra arguments are evaluated by the call and then ignored by the function. However, an implementation may define implementation specific behaviour relating to such arguments as long as the behaviour is not the throwing of a **TypeError** exception that is predicated simply on the presence of an extra argument.

**NOTE** Implementations that add additional capabilities to the set of built-in functions are encouraged to do so by adding new functions rather than adding new parameters to existing functions.

Every built-in function and every built-in constructor has the Function prototype object, which is the initial value of the expression **Function.prototype** (15.3.4), as the value of its [[Prototype]] internal property.

Unless otherwise specified every built-in prototype object has the Object prototype object, which is the initial value of the expression **Object.prototype** (15.2.4), as the value of its [[Prototype]] internal property, except the Object prototype object itself.

None of the built-in functions described in this clause that are not constructors shall implement the [[Construct]] internal method unless otherwise specified in the description of a particular function. The behavior specified in this clause for each built-in function is the specification of the [[Call]] internal method behavior for that function. None of the built-in functions described in this clause shall have a **prototype** property unless otherwise specified in the description of a particular function.

This clause generally describes distinct behaviours for when a constructor is “called as a function” and for when it is “called as part of a **new** expression”. The “called as a function” behaviour corresponds to the invocation of the constructor’s [[Call]] internal method and the “called as part of a new expression” behaviour corresponds to the invocation of the constructor’s [[Construct]] internal method.

**Deleted:** the

**Deleted:** [[Class]] internal property of

**Deleted:** is "Function"

**Deleted:** IsFunction

**Deleted:** , or "Object" if that built-in object does not have a [[Call]] internal property

**Comment [AW85]:** [https://bugs.ecmascript.org/show\\_bug.cgi?id=155](https://bugs.ecmascript.org/show_bug.cgi?id=155)

Every built-in Function object described in this clause—whether as a constructor, an ordinary function, or both—has a `length` property whose value is an integer. Unless otherwise specified, this value is equal to the largest number of named arguments shown in the subclause headings for the function description, including optional parameters.

**NOTE** For example, the Function object that is the initial value of the `slice` property of the String prototype object is described under the subclause heading “`String.prototype.slice (start, end)`” which shows the two named arguments `start` and `end`; therefore the value of the `length` property of that Function object is 2.

In every case, the `length` property of a built-in Function object described in this clause has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }`. Every other property described in this clause has the attributes `{ [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true }` unless otherwise specified.

## 15.1 The Global Object

The unique *global object* is created before control enters any execution context.

Unless otherwise specified, the standard built-in properties of the global object have attributes `{ [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true }`.

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

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

The value of the `[[Prototype]]` internal property of the global object is implementation-dependent.

In addition to the properties defined in this specification the global object may have additional host defined properties. This may include a property whose value is the global object itself; for example, in the HTML document object model the `window` property of the global object is the global object itself.

**Deleted:** s  
**Deleted:** and `[[Class]]`  
**Deleted:** ies  
**Deleted:** are

### 15.1.1 Value Properties of the Global Object

#### 15.1.1.1 `NaN`

The value of `NaN` is `NaN` (see 8.5). This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }`.

#### 15.1.1.2 `Infinity`

The value of `Infinity` is `+∞` (see 8.5). This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }`.

#### 15.1.1.3 `undefined`

The value of `undefined` is `undefined` (see 8.1). This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }`.

### 15.1.2 Function Properties of the Global Object

#### 15.1.2.1 `eval (x)`

When the `eval` function is called with one argument `x`, the following steps are taken:

1. If `Type(x)` is not `String`, return `x`.

2. Let *prog* be the ECMAScript code that is the result of parsing *x* for the goal symbol *Program*. If the parse fails, throw a **SyntaxError** exception (but see also clause 16).
3. Let *evalCtx* be the result of establishing a new execution context (10.4.2) for the eval code *prog*.
  
4. If there is no calling context or if the eval code is not being evaluated by a direct call (15.1.2.1.1) to the eval function then,
  - a. Initialise the execution context as if it was a global execution context using the eval code as *C* as described in 10.4.1.1.
5. Else,
  - a. Set the *ThisBinding* to the same value as the *ThisBinding* of the calling execution context.
  - b. Set the *LexicalEnvironment* to the same value as the *LexicalEnvironment* of the calling execution context.
  - c. Set the *VariableEnvironment* to the same value as the *VariableEnvironment* of the calling execution context.
6. If the eval code is strict code, then
  - a. Let *strictVarEnv* be the result of calling *NewDeclarativeEnvironment* passing the *LexicalEnvironment* as the argument.
  - b. Set the *LexicalEnvironment* to *strictVarEnv*.
  - c. Set the *VariableEnvironment* to *strictVarEnv*.
7. Perform Declaration Binding Instantiation as described in 10.5 using the eval code.
  
8. Let *result* be the result of evaluating the program *prog*.
9. Exit the running execution context *evalCtx*, restoring the previous execution context.
10. If *result.type* is **normal** and its completion value is a value *V*, then return the value *V*.
11. If *result.type* is **normal** and its completion value is **empty**, then return the value **undefined**.
12. Otherwise, *result.type* must be **throw**. Throw *result.value* as an exception.

**NOTE** The eval code cannot instantiate variable or function bindings in the variable environment of the calling context that invoked the eval if either the code of the calling context or the eval code is strict code. Instead such bindings are instantiated in a new VariableEnvironment that is only accessible to the eval code.

**Deleted:** as a

**Comment [AWB786]:** TODO: need to finish refactoring after completing global scope instantiation work.

### 15.1.2.1.1 Direct Call to Eval

A direct call to the eval function is one that is expressed as a *CallExpression* that meets the following two conditions:

The Reference that is the result of evaluating the *MemberExpression* in the *CallExpression* has an environment record as its base value and its reference name is "**eval**".

The result of calling the abstract operation *GetValue* with that Reference as the argument is the standard built-in function defined in 15.1.2.1.

### 15.1.2.2 parseInt (string , radix)

The **parseInt** function produces an integer value dictated by interpretation of the contents of the *string* argument according to the specified *radix*. Leading white space in *string* is ignored. If *radix* is **undefined** or 0, it is assumed to be 10 except when the number begins with the character pairs **0x** or **0X**, in which case a radix of 16 is assumed. If *radix* is 16, the number may also optionally begin with the character pairs **0x** or **0X**.

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

1. Let *inputString* be *ToString(string)*.
2. ReturnIfAbrupt(string).
3. Let *S* be a newly created substring of *inputString* consisting of the first character that is not a *StrWhiteSpaceChar* and all characters following that character. (In other words, remove leading white space.) If *inputString* does not contain any such characters, let *S* be the empty string.
4. Let *sign* be 1.
5. If *S* is not empty and the first character of *S* is a minus sign **-**, let *sign* be **-1**.

6. If  $S$  is not empty and the first character of  $S$  is a plus sign + or a minus sign -, then remove the first character from  $S$ .
7. Let  $R = \text{ToInt32}(radix)$ .
8. `ReturnIfAbrupt(R)`.
9. Let `stripPrefix` be **true**.
10. If  $R \neq 0$ , then
  - a. If  $R < 2$  or  $R > 36$ , then return **NaN**.
  - b. If  $R \neq 16$ , let `stripPrefix` be **false**.
11. Else,  $R = 0$ 
  - a. Let  $R = 10$ .
12. If `stripPrefix` is **true**, then
  - a. If the length of  $S$  is at least 2 and the first two characters of  $S$  are either "0x" or "0X", then remove the first two characters from  $S$  and let  $R = 16$ .
13. If  $S$  contains any character that is not a radix- $R$  digit, then let  $Z$  be the substring of  $S$  consisting of all characters before the first such character; otherwise, let  $Z$  be  $S$ .
14. If  $Z$  is empty, return **NaN**.
15. Let  $mathInt$  be the mathematical integer value that is represented by  $Z$  in radix- $R$  notation, using the letters A-Z and a-z for digits with values 10 through 35. (However, if  $R$  is 10 and  $Z$  contains more than 20 significant digits, every significant digit after the 20th may be replaced by a 0 digit, at the option of the implementation; and if  $R$  is not 2, 4, 8, 10, 16, or 32, then  $mathInt$  may be an implementation-dependent approximation to the mathematical integer value that is represented by  $Z$  in radix- $R$  notation.)
16. Let  $number$  be the Number value for  $mathInt$ .
17. Return  $sign \times number$ .

NOTE `parseInt` may interpret only a leading portion of  $string$  as an integer value; it ignores any characters that cannot be interpreted as part of the notation of an integer, and no indication is given that any such characters were ignored.

#### 15.1.2.3 `parseFloat` (*string*)

The `parseFloat` function produces a Number value dictated by interpretation of the contents of the *string* argument as a decimal literal.

When the `parseFloat` function is called, the following steps are taken:

1. Let  $inputString$  be `ToString(string)`.
2. `ReturnIfAbrupt(string)`.
3. Let  $trimmedString$  be a substring of  $inputString$  consisting of the leftmost character that is not a *StrWhiteSpaceChar* and all characters to the right of that character. (In other words, remove leading white space.) If  $inputString$  does not contain any such characters, let  $trimmedString$  be the empty string.
4. If neither  $trimmedString$  nor any prefix of  $trimmedString$  satisfies the syntax of a *StrDecimalLiteral* (see 9.3.1), return **NaN**.
5. Let  $numberString$  be the longest prefix of  $trimmedString$ , which might be  $trimmedString$  itself, that satisfies the syntax of a *StrDecimalLiteral*.
6. Return the Number value for the MV of  $numberString$ .

NOTE `parseFloat` may interpret only a leading portion of  $string$  as a Number value; it ignores any characters that cannot be interpreted as part of the notation of an decimal literal, and no indication is given that any such characters were ignored.

#### 15.1.2.4 `isNaN` (*number*)

Returns **true** if the argument coerces to **NaN**, and otherwise returns **false**.

1. Let  $num$  be `ToNumber(number)`.
2. ReturnIfAbrupt(num).
3. If  $num$  is **NaN**, return **true**.
4. Otherwise, return **false**.

**Deleted:** `ToNumber(number)`

NOTE A reliable way for ECMAScript code to test if a value  $x$  is a **NaN** is an expression of the form  $x \neq x$ . The result will be **true** if and only if  $x$  is a **NaN**.

### 15.1.2.5 `isFinite (number)`

Returns `false` if the argument coerces to `NaN`, `+∞`, or `-∞`, and otherwise returns `true`.

1. Let `num` be `ToNumber(number)`.
2. ReturnIfAbrupt(`num`).
3. If `ToNumber(num)` is `NaN`, `+∞`, or `-∞`, return `false`.
4. Otherwise, return `true`.

**Deleted:** ber

### 15.1.3 URI Handling Function Properties

Uniform Resource Identifiers, or URIs, are Strings that identify resources (e.g. web pages or files) and transport protocols by which to access them (e.g. HTTP or FTP) on the Internet. The ECMAScript language itself does not provide any support for using URIs except for functions that encode and decode URIs as described in 15.1.3.1, 15.1.3.2, 15.1.3.3 and 15.1.3.4.

**NOTE** Many implementations of ECMAScript provide additional functions and methods that manipulate web pages; these functions are beyond the scope of this standard.

A URI is composed of a sequence of components separated by component separators. The general form is:

*Scheme* : *First* / *Second* ; *Third* ? *Fourth*

where the italicised names represent components and “`:`”, “`/`”, “`;`” and “`?`” are reserved characters used as separators. The `encodeURI` and `decodeURI` functions are intended to work with complete URIs; they assume that any reserved characters in the URI are intended to have special meaning and so are not encoded. The `encodeURIComponent` and `decodeURIComponent` functions are intended to work with the individual component parts of a URI; they assume that any reserved characters represent text and so must be encoded so that they are not interpreted as reserved characters when the component is part of a complete URI.

The following lexical grammar specifies the form of encoded URIs.

#### Syntax

```

uri ::= uriCharactersopt

uriCharacters ::= uriCharacter uriCharactersopt

uriCharacter ::= uriReserved
                uriUnescaped
                uriEscaped

uriReserved ::= one of
              ; / ? : @ & = + $ ,

uriUnescaped ::= uriAlpha
                DecimalDigit
                uriMark

uriEscaped ::= % HexDigit HexDigit

uriAlpha ::= one of
          a b c d e f g h i j k l m n o p q r s t u v w x y z
          A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

```

*uriMark* :::: one of

- \_ . ! ~ \* ' ( )

NOTE The above syntax is based upon RFC 2396 and does not reflect changes introduced by the more recent RFC 3986.

When a character to be included in a URI is not listed above or is not intended to have the special meaning sometimes given to the reserved characters, that character must be encoded. The character is transformed into its UTF-8 encoding, with surrogate pairs first converted from UTF-16 to the corresponding code point value. (Note that for code units in the range [0,127] this results in a single octet with the same value.) The resulting sequence of octets is then transformed into a String with each octet represented by an escape sequence of the form "%xx".

The encoding and escaping process is described by the abstract operation `Encode` taking two String arguments *string* and *unescapeSet*.

1. Let *strLen* be the number of characters in *string*.
2. Let *R* be the empty String.
3. Let *k* be 0.
4. Repeat
  - a. If *k* equals *strLen*, return *R*.
  - b. Let *C* be the character at position *k* within *string*.
  - c. If *C* is in *unescapeSet*, then
    - i. Let *S* be a String containing only the character *C*.
    - ii. Let *R* be a new String value computed by concatenating the previous value of *R* and *S*.
  - d. Else, *C* is not in *unescapeSet*
    - i. If the code unit value of *C* is not less than 0xDC00 and not greater than 0xDFFF, throw a **URIError** exception.
    - ii. If the code unit value of *C* is less than 0xD800 or greater than 0xDBFF, then
      1. Let *V* be the code unit value of *C*.
    - iii. Else,
      1. Increase *k* by 1.
      2. If *k* equals *strLen*, throw a **URIError** exception.
      3. Let *kChar* be the code unit value of the character at position *k* within *string*.
      4. If *kChar* is less than 0xDC00 or greater than 0xDFFF, throw a **URIError** exception.
    5. Let *V* be (((the code unit value of *C*) - 0xD800) × 0x400 + (*kChar* - 0xDC00) + 0x10000).
  - iv. Let *Octets* be the array of octets resulting by applying the UTF-8 transformation to *V*, and let *L* be the array size.
  - v. Let *j* be 0.
  - vi. Repeat, while *j* < *L*
    1. Let *jOctet* be the value at position *j* within *Octets*.
    2. Let *S* be a String containing three characters "%XY" where *XY* are two uppercase hexadecimal digits encoding the value of *jOctet*.
    3. Let *R* be a new String value computed by concatenating the previous value of *R* and *S*.
    4. Increase *j* by 1.
- e. Increase *k* by 1.

The unescaping and decoding process is described by the abstract operation `Decode` taking two String arguments *string* and *reservedSet*.

1. Let *strLen* be the number of characters in *string*.
2. Let *R* be the empty String.
3. Let *k* be 0.
4. Repeat
  - a. If *k* equals *strLen*, return *R*.
  - b. Let *C* be the character at position *k* within *string*.
  - c. If *C* is not '%' , then

- i. Let  $S$  be the String containing only the character  $C$ .
- d. Else,  $C$  is '%'
  - i. Let  $start$  be  $k$ .
  - ii. If  $k + 2$  is greater than or equal to  $strLen$ , throw a **URIError** exception.
  - iii. If the characters at position  $(k+1)$  and  $(k + 2)$  within  $string$  do not represent hexadecimal digits, throw a **URIError** exception.
  - iv. Let  $B$  be the 8-bit value represented by the two hexadecimal digits at position  $(k + 1)$  and  $(k + 2)$ .
  - v. Increment  $k$  by 2.
  - vi. If the most significant bit in  $B$  is 0, then
    - 1. Let  $C$  be the character with code unit value  $B$ .
    - 2. If  $C$  is not in  $reservedSet$ , then
      - a. Let  $S$  be the String containing only the character  $C$ .
    - 3. Else,  $C$  is in  $reservedSet$ 
      - a. Let  $S$  be the substring of  $string$  from position  $start$  to position  $k$  included.
  - vii. Else, the most significant bit in  $B$  is 1
    - 1. Let  $n$  be the smallest non-negative number such that  $(B << n) \& 0x80$  is equal to 0.
    - 2. If  $n$  equals 1 or  $n$  is greater than 4, throw a **URIError** exception.
    - 3. Let  $Octets$  be an array of 8-bit integers of size  $n$ .
    - 4. Put  $B$  into  $Octets$  at position 0.
    - 5. If  $k + (3 \times (n - 1))$  is greater than or equal to  $strLen$ , throw a **URIError** exception.
    - 6. Let  $j$  be 1.
    - 7. Repeat, while  $j < n$ 
      - a. Increment  $k$  by 1.
      - b. If the character at position  $k$  is not '%', throw a **URIError** exception.
      - c. If the characters at position  $(k + 1)$  and  $(k + 2)$  within  $string$  do not represent hexadecimal digits, throw a **URIError** exception.
      - d. Let  $B$  be the 8-bit value represented by the two hexadecimal digits at position  $(k + 1)$  and  $(k + 2)$ .
      - e. If the two most significant bits in  $B$  are not 10, throw a **URIError** exception.
      - f. Increment  $k$  by 2.
      - g. Put  $B$  into  $Octets$  at position  $j$ .
      - h. Increment  $j$  by 1.
    - 8. Let  $V$  be the value obtained by applying the UTF-8 transformation to  $Octets$ , that is, from an array of octets into a 21-bit value. If  $Octets$  does not contain a valid UTF-8 encoding of a Unicode code point throw an **URIError** exception.
    - 9. If  $V$  is less than  $0x10000$ , then
      - a. Let  $C$  be the character with code unit value  $V$ .
      - b. If  $C$  is not in  $reservedSet$ , then
        - i. Let  $S$  be the String containing only the character  $C$ .
      - c. Else,  $C$  is in  $reservedSet$ 
        - i. Let  $S$  be the substring of  $string$  from position  $start$  to position  $k$  included.
    - 10. Else,  $V$  is  $\geq 0x10000$ 
      - a. Let  $L$  be  $((V - 0x10000) \& 0x3FF) + 0xDC00$ .
      - b. Let  $H$  be  $((((V - 0x10000) >> 10) \& 0x3FF) + 0xD800$ .
      - c. Let  $S$  be the String containing the two characters with code unit values  $H$  and  $L$ .
  - e. Let  $R$  be a new String value computed by concatenating the previous value of  $R$  and  $S$ .
  - f. Increase  $k$  by 1.

**NOTE** This syntax of Uniform Resource Identifiers is based upon RFC 2396 and does not reflect the more recent RFC 3986 which replaces RFC 2396. A formal description and implementation of UTF-8 is given in RFC 3629.

In UTF-8, characters are encoded using sequences of 1 to 6 octets. The only octet of a "sequence" of one has the higher-order bit set to 0, the remaining 7 bits being used to encode the character value. In a sequence of  $n$  octets,  $n \geq 1$ , the initial octet has the  $n$  higher-order bits set to 1, followed by a bit set to 0. The remaining bits of that octet contain bits from the value of the character to be encoded. The following octets all have the higher-order bit set to 1 and the following bit set to

0, leaving 6 bits in each to contain bits from the character to be encoded. The possible UTF-8 encodings of ECMAScript characters are specified in Table 22.

**Deleted:** 21

**Deleted:** 21

**Table 22 — UTF-8 Encodings**

<b>Code Unit Value</b>	<b>Representation</b>	<b>1<sup>st</sup> Octet</b>	<b>2<sup>nd</sup> Octet</b>	<b>3<sup>rd</sup> Octet</b>	<b>4<sup>th</sup> Octet</b>
0x0000 – 0x007F	00000000 0zzzzzzz	0zzzzzzz			
0x0080 – 0x07FF	00000yyy yyzzzzzz	110yyyyy	10zzzzzz		
0x0800 – 0xD7FF	xxxxyyyy yyzzzzzz	1110xxxx	10yyyyyy	10zzzzzz	
0xD800 – 0xDBFF followed by 0xDC00 – 0xDFFF	110110vv vvvwwwwxx followed by 110111yy yyzzzzzz	11110uuu	10uuwwww	10xxyyyy	10zzzzzz
0xD800 – 0xDBFF not followed by 0xDC00 – 0xDFFF	causes URIError				
0xDC00 – 0xDFFF	causes URIError				
0xE000 – 0xFFFF	xxxxyyyy yyzzzzzz	1110xxxx	10yyyyyy	10zzzzzz	

Where

$$uuuuu = vvvv + 1$$

to account for the addition of 0x10000 as in Surrogates, section 3.7, of the Unicode Standard.

The range of code unit values 0xD800-0xFFFF is used to encode surrogate pairs; the above transformation combines a UTF-16 surrogate pair into a UTF-32 representation and encodes the resulting 21-bit value in UTF-8. Decoding reconstructs the surrogate pair.

RFC 3629 prohibits the decoding of invalid UTF-8 octet sequences. For example, the invalid sequence C0 80 must not decode into the character U+0000. Implementations of the Decode algorithm are required to throw a **URIError** when encountering such invalid sequences.

#### 15.1.3.1 `decodeURI(encodedURI)`

The `decodeURI` function computes a new version of a URI in which each escape sequence and UTF-8 encoding of the sort that might be introduced by the `encodeURI` function is replaced with the character that it represents. Escape sequences that could not have been introduced by `encodeURI` are not replaced.

When the `decodeURI` function is called with one argument *encodedURI*, the following steps are taken:

1. Let *uriString* be `ToString(encodedURI)`.
2. **ReturnIfAbrupt(uriString)**.
3. Let *reservedURISet* be a String containing one instance of each character valid in *uriReserved* plus “#”.
4. Return the result of calling `Decode(uriString, reservedURISet)`

**NOTE** The character “#” is not decoded from escape sequences even though it is not a reserved URI character.

#### 15.1.3.2 `decodeURIComponent(encodedURIComponent)`

The `decodeURIComponent` function computes a new version of a URI in which each escape sequence and UTF-8 encoding of the sort that might be introduced by the `encodeURIComponent` function is replaced with the character that it represents.

When the `decodeURIComponent` function is called with one argument *encodedURIComponent*, the following steps are taken:

1. Let *componentString* be `ToString(encodedURIComponent)`.
2. **ReturnIfAbrupt(componentString)**.
3. Let *reservedURIComponentSet* be the empty String.
4. Return the result of calling `Decode(componentString, reservedURIComponentSet)`

### 15.1.3.3 `encodeURI (uri)`

The `encodeURI` function computes a new version of a URI in which each instance of certain characters is replaced by one, two, three, or four escape sequences representing the UTF-8 encoding of the character.

When the `encodeURI` function is called with one argument *uri*, the following steps are taken:

1. Let *uriString* be `ToString(uri)`.
2. ReturnIfAbrupt(*uriString*).
3. Let *unescapeURISet* be a String containing one instance of each character valid in *uriReserved* and *uriUnescaped* plus “#”.
4. Return the result of calling `Encode(uriString, unescapeURISet)`

NOTE The character “#” is not encoded to an escape sequence even though it is not a reserved or unescaped URI character.

### 15.1.3.4 `encodeURIComponent (uriComponent)`

The `encodeURIComponent` function computes a new version of a URI in which each instance of certain characters is replaced by one, two, three, or four escape sequences representing the UTF-8 encoding of the character.

When the `encodeURIComponent` function is called with one argument *uriComponent*, the following steps are taken:

1. Let *componentString* be `ToString(uriComponent)`.
2. ReturnIfAbrupt(*componentString*).
3. Let *unescapeURIComponentSet* be a String containing one instance of each character valid in *uriUnescaped*.
4. Return the result of calling `Encode(componentString, unescapeURIComponentSet)`

## 15.1.4 Constructor Properties of the Global Object

### 15.1.4.1 `Object ( . . . )`

See 15.2.1 and 15.2.2.

### 15.1.4.2 `Function ( . . . )`

See 15.3.1 and 15.3.2.

### 15.1.4.3 `Array ( . . . )`

See 15.4.1 and 15.4.2.

### 15.1.4.4 `String ( . . . )`

See 15.5.1 and 15.5.2.

### 15.1.4.5 `Boolean ( . . . )`

See 15.6.1 and 15.6.2.

### 15.1.4.6 `Number ( . . . )`

See 15.7.1 and 15.7.2.

#### **15.1.4.7 Date ( . . . )**

See 15.9.2.

#### **15.1.4.8 RegExp ( . . . )**

See 15.10.3 and 15.10.4.

#### **15.1.4.9 Error ( . . . )**

See 15.11.1 and 15.11.2.

#### **15.1.4.10 EvalError ( . . . )**

See 15.11.6.1.

#### **15.1.4.11 RangeError ( . . . )**

See 15.11.6.2.

#### **15.1.4.12 ReferenceError ( . . . )**

See 15.11.6.3.

#### **15.1.4.13 SyntaxError ( . . . )**

See 15.11.6.4.

#### **15.1.4.14 TypeError ( . . . )**

See 15.11.6.5.

#### **15.1.4.15 URIError ( . . . )**

See 15.11.6.6.

### **15.1.5 Other Properties of the Global Object**

#### **15.1.5.1 Math**

See 15.8.

#### **15.1.5.2 JSON**

See 15.12.

## **15.2 Object Objects**

The abstract operation `ObjectCreate` with optional argument `proto` (an object or null) is used to specify the creation of new Object objects. It performs the following steps:

1. If `proto` was not provided, let `proto` be the intrinsic %ObjectPrototype%.
2. Let `obj` be a newly created ECMAScript object.
3. Set the `[[Prototype]]` internal property of `obj` to `proto`.
4. Set `obj`'s common internal methods to the default definitions specified in 8.12.
5. Set the `[[Extensible]]` internal property of `obj` to `true`.
6. Return `obj`.

### 15.2.1 The Object Constructor Called as a Function

When `Object` is called as a function rather than as a constructor, it performs a type conversion.

#### 15.2.1.1 `Object( [ value ] )`

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

1. If `value` is `null`, `undefined` or not supplied, return [the result of the abstraction operation `ObjectCreate`](#).
2. Return `ToObject(value)`.

### 15.2.2 The Object Constructor

When `Object` is called as part of a `new` expression, it is a constructor that may create an object.

#### 15.2.2.1 `new Object( [ value ] )`

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

1. If `value` is supplied, then
  - a. If `Type(value)` is `Object`, then
    - i. If the `value` is a native ECMAScript object, do not create a new object but simply return `value`.
    - ii. If the `value` is a host object, then actions are taken and a result is returned in an implementation-dependent manner that may depend on the host object.
  - b. If `Type(value)` is `String`, return `ToObject(value)`.
  - c. If `Type(value)` is `Boolean`, return `ToObject(value)`.
  - d. If `Type(value)` is `Number`, return `ToObject(value)`.
2. Assert: The argument `value` was not supplied or its type was `Null` or `Undefined`.
3. Return [the result of the abstraction operation `ObjectCreate`](#).

### 15.2.3 Properties of the Object Constructor

The value of the `[[Prototype]]` internal property of the `Object` constructor is the standard built-in `Function` prototype object.

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

#### 15.2.3.1 `Object.prototype`

The initial value of `Object.prototype` is the standard built-in `Object` prototype object (15.2.4).

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

#### 15.2.3.2 `Object.getPrototypeOf(O)`

When the `getPrototypeOf` function is called with argument `O`, the following steps are taken:

1. If `Type(O)` is not `Object` throw a `TypeError` exception.
2. Return the value of the `[[Prototype]]` internal property of `O`.

#### 15.2.3.3 `Object.getOwnPropertyDescriptor(O, P)`

When the `getOwnPropertyDescriptor` function is called, the following steps are taken:

1. If `Type(O)` is not `Object` throw a `TypeError` exception.

**Deleted:** create and

**Deleted:** a new `Object` object exactly as if the standard built-in `Object` constructor had been called with the same arguments (15.2.2.1)

**Deleted:** <#>Let `obj` be a newly created native ECMAScript object.  
 <#>Set the `[[Prototype]]` internal property of `obj` to the standard built-in `Object` prototype object (15.2.4).  
 <#>Set the `[[Class]]` internal property of `obj` to "`Object`".

<#>Set the `[[Extensible]]` internal property of `obj` to `true`.  
 <#>Set all the internal methods of `obj` as specified in 8.12.

**Deleted:** `obj`

2. Let *name* be *ToString*(*P*).
3. **ReturnIfAbrupt**(*name*).
4. Let *desc* be the result of calling the **[[GetOwnProperty]]** internal method of *O* with argument *name*.
5. Return the result of calling *FromPropertyDescriptor*(*desc*) (8.10.4).

#### 15.2.3.4 Object.getOwnPropertyNames ( *O* )

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

1. If *Type*(*O*) is not *Object* throw a **TypeError** exception.
2. Let *array* be **the result of the abstraction operation ArrayCreate (15.4) with argument 0**.
3. Let *n* be 0.
4. For each named own property *P* of *O*
  - a. Let *name* be the String value that is the name of *P*.
  - b. Call the **[[DefineOwnProperty]]** internal method of *array* with arguments *ToString*(*n*), the *PropertyDescriptor* **{[[Value]]: *name*, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}**, and **false**.
  - c. Increment *n* by 1.
5. Return *array*.

NOTE If *O* is a String instance, the set of own properties processed in step 4 includes the implicit properties defined in 15.5.5.2 that correspond to character positions within the object's **[[PrimitiveValue]]** String.

**Deleted:** the result of creating a new object as if the expression **new Array ()** where **Array** is the standard built-in constructor with that name

#### 15.2.3.5 Object.create ( *O* [, *Properties*] )

The **create** function creates a new object with a specified prototype. When the **create** function is called, the following steps are taken:

1. If *Type*(*O*) is not *Object* or *Null* throw a **TypeError** exception.
2. Let *obj* be **the result of the abstraction operation ObjectCreate with argument O**.
3. If the argument *Properties* is present and not **undefined**, then
  - a. **Return the result of the abstract operation ObjectDefineProperties** with arguments *obj* and *Properties*.
4. Return *obj*.

**Deleted:** the result of creating a new object as if the expression **new Object()** where **Object** is the standard built-in constructor with that name

**Deleted:** <#> Set the **[[Prototype]]** internal property of *obj* to *O*. ¶

**Deleted:** add own properties to *obj* as if by calling the standard built-in function

**Deleted:** **object.defineProperties**

#### 15.2.3.6 Object.defineProperty ( *O*, *P*, *Attributes* )

The **defineProperty** function is used to add an own property and/or update the attributes of an existing own property of an object. When the **defineProperty** function is called, the following steps are taken:

1. If *Type*(*O*) is not *Object* throw a **TypeError** exception.
2. Let *name* be *ToString*(*P*).
3. **ReturnIfAbrupt**(*name*).
4. Let *desc* be the result of calling *ToPropertyDescriptor* with *Attributes* as the argument.
5. Call the **[[DefineOwnProperty]]** internal method of *O* with arguments *name*, *desc*, and **true**.
6. Return *O*.

#### 15.2.3.7 Object.defineProperties ( *O*, *Properties* )

The **defineProperties** function is used to add own properties and/or update the attributes of existing own properties of an object. When the **defineProperties** function is called, the following steps are taken:

1. **Return the result of the abstract operation ObjectDefineProperties with arguments *O* and *Properties*.**

**Formatted:** Font: Italic

**Formatted:** Font: Italic

**The abstract operations ObjectDefineProperties with arguments *O* and *Properties* performs the following steps:**

1. If *Type*(*O*) is not *Object* throw a **TypeError** exception.
2. Let *props* be *ToObject*(*Properties*).
3. Let *names* be an internal list containing the names of each enumerable own property of *props*.

4. Let *descriptors* be an empty internal List.
5. For each element *P* of *names* in list order,
  - a. Let *descObj* be the result of calling the [[Get]] internal method of *props* with *P* as the argument.
  - b. [ReturnIfAbrupt\(descObj\)](#).
  - c. Let *desc* be the result of calling ToPropertyDescriptor with *descObj* as the argument.
  - d. [ReturnIfAbrupt\(desc\)](#).
  - e. Append the pair (a two element List) consisting of *P* and *desc* to the end of *descriptors*.
6. For each *pair* from *descriptors* in list order,
  - a. Let *P* be the first element of *pair*.
  - b. Let *desc* be the second element of *pair*.
  - c. [Let status be the result of calling](#) the [[DefineOwnProperty]] internal method of *O* with arguments *P*, *desc*, and **true**.
  - d. [ReturnIfAbrupt\(status\)](#).
7. Return *O*.

If an implementation defines a specific order of enumeration for the for-in statement, that same enumeration order must be used to order the list elements in step 3 of this algorithm.

**Deleted:** Call

#### 15.2.3.8 Object.seal ( *O* )

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

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. For each named own property name *P* of *O*,
  - a. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with *P*.
  - b. If *desc*.[[Configurable]] is **true**, set *desc*.[[Configurable]] to **false**.
  - c. [Let status be the result of calling](#) the [[DefineOwnProperty]] internal method of *O* with *P*, *desc*, and **true** as arguments.
  - d. [ReturnIfAbrupt\(status\)](#).
3. Set the [[Extensible]] internal property of *O* to **false**.
4. Return *O*.

**Deleted:** Call

#### 15.2.3.9 Object.freeze ( *O* )

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

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. For each named own property name *P* of *O*,
  - a. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with *P*.
  - b. If IsDataDescriptor(*desc*) is **true**, then
    - i. If *desc*.[[Writable]] is **true**, set *desc*.[[Writable]] to **false**.
  - c. If *desc*.[[Configurable]] is **true**, set *desc*.[[Configurable]] to **false**.
  - d. [Let status be the result of calling](#) the [[DefineOwnProperty]] internal method of *O* with *P*, *desc*, and **true** as arguments.
  - e. [ReturnIfAbrupt\(status\)](#).
3. Set the [[Extensible]] internal property of *O* to **false**.
4. Return *O*.

**Deleted:** Call

#### 15.2.3.10 Object.preventExtensions ( *O* )

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

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. Set the [[Extensible]] internal property of *O* to **false**.
3. Return *O*.

#### 15.2.3.11 Object.isSealed ( *O* )

When the **isSealed** function is called with argument *O*, the following steps are taken:

1. If Type(*O*) is not Object throw a **TypeError** exception.

2. For each named own property name  $P$  of  $O$ ,
  - a. Let  $desc$  be the result of calling the `[[GetOwnProperty]]` internal method of  $O$  with  $P$ .
    - b. If  $desc.[[Configurable]]$  is **true**, then return **false**.
3. If the `[[Extensible]]` internal property of  $O$  is **false**, then return **true**.
4. Otherwise, return **false**.

#### **15.2.3.12 Object.isFrozen ( $O$ )**

When the **isFrozen** function is called with argument  $O$ , the following steps are taken:

1. If `Type( $O$ )` is not `Object` throw a **TypeError** exception.
2. For each named own property name  $P$  of  $O$ ,
  - a. Let  $desc$  be the result of calling the `[[GetOwnProperty]]` internal method of  $O$  with  $P$ .
    - b. If `IsDataDescriptor(desc)` is **true** then
      - i. If  $desc.[[Writable]]$  is **true**, return **false**.
      - c. If  $desc.[[Configurable]]$  is **true**, then return **false**.
3. If the `[[Extensible]]` internal property of  $O$  is **false**, then return **true**.
4. Otherwise, return **false**.

#### **15.2.3.13 Object.isExtensible ( $O$ )**

When the **isExtensible** function is called with argument  $O$ , the following steps are taken:

1. If `Type( $O$ )` is not `Object` throw a **TypeError** exception.
2. Return the Boolean value of the `[[Extensible]]` internal property of  $O$ .

#### **15.2.3.14 Object.keys ( $O$ )**

When the **keys** function is called with argument  $O$ , the following steps are taken:

1. If the `Type( $O$ )` is not `Object`, throw a **TypeError** exception.
2. Let  $n$  be the number of own enumerable properties of  $O$ .
3. Let  $array$  be [the result of the abstraction operation `ArrayCreate \(15.4\)` with argument  \$n\$](#) .
4. Let  $index$  be 0.
5. For each own enumerable property of  $O$  whose name String is  $P$ 
  - a. Call the `[[DefineOwnProperty]]` internal method of  $array$  with arguments `ToString(index)`, the PropertyDescriptor `{[[Value]]:  $P$ , [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, and **false**.
  - b. Increment  $index$  by 1.
6. Return  $array$ .

**Deleted:** the result of creating a new Object as is by the expression `new Array(n)` where `Array` is the standard built-in constructor with that name

If an implementation defines a specific order of enumeration for the `for-in` statement, that same enumeration order must be used in step 5 of this algorithm.

#### **15.2.3.15 Object.isObject ( $O$ )**

When the **isObject** function is called with argument  $O$ , the following steps are taken:

1. If `Type( $O$ )` is `Object` return **true**.
2. Return **false**.

#### **15.2.4 Properties of the Object Prototype Object**

The value of the `[[Prototype]]` internal property of the Object prototype object is **null**, and the initial value of the `[[Extensible]]` internal property is **true**.

**Deleted:** , the value of the `[[Class]]` internal property is "Object".

##### **15.2.4.1 Object.prototype.constructor**

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

#### 15.2.4.2 Object.prototype.toString ()

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

1. If the `this` value is `undefined`, return "[object Undefined]".
2. If the `this` value is `null`, return "[object Null]".
3. Let  $O$  be the result of calling `ToObject` passing the `this` value as the argument.
4. If  $O$  has a [[NativeBrand]] internal property, let `tag` be the corresponding value from the Table 23.
5. Else if  $O$  is a host object, let `tag` be an implementation-defined string value. This value may not be "`Array`", "`Boolean`", "`Date`", "`Error`", "`Function`", "`JSON`", "`Math`", "`Number`", "`Object`", "`RegExp`", or "`String`".
6. Else, let `tag` be the string value "`Object`".
7. Return the String value that is the result of concatenating the three Strings "[object ", `tag`, and "]".

**Table 23 — Tags for Classified Native Objects**

[[NativeBrand]] Value	<code>tag</code> Value
<code>NativeFunction</code>	" <code>Function</code> "
<code>NativeArray</code>	" <code>Array</code> "
<code>StringWrapper</code>	" <code>String</code> "
<code>BooleanWrapper</code>	" <code>Boolean</code> "
<code>NumberWrapper</code>	" <code>Number</code> "
<code>NativeMath</code>	" <code>Math</code> "
<code>NativeDate</code>	" <code>Date</code> "
<code>NativeRegExp</code>	" <code>RegExp</code> "
<code>NativeError</code>	" <code>Error</code> "
<code>NativeJSON</code>	" <code>JSON</code> "
<code>NativeArguments</code>	" <code>Arguments</code> "

#### 15.2.4.3 Object.prototype.toLocaleString ()

When the `toLocaleString` method is called, the following steps are taken:

1. Let  $O$  be the `this` value.
2. ReturnIfAbrupt( $O$ ).
3. Return the result of the abstract operation Invoke with arguments "`toString`",  $O$  and an empty arguments List.

NOTE 1 This function is provided to give all Objects a generic `toLocaleString` interface, even though not all may use it. Currently, `Array`, `Number`, and `Date` provide their own locale-sensitive `toLocaleString` methods.

NOTE 2 The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

#### 15.2.4.4 Object.prototype.valueOf ()

When the `valueOf` method is called, the following steps are taken:

1. Let  $O$  be the result of calling `ToObject` passing the `this` value as the argument.
2. ReturnIfAbrupt( $O$ ).
3. If  $O$  is the result of calling the `Object` constructor with a host object (15.2.2.1), then
  - a. Return either  $O$  or another value such as the host object originally passed to the constructor. The specific result that is returned is implementation-defined.
4. Return  $O$ .

#### 15.2.4.5 Object.prototype.hasOwnProperty (V)

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

<b>Deleted:</b> one of the internal properties
<b>Deleted:</b> listed in Table 23
<b>Deleted:</b> <code>t</code>
<b>Deleted:</b> <code>Let</code>
<b>Deleted:</b> <code>class</code>
<b>Deleted:</b> of the [[Class]] internal property of $O$
<b>Deleted:</b> <code>class</code>
<b>Deleted:</b> Internal Property
<b>Deleted:</b> [[IsFunction]]
<b>Deleted:</b> [[Is]]
<b>Deleted:</b> []
<b>Deleted:</b> Function
<b>Deleted:</b> <code>E</code>
<b>Deleted:</b> [[Is]]
<b>Deleted:</b> []
<b>Deleted:</b> <code>RegExp</code>
<b>Deleted:</b> [[Is]]
<b>Deleted:</b> []
<b>Deleted:</b> [[Is]]
<b>Deleted:</b> []
<b>Deleted:</b> [[Is]]
<b>Deleted:</b> result of calling <code>ToObject</code> passing the
<b>Deleted:</b> as the argument
<b>Deleted:</b> <#> Let <code>toString</code> be the result of calling the [[Get internal method of $O$ passing " <code>toString</code> " as the argument.]<#> If <code>IsCallable(toString)</code> is false, throw a <code>TypeError</code> exception.]<#>
<b>Deleted:</b> calling
<b>Deleted:</b> the
<b>Deleted:</b> [[Call]] internal method of
<b>Deleted:</b> <code>toString</code> passing
<b>Deleted:</b> as the <code>this</code> value
<b>Deleted:</b> no

1. Let  $P$  be  $\text{ToString}(V)$ .
2. **ReturnIfAbrupt( $P$ )**.
3. Let  $O$  be the result of calling `ToObject` passing the `this` value as the argument.
4. **ReturnIfAbrupt( $O$ )**.
5. Let  $desc$  be the result of calling the `[[GetOwnProperty]]` internal method of  $O$  passing  $P$  as the argument.
6. If  $desc$  is `undefined`, return `false`.
7. Return `true`.

NOTE 1 Unlike `[[HasProperty]]` (8.12.6), this method does not consider objects in the prototype chain.

NOTE 2 The ordering of steps 1 and 2 is chosen to ensure that any exception that would have been thrown by step 1 in previous editions of this specification will continue to be thrown even if the `this` value is `undefined` or `null`.

#### 15.2.4.6 `Object.prototype.isPrototypeOf(V)`

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

1. If  $V$  is not an object, return `false`.
2. Let  $O$  be the result of calling `ToObject` passing the `this` value as the argument.
3. **ReturnIfAbrupt( $O$ )**.
4. Repeat
  - a. Let  $V$  be the value of the `[[Prototype]]` internal property of  $V$ .
  - b. If  $V$  is `null`, return `false`.
  - c. If  $O$  and  $V$  refer to the same object, return `true`.

NOTE The ordering of steps 1 and 2 is chosen to preserve the behaviour specified by previous editions of this specification for the case where  $V$  is not an object and the `this` value is `undefined` or `null`.

#### 15.2.4.7 `Object.prototype.propertyIsEnumerable(V)`

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

1. Let  $P$  be  $\text{ToString}(V)$ .
2. **ReturnIfAbrupt( $P$ )**.
3. Let  $O$  be the result of calling `ToObject` passing the `this` value as the argument.
4. **ReturnIfAbrupt( $O$ )**.
5. Let  $desc$  be the result of calling the `[[GetOwnProperty]]` internal method of  $O$  passing  $P$  as the argument.
6. If  $desc$  is `undefined`, return `false`.
7. Return the value of  $desc.[[Enumerable]]$ .

NOTE 1 This method does not consider objects in the prototype chain.

NOTE 2 The ordering of steps 1 and 2 is chosen to ensure that any exception that would have been thrown by step 1 in previous editions of this specification will continue to be thrown even if the `this` value is `undefined` or `null`.

#### 15.2.5 Properties of Object Instances

Object instances have no special properties beyond those inherited from the Object prototype object.

### 15.3 Function Objects

#### 15.3.1 The Function Constructor Called as a Function

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

### 15.3.1.1 Function (p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>n</sub>, body)

When the **Function** function is called with some arguments *p*<sub>1</sub>, *p*<sub>2</sub>, ..., *p*<sub>*n*</sub>, *body* (where *n* might be 0, that is, there are no “*p*” arguments, and where *body* might also not be provided), the following steps are taken:

1. Create and return a new Function object as if the standard built-in constructor Function was used in a **new** expression with the same arguments (15.3.2.1).

### 15.3.2 The Function Constructor

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

#### 15.3.2.1 new Function (p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>n</sub>, body)

The last argument specifies the body (executable code) of a function; any preceding arguments specify formal parameters.

When the **Function** constructor is called with some arguments *p*<sub>1</sub>, *p*<sub>2</sub>, ..., *p*<sub>*n*</sub>, *body* (where *n* might be 0, that is, there are no “*p*” arguments, and where *body* might also not be provided), the following steps are taken:

1. Let *argCount* be the total number of arguments passed to this function invocation.
2. Let *P* be the empty String.
3. If *argCount* = 0, let *bodyText* be the empty String.
4. Else if *argCount* = 1, let *bodyText* be that argument.
5. Else, *argCount* > 1
  - a. Let *firstArg* be the first argument.
  - b. Let *P* be *ToString(firstArg)*.
  - c. **ReturnIfAbrupt(P)**.
  - d. Let *k* be 2.
  - e. Repeat, while *k* < *argCount*
    - i. Let *nextArg* be the *k*'th argument.
    - ii. Let *nextArgString* be *ToString(nextArg)*.
    - iii. **ReturnIfAbrupt(nextArgString)**.
    - iv. Let *P* be the result of concatenating the previous value of *P*, the String " , " (a comma), and *nextArgString*.
    - v. Increase *k* by 1.
  - f. Let *bodyText* be the *k*'th argument.
6. Let *bodyText* be *ToString(bodyText)*.
7. **ReturnIfAbrupt(bodyText)**.
8. **Let parameters be the result of parsing P using FormalParameterList as the goal symbol. Throw a SyntaxError exception if the parse fails.**
9. **Let body be the result of parsing bodyText using FunctionBody as the goal symbol. Throw a SyntaxError exception if the parse fails or if any static semantics errors are detected.**
10. If *bodyText* is strict mode code (see 10.1.1) then let *strict* be **true**, else let *strict* be **false**.
11. Return a new Function object created as specified in 13.2 passing *parameters* as the *FormalParameterList*, and *body* as the *FunctionBody*. Pass in the Global Environment as the *Scope* parameter and *strict* as the *Strict* flag.

A **prototype** property is automatically created for every function, to provide for the possibility that the function will be used as a constructor.

**NOTE** It is permissible but not necessary to have one argument for each formal parameter to be specified. For example, all three of the following expressions produce the same result:

```
new Function("a", "b", "c", "return a+b+c")
new Function("a, b, c", "return a+b+c")
new Function("a,b", "c", "return a+b+c")
```

**Deleted:** *ToString*  
**Deleted:** *)*

**Deleted:** *If*  
**Deleted:** is not parsable as a

**Deleted:** *opt*  
**Deleted:** then *t*

**Comment [AWB 387]:** TODO: need to figure out how we decide if these need to be parsed as extended code

**Deleted:** <#> If *body* is not parsable as *FunctionBody* then throw a *SyntaxError* exception. ¶

**Deleted:** <#> If *strict* is **true**, throw any exceptions specified in 13.1 that apply. ¶

**Deleted:** *P*  
**Deleted:** *opt*

### 15.3.3 Properties of the Function Constructor

The Function constructor is itself a Function object and has a [[NativeBrand]] internal property whose value is NativeFunction. The value of the [[Prototype]] internal property of the Function constructor is the standard built-in Function prototype object (15.3.4).

The value of the [[Extensible]] internal property of the Function constructor is **true**.

The Function constructor has the following properties:

#### 15.3.3.1 Function.prototype

The initial value of `Function.prototype` is the standard built-in Function prototype object (15.3.4).

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

#### 15.3.3.2 Function.length

This is a data property with a value of 1. This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

### 15.3.4 Properties of the Function Prototype Object

The Function prototype object is itself a Function object and has a [[NativeBrand]] internal property whose value is NativeFunction. When invoked, it accepts any arguments and returns **undefined**.

The value of the [[Prototype]] internal property of the Function prototype object is the standard built-in Object prototype object (15.2.4). The initial value of the [[Extensible]] internal property of the Function prototype object is **true**.

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

The `length` property of the Function prototype object is **0**.

#### 15.3.4.1 Function.prototype.constructor

The initial value of `Function.prototype.constructor` is the built-in `Function` constructor.

#### 15.3.4.2 Function.prototype.toString()

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

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

#### 15.3.4.3 Function.prototype.apply (thisArg, argArray)

When the `apply` method is called on an object `func` with arguments `thisArg` and `argArray`, the following steps are taken:

1. If `IsCallable(func)` is **false**, then throw a `TypeError` exception.
2. If `argArray` is `null` or `undefined`, then
  - a. Return the result of calling the [[Call]] internal method of `func`, providing `thisArg` as the `this` value and an empty list of arguments.
3. If `Type(argArray)` is not `Object`, then throw a `TypeError` exception.

**Deleted:** its  
**Deleted:** Class  
**Deleted:** IsFunction  
**Deleted:** is "Function"

**Deleted:** IsFunction  
**Deleted:** (its [[Class]] is "Function") that,  
**Deleted:** when

4. Let *len* be the result of calling the [[Get]] internal method of *argArray* with argument "length".
5. Let *n* be ToUint32(*len*).
- 6. ReturnIfAbrupt(*n*).**
7. Let *argList* be an empty List.
8. Let *index* be 0.
9. Repeat while *index* < *n*
  - a. Let *indexName* be ToString(*index*).
  - b. Let *nextArg* be the result of calling the [[Get]] internal method of *argArray* with *indexName* as the argument.
  - c. ReturnIfAbrupt(*nextArg*).**
  - d. Append *nextArg* as the last element of *argList*.
  - e. Set *index* to *index* + 1.
10. Return the result of calling the [[Call]] internal method of *func*, providing *thisArg* as the this value and *argList* as the list of arguments.

The length property of the apply method is 2.

**NOTE** The thisArg value is passed without modification as the this value. This is a change from Edition 3, where a undefined or null thisArg is replaced with the global object and ToObject is applied to all other values and that result is passed as the this value.

#### 15.3.4.4 Function.prototype.call (thisArg [ , arg1 [ , arg2, ... ]])

When the call method is called on an object *func* with argument *thisArg* and optional arguments *arg1*, *arg2* etc, the following steps are taken:

1. If IsCallable(*func*) is false, then throw a **TypeError** exception.
2. Let *argList* be an empty List.
3. If this method was called with more than one argument then in left to right order starting with *arg1* append each argument as the last element of *argList*
4. Return the result of calling the [[Call]] internal method of *func*, providing *thisArg* as the this value and *argList* as the list of arguments.

The length property of the call method is 1.

**NOTE** The thisArg value is passed without modification as the this value. This is a change from Edition 3, where a undefined or null thisArg is replaced with the global object and ToObject is applied to all other values and that result is passed as the this value.

#### 15.3.4.5 Function.prototype.bind (thisArg [ , arg1 [ , arg2, ...]])

The bind method takes one or more arguments, *thisArg* and (optionally) *arg1*, *arg2*, etc, and returns a new function object by performing the following steps:

1. Let *Target* be the this value.
2. If IsCallable(*Target*) is false, throw a **TypeError** exception.
3. Let *A* be a new (possibly empty) internal list of all of the argument values provided after *thisArg* (*arg1*, *arg2* etc), in order.
4. Let *F* be a new native ECMAScript object .
5. Set all the internal methods of *F* as specified in 8.12.
- 6. Set the [[Extensible]] internal property of *F* to true.**
7. Set the [[TargetFunction]] internal property of *F* to *Target*.
8. Set the [[BoundThis]] internal property of *F* to the value of *thisArg*.
9. Set the [[BoundArgs]] internal property of *F* to *A*.
10. **Add the [[NativeBrand]] internal property with value NativeFunction to *F*.**
11. Set the [[Prototype]] internal property of *F* to the intrinsic %FunctionPrototype%.
12. Set the [[Call]] internal property of *F* as described in 15.3.4.5.1.
13. Set the [[Construct]] internal property of *F* as described in 15.3.4.5.2.
14. Set the [[HasInstance]] internal property of *F* as described in 15.3.4.5.3.
15. If *Target* has the [[NativeBrand]] internal property with value NativeFunction, then

<b>Deleted:</b> , except for [[Get]],
<b>Deleted:</b> <#>Set the [[Get]] internal property of <i>F</i> as specified in 15.3.5.4.¶
<b>Deleted:</b> Set
<b>Deleted:</b> Class
<b>Deleted:</b> IsFunction
<b>Deleted:</b> of
<b>Deleted:</b> to
<b>Deleted:</b> "Function"
<b>Deleted:</b> standard built-in Function prototype object as specified in 15.3.3.1
<b>Deleted:</b> the [[Class]] internal property of
<b>Deleted:</b> is "Function"
<b>Deleted:</b> IsFunction

- a. Let `targetLen` be the result of calling the `[[Get]]` internal method of `Target` with argument `"length"`.
- b. `ReturnIfAbrupt(targetLen)`.
- c. Let `L` be the larger of `0` and the result of `targetLen`, minus the `number of element` of `A`.
16. Else let `L` be `0`.
17. Call the `[[DefineOwnProperty]]` internal method of `F` with arguments `"length"`, `PropertyDescriptor` `{[[value]]: L, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}`, and `false`.
18. Let `thrower` be the `[[ThrowTypeError]]` function Object (13.2.3).
19. Call the `[[DefineOwnProperty]]` internal method of `F` with arguments `"caller"`, `PropertyDescriptor` `{[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: false}`, and `false`.
20. Call the `[[DefineOwnProperty]]` internal method of `F` with arguments `"arguments"`, `PropertyDescriptor` `{[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: false}`, and `false`.
21. Return `F`.

The `length` property of the `bind` method is `1`.

**NOTE** Function objects created using `Function.prototype.bind` do not have a `prototype` property or the `[[Code]]`, `[[FormalParameters]]`, and `[[Scope]]` internal properties.

#### 15.3.4.5.1 `[[Call]]`

When the `[[Call]]` internal method of a function object, `F`, which was created using the `bind` function is called with a `this` value and a list of arguments `ExtraArgs`, the following steps are taken:

1. Let `boundArgs` be the value of `F`'s `[[BoundArgs]]` internal property.
2. Let `boundThis` be the value of `F`'s `[[BoundThis]]` internal property.
3. Let `target` be the value of `F`'s `[[TargetFunction]]` internal property.
4. Let `args` be a new list containing the same values as the list `boundArgs` in the same order followed by the same values as the list `ExtraArgs` in the same order.
5. Return the result of calling the `[[Call]]` internal method of `target` providing `boundThis` as the `this` value and providing `args` as the arguments.

#### 15.3.4.5.2 `[[Construct]]`

When the `[[Construct]]` internal method of a function object, `F` that was created using the `bind` function is called with a list of arguments `ExtraArgs`, the following steps are taken:

1. Let `target` be the value of `F`'s `[[TargetFunction]]` internal property.
2. If `target` has no `[[Construct]]` internal method, a **TypeError** exception is thrown.
3. Let `boundArgs` be the value of `F`'s `[[BoundArgs]]` internal property.
4. Let `args` be a new list containing the same values as the list `boundArgs` in the same order followed by the same values as the list `ExtraArgs` in the same order.
5. Return the result of calling the `[[Construct]]` internal method of `target` providing `args` as the arguments.

#### 15.3.4.5.3 `[[HasInstance]]` (V)

When the `[[HasInstance]]` internal method of a function object `F`, that was created using the `bind` function is called with argument `V`, the following steps are taken:

1. Let `target` be the value of `F`'s `[[TargetFunction]]` internal property.
2. If `target` has no `[[HasInstance]]` internal method, a **TypeError** exception is thrown.
3. Return the result of calling the `[[HasInstance]]` internal method of `target` providing `V` as the argument.

#### 15.3.5 Properties of Function Instances

In addition to the required internal properties, every function instance has a `[[Call]]` internal property and in most cases uses a different version of the `[[Get]]` internal property. Depending on how they are created (see 8.6.2, 13.2, 15, and 15.3.4.5), function instances may have a `[[HasInstance]]` internal property, a `[[Scope]]` internal property, a `[[Construct]]` internal property, a `[[FormalParameters]]` internal property, a `[[Code]]` internal property, a `[[TargetFunction]]` internal property, a `[[BoundThis]]` internal property, and a `[[BoundArgs]]` internal property.

<b>Deleted:</b> the <code>length</code> property of <code>Target</code>
<b>Deleted:</b> <code>length</code>
<b>Deleted:</b> <#> Set the <code>length</code> own property of <code>F</code> either <code>0</code> or <code>L</code> , whichever is larger. ¶
<b>Deleted:</b> <code>set</code>
<b>Deleted:</b> the <code>length</code> own property of <code>F</code>
<b>Deleted:</b> <#> Set the attributes of the <code>length</code> own property of <code>F</code> to the values specified in 15.3.5.1.¶
<b>Deleted:</b> <#> Set the <code>[[Extensible]]</code> internal property of <code>F</code> to <code>true</code> .¶

## Every function instance has a `[[NativeBrand]]` internal property whose value is `NativeFunction`.

Function instances that correspond to strict mode functions (13.2) and function instances created using the `Function.prototype.bind` method (15.3.4.5) have properties named "caller" and "arguments" that throw a `TypeError` exception. An ECMAScript implementation must not associate any implementation specific behaviour with accesses of these properties from strict mode function code.

**Deleted:** The value of the `[[Class]]` internal property is "Function".

**Deleted:** `IsFunction`

### 15.3.5.1 `length`

The value of the `length` property is an integer that indicates the "typical" number of arguments expected by the function. However, the language permits the function to be invoked with some other number of arguments. The behaviour of a function when invoked on a number of arguments other than the number specified by its `length` property depends on the function. This property has the attributes { `[[Writable]]: false`, `[[Enumerable]]: false`, `[[Configurable]]: false` }.

### 15.3.5.2 `prototype`

The value of the `prototype` property is used to initialise the `[[Prototype]]` internal property of a newly created object before the `Function` object is invoked as a constructor for that newly created object. This property has the attribute { `[[Writable]]: true`, `[[Enumerable]]: false`, `[[Configurable]]: false` }.

**NOTE** Function objects created using `Function.prototype.bind` do not have a `prototype` property.

### 15.3.5.3 `[[HasInstance]] (V)`

Assume  $F$  is a `Function` object.

When the `[[HasInstance]]` internal method of  $F$  is called with value  $V$ , the following steps are taken:

1. If  $V$  is not an object, return `false`.
2. Let  $O$  be the result of calling the `[[Get]]` internal method of  $F$  with property name "`prototype`".
3. `ReturnIfAbrupt(O)`.
4. If `Type(O)` is not `Object`, throw a `TypeError` exception.
5. Repeat
  - a. Let  $V$  be the value of the `[[Prototype]]` internal property of  $V$ .
  - b. If  $V$  is `null`, return `false`.
  - c. If  $O$  and  $V$  refer to the same object, return `true`.

**NOTE** Function objects created using `Function.prototype.bind` have a different implementation of `[[HasInstance]]` defined in 15.3.4.5.3.

### 15.3.5.4 `[[Get]] (P)`

Function objects use a variation of the `[[Get]]` internal method used for other native ECMAScript objects (8.12.3).

Assume  $F$  is a `Function` object. When the `[[Get]]` internal method of  $F$  is called with property name  $P$ , the following steps are taken:

1. Let  $v$  be the result of calling the default `[[Get]]` internal method (8.12.3) on  $F$  passing  $P$  as the property name argument.
2. If  $P$  is "`caller`" and  $v$  is a strict mode `Function` object, throw a `TypeError` exception.
3. Return  $v$ .

**NOTE** Function objects created using `Function.prototype.bind` use the default `[[Get]]` internal method.

## 15.4 Array Objects

Array objects give special treatment to a certain class of property names. A property name  $P$  (in the form of a String value) is an *array index* if and only if  $\text{ToString}(\text{ToUint32}(P))$  is equal to  $P$  and  $\text{ToUint32}(P)$  is not equal to  $2^{32}-1$ . A property whose property name is an array index is also called an *element*. Every Array object has a **length** property whose value is always a nonnegative integer less than  $2^{32}$ . The value of the **length** property is numerically greater than the name of every property whose name is an array index; whenever a property of an Array object is created or changed, other properties are adjusted as necessary to maintain this invariant. Specifically, whenever a property is added whose name is an array index, the **length** property is changed, if necessary, to be one more than the numeric value of that array index; and whenever the **length** property is changed, every property whose name is an array index whose value is not smaller than the new length is automatically deleted. This constraint applies only to own properties of an Array object and is unaffected by **length** or array index properties that may be inherited from its prototypes.

An object,  $O$ , is said to be *sparse* if the following algorithm returns **true**:

1. Let  $len$  be the result of calling the **[[Get]]** internal method of  $O$  with argument **"length"**.
2. For each integer  $i$  in the range  $0 \leq i < \text{ToUint32}(len)$ 
  - a. Let  $elem$  be the result of calling the **[[GetOwnProperty]]** internal method of  $O$  with argument  $\text{ToString}(i)$ .
  - b. If  $elem$  is **undefined**, return **true**.
3. Return **false**.

The abstract operation **ArrayCreate** with argument **length** (a positive integer) is used to specify the creation of new Array objects. It performs the following steps:

1. Let  $A$  be a newly created ECMAScript object.
2. Set the **[[Prototype]]** internal property of  $A$  to the intrinsic object **%ArrayPrototype%**.
3. Set  $A$ 's common internal methods except for **[[DefineOwnProperty]]** to the default definitions specified in 8.12.
4. Set  $A$ 's **[[DefineOwnProperty]]** internal methods to the definition given in 15.4.5.1.
5. Set the **[[NativeBrand]]** internal property of  $A$  to the value **NativeArray**.
6. Set the **[[Extensible]]** internal property of  $A$  to **true**.
7. Call the **[[DefineOwnProperty]]** internal method of  $A$  with arguments **"length"**, **Property Descriptor** **{[[Value]]:  $length$ , [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false**}**, and **false**.
8. Return  $A$ .

### 15.4.1 The Array Constructor Called as a Function

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

**Formatted:** Note

#### 15.4.1.1 **Array ([ item1 [, item2 [, ... ]]])**

When the **Array** function is called the following steps are taken:

1. Return the result that would be obtained if this functions had been called with the same arguments, as constructor. This result is defined by 15.4.2.1 or 15.4.2.2 depending upon the actual number of arguments.

**Deleted:** Create and return a new Array object exactly as if the standard built-in constructor **Arr** was used in a **new** expression with the same arguments (15.4.2).

### 15.4.2 The Array Constructor

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

**Formatted:** Note

#### 15.4.2.1 **new Array ([ item0 [, item1 [, ... ]]])**

This description applies if and only if the Array constructor is given no arguments or at least two arguments.

1. Let  $len$  be the number of arguments passed to this constructor call.
2. Let  $array$  be the result of the abstraction operation **ArrayCreate** (15.4) with argument  $len$ .

3. ReturnIfAbrupt(array).
4. Let  $k$  be 0.
5. Let  $items$  be a zero-origin List contain the argument items in order.
6. Repeat, while  $k < len$ 
  - a. Let  $P_k$  be `ToString(k)`.
  - b. Let  $itemK$  be  $k^{\text{th}}$  element of  $items$ .
  - c. Let  $defineStatus$  be the result of calling the `[[DefineOwnProperty]]` internal method of  $array$  with arguments  $P_k$ , Property Descriptor `{[[Value]]: itemK, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, and  $true$ .
  - d. ReturnIfAbrupt(defineStatus).
  - e. Increase  $k$  by 1.
7. Let  $putStatus$  be the result of calling the `[[Put]]` internal method of  $array$  with arguments `"length"`,  $len$ , and  $true$ .
8. ReturnIfAbrupt(putStatus).
9. Return  $array$ .

#### 15.4.2.2 `new Array (len)`

This description applies if and only if the `Array` constructor is given exactly one argument.

1. If `Type(len)` is not `Number`, then
  - a. Let  $array$  be the result of the abstraction operation `ArrayCreate (15.4)` with argument 1.
  - b. ReturnIfAbrupt(array).
  - c. Let  $defineStatus$  be the result of calling the `[[DefineOwnProperty]]` internal method of  $array$  with arguments `"0"`, Property Descriptor `{[[Value]]: len, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, and  $true$ .
  - d. ReturnIfAbrupt(defineStatus).
  - e. Return array.
2. Let  $intLen$  be `ToUint32(len)`.
3. If  $intLen \neq len$ , then throw a `RangeError` exception.
4. Let  $array$  be the result of the abstraction operation `ArrayCreate (15.4)` with argument  $intLen$ .
5. Return array.

#### 15.4.3 Properties of the `Array` Constructor

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

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

##### 15.4.3.1 `Array.prototype`

The initial value of `Array.prototype` is the `Array` prototype object (15.4.4).

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

##### 15.4.3.2 `Array.isArray ( arg )`

The `isArray` function takes one argument  $arg$ , and returns the Boolean value `true` if the argument is an object whose class internal property is `"Array"`; otherwise it returns `false`. The following steps are taken:

1. If `Type(arg)` is not `Object`, return `false`.
2. If  $arg$  has the `[[NativeBrand]]` internal property with value `NativeArray`, then return `true`.
3. Return `false`.

##### 15.4.3.3 `Array.of ( ...items )`

When the `of` method is called with any number of arguments, the following steps are taken:

**Deleted:** The `[[Prototype]]` internal property of the newly constructed object is set to the original `Array` prototype object, the one that is the initial value of `Array.prototype` (15.4.3.1).  
 The `[[Class]]` internal property of the newly constructed object is set to `"Array"` has the `[[IsArrayNativeBrand]]` internal property with value `NativeArray`.  
 The `[[Extensible]]` internal property of the newly constructed object is set to `true`.  
 The `length` property of the newly constructed object is set to the number of arguments.  
 The `o` property of the newly constructed object is set to `item0` (if supplied); the `1` property of the newly constructed object is set to `item1` (if supplied); and, in general, for all  $k$  properties of the newly constructed object is set to argument  $k$ , where the first argument is considered to be argument number 0. The properties all have the attributes `{ [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true }`.

**Deleted:** The `[[Prototype]]` internal property of the newly constructed object is set to the original `Array` prototype object, the one that is the initial value of `Array.prototype` (15.4.3.1). The newly constructed object has the `[[IsArrayNativeBrand]]` internal property with value `NativeArray`. The `[[Class]]` internal property of the newly constructed object is set to `"Array"`. The `[[Extensible]]` internal property of the newly constructed object is set to `true`.  
 If the argument  $len$  is a Number and `ToUint32(len)` is equal to  $len$ , then the `length` property of the newly constructed object is set to `ToUint32(len)`. If the argument  $len$  is a Number and `ToUint32(len)` is not equal to  $len$ , a `RangeError` exception is thrown.  
 If the argument  $len$  is not a Number, then the `length` property of the newly constructed object is set to 1 and the `o` property of the newly constructed object is set to with attributes `{ [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true }`.

**Deleted:** the value of the `[[Class]]` internal property of

**Deleted:** is `"Array"`

**Deleted:** `isArray`

1. Let *lenValue* be the result of calling the **[[Get]]** internal method of *items* with the argument **"length"**.
2. Let *len* be **ToInteger(lenValue)**.
3. Let *C* be the **this** value.
4. If **isConstructor(C)** is **true**, then
  - a. Let *newObj* be the result of calling the **[[Construct]]** internal method of *C* with an argument list containing the single item *len*.
  - b. Let *A* be **ToObject(newObj)**.
5. Else,
  - a. Let *A* be the result of the abstraction operation **ArrayCreate (15.4)** with argument *len*.
6. **ReturnIfAbrupt(A)**.
7. Let *k* be **0**.
8. Repeat, while *k* < *len*
  - a. Let *Pk* be **ToString(k)**.
  - b. Let *kValue* be the result of calling the **[[Get]]** internal method of *items* with argument *Pk*.
  - c. Let *defineStatus* be the result of calling the **[[DefineOwnProperty]]** internal method of *A* with arguments *Pk*, **Property Descriptor** **{[[Value]]: kValue, [[value]], [[Writable]], true, [[Enumerable]], true, [[Configurable]], true}**, and **true**.
  - d. **ReturnIfAbrupt(defineStatus)**.
  - e. Increase *k* by **1**.
9. Let *putStatus* be the result of calling the **[[Put]]** internal method of *A* with arguments **"length"**, *len*, and **true**.
10. **ReturnIfAbrupt(putStatus)**.
11. Return *A*.

The **length** property of the **of** method is **0**.

**NOTE 1** The *items* argument is assumed to be a well-formed rest argument value.

**NOTE 2** The **of** function is an intentionally generic factory method; it does not require that its **this** value be the Array constructor. Therefore it can be transferred to or inherited by other constructors that may be called with a single numeric argument.

#### 15.4.3.3 **Array.from (arrayLike)**

When the **from** method is called with any number of arguments, the following steps are taken:

1. Let *items* be **ToObject(arrayLike)**.
2. **ReturnIfAbrupt(items)**.
3. Let *lenValue* be the result of calling the **[[Get]]** internal method of *items* with the argument **"length"**.
4. Let *len* be **ToInteger(lenValue)**.
5. **ReturnIfAbrupt(len)**.
6. Let *C* be the **this** value.
7. If **isConstructor(C)** is **true**, then
  - a. Let *newObj* be the result of calling the **[[Construct]]** internal method of *C* with an argument list containing the single item *len*.
  - b. Let *A* be **ToObject(newObj)**.
8. Else,
  - a. Let *A* be the result of the abstraction operation **ArrayCreate (15.4)** with argument *len*.
9. **ReturnIfAbrupt(A)**.
10. Let *k* be **0**.
11. Repeat, while *k* < *len*
  - a. Let *Pk* be **ToString(k)**.
  - b. Let *kPresent* be the result of calling the **[[HasProperty]]** internal method of *items* with argument *Pk*.
  - c. If *kPresent* is **true**, then
    - i. Let *kValue* be the result of calling the **[[Get]]** internal method of *items* with argument *Pk*.
    - ii. **ReturnIfAbrupt(kValue)**.
    - iii. Let *defineStatus* be the result of calling the **[[DefineOwnProperty]]** internal method of *A* with arguments *Pk*, **Property Descriptor** **{[[Value]]: kValue, [[value]], [[Writable]], true, [[Enumerable]], true, [[Configurable]], true}**, and **true**.
    - iv. **ReturnIfAbrupt(defineStatus)**.

**Comment [AWB788]:** It would be nice to have a more explicit method to invoke to create a collection with a pre-specified number of elements.

**Comment [AWB789]:** It would be nice to have a more explicit method to invoke to create a collection with a pre-specified number of elements.

d. Increase  $k$  by 1.

12. Let  $putStatus$  be the result of calling the  $[[Put]]$  internal method of  $A$  with arguments  $"length"$ ,  $len$ , and  $true$ .

13. ReturnIfAbrupt( $putStatus$ ).

14. Return  $A$ .

**NOTE** The `from` function is an intentionally generic factory method; it does not require that its `this` value be the `Array` constructor. Therefore it can be transferred to or herited by any other constructors that may be called with a single numeric argument.

#### 15.4.4 Properties of the Array 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 `Array` prototype object is itself an array; it has an  $[[NativeBrand]]$  internal property with value `NativeArray`, and it has a `length` property (whose initial value is `+0`) and the special  $[[DefineOwnProperty]]$  internal method described in 15.4.5.1.

In following descriptions of functions that are properties of the `Array` prototype object, the phrase “this object” refers to the object that is the `this` value for the invocation of the function. It is permitted for the `this` to be an object which does not have an  $[[NativeBrand]]$  internal property with value `NativeArray`.

**NOTE** The `Array` prototype object does not have a `valueOf` property of its own; however, it inherits the `valueOf` property from the standard built-in `Object` prototype Object.

##### 15.4.4.1 `Array.prototype.constructor`

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

##### 15.4.4.2 `Array.prototype.toString()`

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

6. Let  $array$  be the result of calling `ToObject` on the `this` value.

7. ReturnIfAbrupt( $array$ ).

8. Let  $func$  be the result of calling the  $[[Get]]$  internal method of  $array$  with argument `"join"`.

9. ReturnIfAbrupt( $func$ ).

10. If `IsCallable(func)` is `false`, then let  $func$  be the standard built-in method `Object.prototype.toString` (15.2.4.2).  
 11. Return the result of calling the  $[[Call]]$  internal method of  $func$  providing  $array$  as the `this` value and an empty arguments list.

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

##### 15.4.4.3 `Array.prototype.toLocaleString()`

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

The result is calculated as follows:

1. Let  $array$  be the result of calling `ToObject` passing the `this` value as the argument.

2. ReturnIfAbrupt( $array$ ).

3. Let  $arrayLen$  be the result of calling the  $[[Get]]$  internal method of  $array$  with argument `"length"`.

4. Let  $len$  be `ToInt32(arrayLen)`.

5. ReturnIfAbrupt( $len$ ).

**Deleted:** its  $[[Class]]$  is "Array"

**Deleted:** `isArray`

**Deleted:** `for`

**Deleted:** the value of the  $[[Class]]$  internal property is "Array"

**Deleted:** `isArray`

**Deleted:** If  $array$  is an abrupt completion, return  $array$ .

**Deleted:** If  $func$  is an abrupt completion, return  $func$ .

6. Let *separator* be the String value for the list-separator String appropriate for the host environment's current locale (this is derived in an implementation-defined way).
7. If *len* is zero, return the empty String.
8. Let *firstElement* be the result of calling the [[Get]] internal method of *array* with argument "0".
9. ReturnIfAbrupt(*firstElement*).
10. If *firstElement* is **undefined** or **null**, then
  - a. Let *R* be the empty String.
11. Else
  - a. Let *R* be the result of calling Invoke with arguments "toLocaleString", *firstElement*, and an empty arguments List.
  - b. Let *R* be ToString(*R*).
  - c. ReturnIfAbrupt(*R*).
12. Let *k* be 1.
13. Repeat, while *k* < *len*
  - a. Let *S* be a String value produced by concatenating *R* and *separator*.
  - b. Let *nextElement* be the result of calling the [[Get]] internal method of *array* with argument ToString(*k*).
  - c. ReturnIfAbrupt(*nextElement*).
  - d. If *nextElement* is **undefined** or **null**, then
    - i. Let *R* be the empty String.
  - e. Else
    - i. Let *R* be the result of calling Invoke with arguments "toLocaleString", *nextElement*, and an empty arguments List.
    - ii. Let *R* be ToString(*R*).
    - iii. ReturnIfAbrupt(*R*).
  - f. Let *R* be a String value produced by concatenating *S* and *R*.
  - g. Increase *k* by 1.
14. Return *R*.

NOTE 1 The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

NOTE 2 The `toLocaleString` function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `toLocaleString` function can be applied successfully to a host object is implementation-dependent.

#### 15.4.4.4 Array.prototype.concat ( [ *item1* [ , *item2* [ , ... ] ] ] )

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

The following steps are taken:

1. Let *O* be the result of calling `ToObject` passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *A* be the result of the abstraction operation `ArrayCreate` (15.4) with argument 0.
4. Let *n* be 0.
5. Let *items* be an internal List whose first element is *O* and whose subsequent elements are, in left to right order, the arguments that were passed to this function invocation.
6. Repeat, while *items* is not empty
  - a. Remove the first element from *items* and let *E* be the value of the element.
  - b. If *E* has the `[[NativeBrand]]` internal property with value `NativeArray`, then
    - i. Let *k* be 0.
    - ii. Let *len* be the result of calling the [[Get]] internal method of *E* with argument "`length`".
    - iii. ReturnIfAbrupt(*len*).
    - iv. Repeat, while *k* < *len*
      1. Let *P* be `ToString`(*k*).
      2. Let *exists* be the result of calling the [[HasProperty]] internal method of *E* with *P*.
      3. If *exists* is **true**, then

**Deleted:** <#>Let *elementObj* be `ToObject`(*firstElement*).  
 <#>Let *func* be the result of calling the [[Get]] internal method of *elementObj* with argument "`toLocaleString`".  
 <#>If `IsCallable`(*func*) is **false**, throw a `TypeError` exception.  
**Comment [AWB790]:** This step was missing in ES<=5.1

**Deleted:** If *R* is an abrupt completion, return *R*.

**Deleted:** Let *R* be the result of calling the [[Call]] internal method of *func* providing *elementObj* as the **this** value and an empty arguments list.

**Deleted:** <#>Let *elementObj* be `ToObject`(*nextElement*).  
**Comment [AWB791]:** This step was missing in ES<=5.1

**Deleted:** If *R* is an abrupt completion, return *R*.

**Deleted:** <#>Let *func* be the result of calling the [[Get]] internal method of *elementObj* with argument "`toLocaleString`".  
 <#>If `IsCallable`(*func*) is **false**, throw a `TypeError` exception.  
 <#>Let *R* be the result of calling the [[Call]] internal method of *func* providing *elementObj* as the **this** value and an empty arguments list.

**Deleted:** a new array created as if by the expression `new Array()` where `Array` is the standard built-in constructor with that name

**Deleted:** the value of the [[Class]] internal property of

**Deleted:** is "Array"

**Deleted:** isArray

- a. Let *subElement* be the result of calling the [[Get]] internal method of *E* with argument *P*.
- b. Call the [[DefineOwnProperty]] internal method of *A* with arguments *ToString(n)*, Property Descriptor {[[Value]]: *subElement*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
4. Increase *n* by 1.
5. Increase *k* by 1.
- c. Else, *E* is not an Array
  - i. Call the [[DefineOwnProperty]] internal method of *A* with arguments *ToString(n)*, Property Descriptor {[[Value]]: *E*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
  - ii. Increase *n* by 1.

7. Let *putStatus* be the result of calling the [[Put]] internal method of *A* with arguments "**length**", *n*, and **true**.

8. ReturnIfAbrupt(*putStatus*).

9. Return *A*.

**Deleted:** Call

**Deleted:** If *putStatus* is an abrupt completion, return *putStatus*.

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

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

#### 15.4.4.5 Array.prototype.join (separator)

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

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

1. Let *O* be the result of calling **ToObject** passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *lenVal* be the result of calling the [[Get]] internal method of *O* with argument "**length**".
4. Let *len* be **ToUint32**(*lenVal*).
5. ReturnIfAbrupt(*len*).
6. If *separator* is **undefined**, let *separator* be the single-character String **,**.
7. Let *sep* be **ToString**(*separator*).
8. If *len* is zero, return the empty String.
9. Let *element0* be the result of calling the [[Get]] internal method of *O* with argument "**0**".
10. If *element0* is **undefined** or **null**, let *R* be the empty String; otherwise, Let *R* be **ToString**(*element0*).
11. If *R* is an abrupt completion, return *R*.
12. Let *k* be **1**.
13. Repeat, while *k* < *len*
  - a. Let *S* be the String value produced by concatenating *R* and *sep*.
  - b. Let *element* be the result of calling the [[Get]] internal method of *O* with argument **ToString**(*k*).
  - c. If *element* is **undefined** or **null**, Let *next* be the empty String; otherwise, let *next* be **ToString**(*element*).
  - d. ReturnIfAbrupt(*next*).
  - e. Let *R* be a String value produced by concatenating *S* and *next*.
  - f. Increase *k* by 1.
14. Return *R*.

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

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

**Deleted:** If *next* is an abrupt completion, return *next*.

#### 15.4.4.6 Array.prototype.pop ()

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

1. Let  $O$  be the result of calling `ToObject` passing the `this` value as the argument.
2. `ReturnIfAbrupt( $O$ )`
3. Let  $lenVal$  be the result of calling the `[[Get]]` internal method of  $O$  with argument `"length"`.
4. Let  $len$  be `ToInt32(lenVal)`.
5. `ReturnIfAbrupt(len)`
6. If  $len$  is zero,
  - a. Let  $putStatus$  be the result of calling the `[[Put]]` internal method of  $O$  with arguments `"length"`,  $0$ , and `true`.
  - b. ReturnIfAbrupt(putStatus)
  - c. Return `undefined`.
7. Else,  $len > 0$ 
  - a. Let  $newLen$  be  $len - 1$ .
  - b. Let  $indx$  be `ToString(newLen)`.
  - c. Let  $element$  be the result of calling the `[[Get]]` internal method of  $O$  with argument  $indx$ .
  - d. ReturnIfAbrupt(element)
  - e. Let  $deleteStatus$  be the result of calling the `[[Delete]]` internal method of  $O$  with arguments  $indx$  and `true`.
  - f. ReturnIfAbrupt(deleteStatus)
  - g. Let  $putStatus$  be the result of calling the `[[Put]]` internal method of  $O$  with arguments `"length"`,  $newLen$ , and `true`.
  - h. ReturnIfAbrupt(putStatus)
  - i. Return  $element$ .



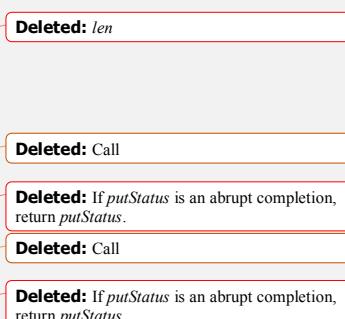
**NOTE** The `pop` function is intentionally generic; it does not require that its `this` value be an `Array` object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `pop` function can be applied successfully to a host object is implementation-dependent.

#### 15.4.4.7 Array.prototype.push ([ item1 [, item2 [, ... ]]])

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

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

1. Let  $O$  be the result of calling `ToObject` passing the `this` value as the argument.
2. `ReturnIfAbrupt( $O$ )`
3. Let  $lenVal$  be the result of calling the `[[Get]]` internal method of  $O$  with argument `"length"`.
4. Let  $n$  be `ToInt32(lenVal)`.
5. `ReturnIfAbrupt( $p$ )`
6. Let  $items$  be an internal List whose elements are, in left to right order, the arguments that were passed to this function invocation.
7. Repeat, while  $items$  is not empty
  - a. Remove the first element from  $items$  and let  $E$  be the value of the element.
  - b. Let  $putStatus$  be the result of calling the `[[Put]]` internal method of  $O$  with arguments `ToString(n)`,  $E$ , and `true`.
  - c. ReturnIfAbrupt(putStatus)
  - d. Increase  $n$  by 1.
8. Let  $putStatus$  be the result of calling the `[[Put]]` internal method of  $O$  with arguments `"length"`,  $n$ , and `true`.
9. `ReturnIfAbrupt(putStatus)`
10. Return  $n$ .



The `length` property of the `push` method is 1.

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

#### 15.4.4.8 `Array.prototype.reverse()`

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

1. Let  $O$  be the result of calling `ToObject` passing the `this` value as the argument.
2. `ReturnIfAbrupt( $O$ )`.
3. Let  $lenVal$  be the result of calling the `[[Get]` internal method of  $O$  with argument `"length"`.
4. Let  $len$  be `ToInt32(lenVal)`.
5. `ReturnIfAbrupt(len)`.
6. Let  $middle$  be  $\text{floor}(len/2)$ .
7. Let  $lower$  be `0`.
8. Repeat, while  $lower \neq middle$ 
  - a. Let  $upper$  be  $len - lower - 1$ .
  - b. Let  $upperP$  be `ToString(upper)`.
  - c. Let  $lowerP$  be `ToString(lower)`.
  - d. Let  $lowerValue$  be the result of calling the `[[Get]` internal method of  $O$  with argument  $lowerP$ .
  - e. `ReturnIfAbrupt(lowerValue)`.
  - f. Let  $upperValue$  be the result of calling the `[[Get]` internal method of  $O$  with argument  $upperP$ .
  - g. `ReturnIfAbrupt(upperValue)`.
  - h. Let  $lowerExists$  be the result of calling the `[[HasProperty]]` internal method of  $O$  with argument  $lowerP$ .
  - i. Let  $upperExists$  be the result of calling the `[[HasProperty]]` internal method of  $O$  with argument  $upperP$ .
  - j. If  $lowerExists$  is `true` and  $upperExists$  is `true`, then
    - i. `Let putStatus be the result of calling the [[Put]] internal method of O with arguments lowerP, upperValue, and true .`
    - ii. `ReturnIfAbrupt(putStatus)`.
    - iii. `Let putStatus be the result of calling the [[Put]] internal method of O with arguments upperP, lowerValue, and true .`
    - iv. `ReturnIfAbrupt(putStatus)`.
  - k. Else if  $lowerExists$  is `false` and  $upperExists$  is `true`, then
    - i. `Let putStatus be the result of calling the [[Put]] internal method of O with arguments lowerP, upperValue, and true .`
    - ii. `ReturnIfAbrupt(putStatus)`.
    - iii. `Let deleteStatus be the result of calling the [[Delete]] internal method of O, with arguments upperP and true .`
    - iv. `ReturnIfAbrupt(deleteStatus)`.
  - l. Else if  $lowerExists$  is `true` and  $upperExists$  is `false`, then
    - i. `Let deleteStatus be the result of calling the [[Delete]] internal method of O, with arguments lowerP and true .`
    - ii. `ReturnIfAbrupt(deleteStatus)`.
    - iii. `Let putStatus be the result of calling the [[Put]] internal method of O with arguments upperP, lowerValue, and true .`
    - iv. `ReturnIfAbrupt(putStatus)`.
  - m. Else, both  $lowerExists$  and  $upperExists$  are `false`
    - i. No action is required.
    - n. Increase  $lower$  by 1.
9. Return  $O$ .

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

**Deleted:** If  $lowerValue$  is an abrupt completion, return  $lowerValue$

**Deleted:** If  $upperValue$  is an abrupt completion, return  $upperValue$

**Deleted:** Call

**Deleted:** If  $putStatus$  is an abrupt completion, return  $putStatus$

**Deleted:** Call

**Deleted:** If  $putStatus$  is an abrupt completion, return  $putStatus$

**Deleted:** Call

**Deleted:** If  $putStatus$  is an abrupt completion, return  $putStatus$

**Deleted:** Call

**Deleted:** If  $putStatus$  is an abrupt completion, return  $putStatus$

**Deleted:** Call

**Deleted:** If  $deleteStatus$  is an abrupt completion, return  $deleteStatus$

**Deleted:** Call

**Deleted:** If  $deleteStatus$  is an abrupt completion, return  $deleteStatus$

**Deleted:** Call

**Deleted:** If  $putStatus$  is an abrupt completion, return  $putStatus$

#### 15.4.4.9 Array.prototype.shift()

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

1. Let  $O$  be the result of calling `ToObject` passing the `this` value as the argument.
2. [ReturnIfAbrupt\( \$O\$ \)](#)
3. Let  $lenVal$  be the result of calling the `[[Get]]` internal method of  $O$  with argument `"length"`.
4. Let  $len$  be `ToInt32(lenVal)`.
5. [ReturnIfAbrupt\(len\)](#)
6. If  $len$  is zero, then
  - a. [Let putStatus be the result of calling the \[\[Put\]\] internal method of  \$O\$  with arguments "length", 0, and true.](#)
  - b. [ReturnIfAbrupt\(putStatus\)](#)
  - c. Return `undefined`.
7. Let  $first$  be the result of calling the `[[Get]]` internal method of  $O$  with argument `"0"`.
8. [ReturnIfAbrupt\(first\)](#)
9. Let  $k$  be 1.
10. Repeat, while  $k < len$ 
  - a. Let  $from$  be `ToString(k)`.
  - b. Let  $to$  be `ToString( $k-1$ )`.
  - c. Let  $fromPresent$  be the result of calling the `[[HasProperty]]` internal method of  $O$  with argument  $from$ .
  - d. If  $fromPresent$  is `true`, then
    - i. Let  $fromVal$  be the result of calling the `[[Get]]` internal method of  $O$  with argument  $from$ .
    - ii. [ReturnIfAbrupt\(fromVal\)](#)
    - iii. [Let putStatus be the result of calling the \[\[Put\]\] internal method of  \$O\$  with arguments  \$to\$ ,  \$fromVal\$ , and true.](#)
    - iv. [ReturnIfAbrupt\(putStatus\)](#)
  - e. Else,  $fromPresent$  is `false`
    - i. [Let deleteStatus be the result of calling the \[\[Delete\]\] internal method of  \$O\$  with arguments  \$to\$  and true.](#)
    - ii. [ReturnIfAbrupt\(deleteStatus\)](#)
  - f. Increase  $k$  by 1.
11. [Let deleteStatus be the result of calling the \[\[Delete\]\] internal method of  \$O\$  with arguments `ToString\(len-1\)` and true.](#)
12. [ReturnIfAbrupt\(deleteStatus\)](#)
13. [Let putStatus be the result of calling the \[\[Put\]\] internal method of  \$O\$  with arguments "length",  \$\(len-1\)\$ , and true.](#)
14. [ReturnIfAbrupt\(putStatus\)](#)
15. Return  $first$ .

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

#### 15.4.4.10 Array.prototype.slice (start, end)

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

1. Let  $O$  be the result of calling `ToObject` passing the `this` value as the argument.
2. [ReturnIfAbrupt\( \$O\$ \)](#)
3. Let  $A$  be [the result of the abstraction operation `ArrayCreate` \(15.4\) with argument 0](#).
4. Let  $lenVal$  be the result of calling the `[[Get]]` internal method of  $O$  with argument `"length"`.
5. Let  $len$  be `ToInt32(lenVal)`.
6. [ReturnIfAbrupt\(len\)](#)
7. Let  $relativeStart$  be `ToInteger(start)`.

<b>Deleted:</b> Call
<b>Deleted:</b> If <code>putStatus</code> is an abrupt completion, return <code>putStatus</code>
<b>Deleted:</b> If <code>first</code> is an abrupt completion, return <code>first</code>
<b>Deleted:</b> If <code>fromVal</code> is an abrupt completion, return <code>fromVal</code>
<b>Deleted:</b> Call
<b>Deleted:</b> If <code>putStatus</code> is an abrupt completion, return <code>putStatus</code>
<b>Deleted:</b> Call
<b>Deleted:</b> If <code>deleteStatus</code> is an abrupt completion, return <code>deleteStatus</code>
<b>Deleted:</b> Call
<b>Deleted:</b> If <code>putStatus</code> is an abrupt completion, return <code>putStatus</code>

8. *ReturnIfAbrupt(relativeStart)*

9. If *relativeStart* is negative, let *k* be  $\max((\text{len} + \text{relativeStart}), 0)$ ; else let *k* be  $\min(\text{relativeStart}, \text{len})$ .  
 10. If *end* is **undefined**, let *relativeEnd* be *len*; else let *relativeEnd* be *ToInteger(end)*.

11. *ReturnIfAbrupt(relativeEnd)*

12. If *relativeEnd* is negative, let *final* be  $\max((\text{len} + \text{relativeEnd}), 0)$ ; else let *final* be  $\min(\text{relativeEnd}, \text{len})$ .  
 13. Let *n* be 0.

14. Repeat, while *k* < *final*

- a. Let *Pk* be *ToString(k)*.
- b. Let *kPresent* be the result of calling the **[[HasProperty]]** internal method of *O* with argument *Pk*.
- c. If *kPresent* is **true**, then

i. Let *kValue* be the result of calling the **[[Get]]** internal method of *O* with argument *Pk*.

ii. *ReturnIfAbrupt(kValue)*

iii. Call the **[[DefineOwnProperty]]** internal method of *A* with arguments *ToString(n)*, **Property Descriptor** {**[[Value]]**: *kValue*, **[[Writable]]**: **true**, **[[Enumerable]]**: **true**, **[[Configurable]]**: **true**}, and **false**.

d. Increase *k* by 1.

e. Increase *n* by 1.

15. Return *A*.

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

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

#### 15.4.4.11 **Array.prototype.sort (comparefn)**

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

Let *obj* be the result of calling **ToObject** passing the **this** value as the argument.

Let *len* be the result of applying **Uint32** to the result of calling the **[[Get]]** internal method of *obj* with argument "**length**".

If *comparefn* is not **undefined** and is not a consistent comparison function for the elements of this array (see below), the behaviour of **sort** is implementation-defined.

Let *proto* be the value of the **[[Prototype]]** internal property of *obj*. If *proto* is not **null** and there exists an integer *j* such that all of the conditions below are satisfied then the behaviour of **sort** is implementation-defined:

- *obj* is sparse (15.4)
- $0 \leq j < \text{len}$
- The result of calling the **[[HasProperty]]** internal method of *proto* with argument *ToString(j)* is **true**.

The behaviour of **sort** is also implementation defined if *obj* is sparse and any of the following conditions are true:

- The **[[Extensible]]** internal property of *obj* is **false**.
- Any array index property of *obj* whose name is a nonnegative integer less than *len* is a data property whose **[[Configurable]]** attribute is **false**.

The behaviour of **sort** is also implementation defined if any array index property of *obj* whose name is a nonnegative integer less than *len* is an accessor property or is a data property whose **[[Writable]]** attribute is **false**.

Otherwise, the following steps are taken.

**Deleted:** If *relativeStart* is an abrupt completion, return *relativeStart*

**Deleted:** If *relativeEnd* is an abrupt completion, return *relativeEnd*

**Deleted:** If *kValue* is an abrupt completion, return *kValue*

1. Perform an implementation-dependent sequence of calls to the `[[Get]]`, `[[Put]]`, and `[[Delete]]` internal methods of `obj` and to `SortCompare` (described below), where the first argument for each call to `[[Get]]`, `[[Put]]`, or `[[Delete]]` is a nonnegative integer less than `len` and where the arguments for calls to `SortCompare` are results of previous calls to the `[[Get]]` internal method. The throw argument to the `[[Put]]` and `[[Delete]]` internal methods will be the value `true`. If `obj` is not sparse then `[[Delete]]` must not be called. If an abrupt completion is returned from any of these operations, it is immediately returned as the value of this function.
2. Return `obj`.

The returned object must have the following two properties.

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

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

A function `comparefn` is a consistent comparison function for a set of values  $S$  if all of the requirements below are met for all values  $a$ ,  $b$ , and  $c$  (possibly the same value) in the set  $S$ : The notation  $a <_{\text{CF}} b$  means  $\text{comparefn}(a,b) < 0$ ;  $a =_{\text{CF}} b$  means  $\text{comparefn}(a,b) = 0$  (of either sign); and  $a >_{\text{CF}} b$  means  $\text{comparefn}(a,b) > 0$ .

- Calling `comparefn(a,b)` always returns the same value  $v$  when given a specific pair of values  $a$  and  $b$  as its two arguments. Furthermore, `Type(v)` is `Number`, and  $v$  is not `Nan`. Note that this implies that exactly one of  $a <_{\text{CF}} b$ ,  $a =_{\text{CF}} b$ , and  $a >_{\text{CF}} b$  will be true for a given pair of  $a$  and  $b$ .
- Calling `comparefn(a,b)` does not modify the `this` object.
- $a =_{\text{CF}} a$  (reflexivity)
- If  $a =_{\text{CF}} b$ , then  $b =_{\text{CF}} a$  (symmetry)
- If  $a =_{\text{CF}} b$  and  $b =_{\text{CF}} c$ , then  $a =_{\text{CF}} c$  (transitivity of  $=_{\text{CF}}$ )
- If  $a <_{\text{CF}} b$  and  $b <_{\text{CF}} c$ , then  $a <_{\text{CF}} c$  (transitivity of  $<_{\text{CF}}$ )
- If  $a >_{\text{CF}} b$  and  $b >_{\text{CF}} c$ , then  $a >_{\text{CF}} c$  (transitivity of  $>_{\text{CF}}$ )

**NOTE** The above conditions are necessary and sufficient to ensure that `comparefn` divides the set  $S$  into equivalence classes and that these equivalence classes are totally ordered.

When the `SortCompare` abstract operation is called with two arguments  $j$  and  $k$ , the following steps are taken:

1. Let `jString` be `ToString(j)`.
2. Let `kString` be `ToString(k)`.
3. Let `hasj` be the result of calling the `[[HasProperty]]` internal method of `obj` with argument `jString`.
4. Let `hask` be the result of calling the `[[HasProperty]]` internal method of `obj` with argument `kString`.
5. If `hasj` and `hask` are both `false`, then return `+0`.
6. If `hasj` is `false`, then return `1`.
7. If `hask` is `false`, then return `-1`.
8. Let `x` be the result of calling the `[[Get]]` internal method of `obj` with argument `jString`.  
9. `ReturnIfAbrupt(x)`.
10. Let `y` be the result of calling the `[[Get]]` internal method of `obj` with argument `kString`.  
11. `ReturnIfAbrupt(y)`.
12. If `x` and `y` are both `undefined`, return `+0`.
13. If `x` is `undefined`, return `1`.
14. If `y` is `undefined`, return `-1`.
15. If the argument `comparefn` is not `undefined`, then
  - a. If `IsCallable(comparefn)` is `false`, throw a `TypeError` exception.
  - b. Return the result of calling the `[[Call]]` internal method of `comparefn` passing `undefined` as the `this` value and with arguments `x` and `y`.
16. Let `xString` be `ToString(x)`.  
17. `ReturnIfAbrupt(xString)`.

18. Let *yString* be *ToString(y)*.

19. ReturnIfAbrupt(*yString*).

20. If *xString* < *yString*, return -1.

21. If *xString* > *yString*, return 1.

22. Return +0.

NOTE 1 Because non-existent property values always compare greater than **undefined** property values, and **undefined** always compares greater than any other value, undefined property values always sort to the end of the result, followed by non-existent property values.

NOTE 2 The **sort** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method. Whether the **sort** function can be applied successfully to a host object is implementation-dependent.

#### 15.4.4.12 **Array.prototype.splice (start, deleteCount [, item1 [, item2 [, ...]]])**

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

1. Let *O* be the result of calling **ToObject** passing the **this** value as the argument.
2. ReturnIfAbrupt(*O*).
3. Let *A* be the result of the abstraction operation **ArrayCreate** (15.4) with argument 0.
4. Let *lenVal* be the result of calling the **[[Get]** internal method of *O* with argument "**length**".
5. Let *len* be **ToInt32**(*lenVal*).
6. ReturnIfAbrupt(*len*).
7. Let *relativeStart* be **ToInteger**(*start*).
8. ReturnIfAbrupt(*relativeStart*).
9. If *relativeStart* is negative, let *actualStart* be **max**((*len* + *relativeStart*), 0); else let *actualStart* be **min**(*relativeStart*, *len*).
10. Let *actualDeleteCount* be **min**(**max**(**ToInteger**(*deleteCount*), 0), *len* - *actualStart*).
11. Let *k* be 0.
12. Repeat, while *k* < *actualDeleteCount*
  - a. Let *from* be **ToString**(*actualStart*+*k*).
  - b. Let *fromPresent* be the result of calling the **[[HasProperty]** internal method of *O* with argument *from*.
  - c. If *fromPresent* is **true**, then
    - i. Let *fromValue* be the result of calling the **[[Get]** internal method of *O* with argument *from*.
    - ii. ReturnIfAbrupt(*fromValue*).
    - iii. Call the **[[DefineOwnProperty]** internal method of *A* with arguments **ToString**(*k*), **Property Descriptor** {**[[Value]]**: *fromValue*, **[[Writable]]**: **true**, **[[Enumerable]]**: **true**, **[[Configurable]]**: **true**}, and **false**.
  - d. Increment *k* by 1.
13. Let *putStatus* be the result of calling the **[[Put]** internal method of *A* with arguments "**length**", *actualDeleteCount*, and **true**.
14. ReturnIfAbrupt(*putStatus*).
15. Let *items* be an internal List whose elements are, in left to right order, the portion of the actual argument list starting with *item1*. The list will be empty if no such items are present.
16. Let *itemCount* be the number of elements in *items*.
17. If *itemCount* < *actualDeleteCount*, then
  - a. Let *k* be *actualStart*.
  - b. Repeat, while *k* < (*len* - *actualDeleteCount*)
    - i. Let *from* be **ToString**(*k*+*actualDeleteCount*).
    - ii. Let *to* be **ToString**(*k*+*itemCount*).
    - iii. Let *fromPresent* be the result of calling the **[[HasProperty]** internal method of *O* with argument *from*.
    - iv. If *fromPresent* is **true**, then
      1. Let *fromValue* be the result of calling the **[[Get]** internal method of *O* with argument *from*.
      2. ReturnIfAbrupt(*fromValue*).

**Deleted:** a new array created as if by the expression **new Array()** where **Array** is the standard built-in constructor with that name

3. Let `putStatus` be the result of calling the `[[Put]]` internal method of  $O$  with arguments  $to$ ,  $fromValue$ , and `true`. Deleted: Call
4. `ReturnIfAbrupt(putStatus)`.
- v. Else,  $fromPresent$  is `false`
1. Let `deleteStatus` be the result of calling the `[[Delete]]` internal method of  $O$  with arguments  $to$  and `true`. Deleted: Call
  2. `ReturnIfAbrupt(deleteStatus)`.
- vi. Increase  $k$  by 1.
- c. Let  $k$  be  $len$ .
- d. Repeat, while  $k > (len - actualDeleteCount + itemCount)$
- i. Let `deleteStatus` be the result of calling the `[[Delete]]` internal method of  $O$  with arguments `ToString(k-1)` and `true`. Deleted: Call
  - ii. `ReturnIfAbrupt(deleteStatus)`.
  - iii. Decrease  $k$  by 1.
18. Else if  $itemCount > actualDeleteCount$ , then
- a. Let  $k$  be  $(len - actualDeleteCount)$ .
  - b. Repeat, while  $k > actualStart$ 
    - i. Let  $from$  be `ToString(k + actualDeleteCount - 1).`
    - ii. Let  $to$  be `ToString(k + itemCount - 1)`
    - iii. Let  $fromPresent$  be the result of calling the `[[HasProperty]]` internal method of  $O$  with argument  $from$ .
    - iv. If  $fromPresent$  is `true`, then
      1. Let  $fromValue$  be the result of calling the `[[Get]]` internal method of  $O$  with argument  $from$ .
      2. `ReturnIfAbrupt(fromValue)`.
      3. Let `putStatus` be the result of calling the `[[Put]]` internal method of  $O$  with arguments  $to$ ,  $fromValue$ , and `true`. Deleted: Call
      4. `ReturnIfAbrupt(putStatus)`.
    - v. Else,  $fromPresent$  is `false`
      1. Let `deleteStatus` be the result of calling the `[[Delete]]` internal method of  $O$  with argument  $to$  and `true`. Deleted: Call
      2. `ReturnIfAbrupt(deleteStatus)`.
    - vi. Decrease  $k$  by 1.

19. Let  $k$  be  $actualStart$ .

20. Repeat, while  $items$  is not empty

    - a. Remove the first element from  $items$  and let  $E$  be the value of that element.
    - b. Let `putStatus` be the result of calling the `[[Put]]` internal method of  $O$  with arguments `ToString(k)`,  $E$ , and `true`. Deleted: Call
    - c. `ReturnIfAbrupt(putStatus)`.
    - d. Increase  $k$  by 1.

21. Let `putStatus` be the result of calling the `[[Put]]` internal method of  $O$  with arguments `"length"`,  $(len - actualDeleteCount + itemCount)$ , and `true`. Deleted: Call

22. `ReturnIfAbrupt(putStatus)`.

23. Return  $A$ .

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

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

#### 15.4.4.13 Array.prototype.unshift ( [ item1 [, item2 [, ... ]]] )

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

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

1. Let  $O$  be the result of calling `ToObject` passing the `this` value as the argument.

2. *ReturnIfAbrupt(O).*
3. Let *lenVal* be the result of calling the [[Get]] internal method of *O* with argument "length".
  4. Let *len* be ToUint32(*lenVal*).

5. *ReturnIfAbrupt(len).*

  6. Let *argCount* be the number of actual arguments.
  7. Let *k* be *len*.
  8. Repeat, while *k* > 0,
    - a. Let *from* be ToString(*k*-1).
    - b. Let *to* be ToString(*k+argCount*-1).
    - c. Let *fromPresent* be the result of calling the [[HasProperty]] internal method of *O* with argument *from*.
    - d. If *fromPresent* is true, then
      - i. Let *fromValue* be the result of calling the [[Get]] internal method of *O* with argument *from*.
      - ii. *ReturnIfAbrupt(fromValue).*
      - iii. *Let putStatus be the result of calling the [[Put]] internal method of O with arguments to, fromValue, and true.*
      - iv. *ReturnIfAbrupt(putStatus).*
    - e. Else, *fromPresent* is false
      - i. *Let deleteStatus be the result of calling the [[Delete]] internal method of O with arguments to, and true.*
      - ii. *ReturnIfAbrupt(deleteStatus).*
    - f. Decrease *k* by 1.
  9. Let *j* be 0.
  10. Let *items* be an internal List whose elements are, in left to right order, the arguments that were passed to this function invocation.
  11. Repeat, while *items* is not empty
    - a. Remove the first element from *items* and let *E* be the value of that element.
    - b. *Let putStatus be the result of calling the [[Put]] internal method of O with arguments ToString(j), E, and true.*
    - c. *ReturnIfAbrupt(putStatus).*
    - d. Increase *j* by 1.
  12. *Let putStatus be the result of calling the [[Put]] internal method of O with arguments "length", len+argCount, and true.*

13. *ReturnIfAbrupt(putStatus).*

  14. Return *len+argCount*.

The length property of the unshift method is 1.

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

#### 15.4.4.14 Array.prototype.indexOf ( *searchElement* [ , *fromIndex* ] )

**indexOf** compares *searchElement* to the elements of the array, in ascending order, using the internal Strict Equality Comparison Algorithm (11.9.1), and if found at one or more positions, returns the index of the first such position; otherwise, -1 is returned.

The optional second argument *fromIndex* defaults to 0 (i.e. the whole array is searched). If it is greater than or equal to the length of the array, -1 is returned, i.e. the array will not be searched. If it is negative, it is used as the offset from the end of the array to compute *fromIndex*. If the computed index is less than 0, the whole array will be searched.

When the **indexOf** method is called with one or two arguments, the following steps are taken:

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

2. *ReturnIfAbrupt(O).*

3. Let *lenValue* be the result of calling the [[Get]] internal method of *O* with the argument "length".
4. Let *len* be ToUint32(*lenValue*).

**Deleted:** Call

**Deleted:** Call

**Deleted:** Call

**Deleted:** Call

**Deleted:** 6

5. *ReturnIfAbrupt(len)*

6. If *len* is 0, return -1.
  7. If argument *fromIndex* was passed let *n* be *ToInteger(fromIndex)*; else let *n* be 0.
  8. *ReturnIfAbrupt(n)*
  9. If *n*  $\geq$  *len*, return -1.
  10. If *n*  $\geq$  0, then
    - a. Let *k* be *n*.
  11. Else, *n* < 0
    - a. Let *k* be *len* - *abs(n)*.
    - b. If *k*  $\leq$  0, then let *k* be 0.
  12. Repeat, while *k* < *len*
    - a. Let *kPresent* be the result of calling the **[[HasProperty]]** internal method of *O* with argument *ToString(k)*.
    - b. If *kPresent* is **true**, then
      - i. Let *elementK* be the result of calling the **[[Get]]** internal method of *O* with the argument *ToString(k)*.
      - ii. *ReturnIfAbrupt(elementK)*
      - iii. Let *same* be the result of applying the Strict Equality Comparison Algorithm to *searchElement* and *elementK*.
      - iv. If *same* is **true**, return *k*.
  - c. Increase *k* by 1.
13. Return -1.

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

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

#### 15.4.4.15 **Array.prototype.lastIndexOf ( *searchElement* [ , *fromIndex* ] )**

**lastIndexOf** compares *searchElement* to the elements of the array in descending order using the internal Strict Equality Comparison Algorithm (11.9.1), and if found at one or more positions, returns the index of the last such position; otherwise, -1 is returned.

**Deleted:** 6

The optional second argument *fromIndex* defaults to the array's length minus one (i.e. the whole array is searched). If it is greater than or equal to the length of the array, the whole array will be searched. If it is negative, it is used as the offset from the end of the array to compute *fromIndex*. If the computed index is less than 0, -1 is returned.

When the **lastIndexOf** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the result of calling **ToObject** passing the **this** value as the argument.
2. *ReturnIfAbrupt(O)*
3. Let *lenValue* be the result of calling the **[[Get]]** internal method of *O* with the argument "**length**".
4. Let *len* be *ToInt32(lenValue)*.
5. *ReturnIfAbrupt(len)*
6. If *len* is 0, return -1.
7. If argument *fromIndex* was passed let *n* be *ToInteger(fromIndex)*; else let *n* be *len*-1.
8. *ReturnIfAbrupt(n)*
9. If *n*  $\geq$  0, then let *k* be *min(n, len - 1)*.
10. Else, *n* < 0
  - a. Let *k* be *len* - *abs(n)*.
11. Repeat, while *k*  $\geq$  0
  - a. Let *kPresent* be the result of calling the **[[HasProperty]]** internal method of *O* with argument *ToString(k)*.
  - b. If *kPresent* is **true**, then
    - i. Let *elementK* be the result of calling the **[[Get]]** internal method of *O* with the argument *ToString(k)*.
    - ii. *ReturnIfAbrupt(elementK)*

**Deleted:** If *n* is an abrupt completion, return *n*.

**Deleted:** If *elementK* is an abrupt completion, return *elementK*.

- iii. Let *same* be the result of applying the Strict Equality Comparison Algorithm to *searchElement* and *elementK*.
- iv. If *same* is **true**, return *k*.

c. Decrease *k* by 1.

12. Return -1.

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

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

#### 15.4.4.16 Array.prototype.every ( callbackfn [ , thisArg ] )

*callbackfn* should be a function that accepts three arguments and returns a value that is coercible to the Boolean value **true** or **false**. **every** calls *callbackfn* once for each element present in the array, in ascending order, until it finds one where *callbackfn* returns **false**. If such an element is found, **every** immediately returns **false**. Otherwise, if *callbackfn* returned **true** for all elements, **every** will return **true**. *callbackfn* is called only for elements of the array which actually exist; it is not called for missing elements of the array.

If a *thisArg* parameter is provided, it will be used as the **this** value for each invocation of *callbackfn*. If it is not provided, **undefined** is used instead.

*callbackfn* is called with three arguments: the value of the element, the index of the element, and the object being traversed.

**every** does not directly mutate the object on which it is called but the object may be mutated by the calls to *callbackfn*.

The range of elements processed by **every** is set before the first call to *callbackfn*. Elements which are appended to the array after the call to **every** begins will not be visited by *callbackfn*. If existing elements of the array are changed, their value as passed to *callbackfn* will be the value at the time **every** visits them; elements that are deleted after the call to **every** begins and before being visited are not visited. **every** acts like the "for all" quantifier in mathematics. In particular, for an empty array, it returns **true**.

When the **every** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the result of calling **ToObject** passing the **this** value as the argument.
2. **ReturnIfAbrupt(O)**.
3. Let *lenValue* be the result of calling the **[[Get]** internal method of *O* with the argument "**length**".
4. Let *len* be **ToUint32(lenValue)**.
5. **ReturnIfAbrupt(len)**.
6. If **IsCallable(callbackfn)** is **false**, throw a **TypeError** exception.
7. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
8. Let *k* be 0.
9. Repeat, while *k* < *len*.
  - a. Let *Pk* be **ToString(k)**.
  - b. Let *kPresent* be the result of calling the **[[HasProperty]** internal method of *O* with argument *Pk*.
  - c. If *kPresent* is **true**, then
    - i. Let *kValue* be the result of calling the **[[Get]** internal method of *O* with argument *Pk*.
    - ii. **ReturnIfAbrupt(kValue)**.
    - iii. Let *testResult* be the result of calling the **[[Call]** internal method of *callbackfn* with *T* as the **this** value and argument list containing *kValue*, *k*, and *O*.
    - iv. **ReturnIfAbrupt(testResult)**.
    - v. If **ToBoolean(testResult)** is **false**, return **false**.
  - d. Increase *k* by 1.
10. Return **true**.

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

**Deleted:** If *kValue* is an abrupt completion, return *kValue*.

**Deleted:** If *testResult* is an abrupt completion, return *testResult*.

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

#### 15.4.4.17 Array.prototype.some ( callbackfn [ , thisArg ] )

*callbackfn* should be a function that accepts three arguments and returns a value that is coercible to the Boolean value **true** or **false**. **some** calls *callbackfn* once for each element present in the array, in ascending order, until it finds one where *callbackfn* returns **true**. If such an element is found, **some** immediately returns **true**. Otherwise, **some** returns **false**. *callbackfn* is called only for elements of the array which actually exist; it is not called for missing elements of the array.

If a *thisArg* parameter is provided, it will be used as the **this** value for each invocation of *callbackfn*. If it is not provided, **undefined** is used instead.

*callbackfn* is called with three arguments: the value of the element, the index of the element, and the object being traversed.

**some** does not directly mutate the object on which it is called but the object may be mutated by the calls to *callbackfn*.

The range of elements processed by **some** is set before the first call to *callbackfn*. Elements that are appended to the array after the call to **some** begins will not be visited by *callbackfn*. If existing elements of the array are changed, their value as passed to *callbackfn* will be the value at the time that **some** visits them; elements that are deleted after the call to **some** begins and before being visited are not visited. **some** acts like the "exists" quantifier in mathematics. In particular, for an empty array, it returns **false**.

When the **some** method is called with one or two arguments, the following steps are taken:

1. Let *O* be the result of calling **ToObject** passing the **this** value as the argument.
2. [ReturnIfAbrupt\(\*O\*\)](#).
3. Let *lenValue* be the result of calling the **[[Get]** internal method of *O* with the argument **"length"**.
4. Let *len* be **ToUint32(*lenValue*)**.
5. [ReturnIfAbrupt\(\*len\*\)](#).
6. If **IsCallable(*callbackfn*)** is **false**, throw a **TypeError** exception.
7. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
8. Let *k* be 0.
9. Repeat, while *k* < *len*
  - a. Let *Pk* be **ToString(*k*)**.
  - b. Let *kPresent* be the result of calling the **[[HasProperty]** internal method of *O* with argument *Pk*.
  - c. If *kPresent* is **true**, then
    - i. Let *kValue* be the result of calling the **[[Get]** internal method of *O* with argument *Pk*.
    - ii. [ReturnIfAbrupt\(\*kValue\*\)](#).
    - iii. Let *testResult* be the result of calling the **[[Call]** internal method of *callbackfn* with *T* as the **this** value and argument list containing *kValue*, *k*, and *O*.
    - iv. [ReturnIfAbrupt\(\*testResult\*\)](#).
    - v. If **ToBoolean(*testResult*)** is **true**, return **true**.
10. Increase *k* by 1.

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

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

**Deleted:** If *kValue* is an abrupt completion, return *kValue*.

**Deleted:** If *testResult* is an abrupt completion, return *testResult*.

#### 15.4.4.18 Array.prototype.forEach ( callbackfn [ , thisArg ] )

*callbackfn* should be a function that accepts three arguments. **forEach** calls *callbackfn* once for each element present in the array, in ascending order. *callbackfn* is called only for elements of the array which actually exist; it is not called for missing elements of the array.

If a *thisArg* parameter is provided, it will be used as the **this** value for each invocation of *callbackfn*. If it is not provided, **undefined** is used instead.

*callbackfn* is called with three arguments: the value of the element, the index of the element, and the object being traversed.

**forEach** does not directly mutate the object on which it is called but the object may be mutated by the calls to *callbackfn*.

The range of elements processed by **forEach** is set before the first call to *callbackfn*. Elements which are appended to the array after the call to **forEach** begins will not be visited by *callbackfn*. If existing elements of the array are changed, their value as passed to callback will be the value at the time **forEach** visits them; elements that are deleted after the call to **forEach** begins and before being visited are not visited.

When the **forEach** method is called with one or two arguments, the following steps are taken:

15. Let *O* be the result of calling **ToObject** passing the **this** value as the argument.
16. **ReturnIfAbrupt(*O*)**.
17. Let *lenValue* be the result of calling the **[[Get]]** internal method of *O* with the argument **"length"**.
18. Let *len* be **ToInt32(*lenValue*)**.
19. **ReturnIfAbrupt(*len*)**.
20. If **IsCallable(*callbackfn*)** is **false**, throw a **TypeError** exception.
21. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
22. Let *k* be 0.
23. Repeat, while *k* < *len*
  - a. Let *Pk* be **ToString(*k*)**.
  - b. Let *kPresent* be the result of calling the **[[HasProperty]]** internal method of *O* with argument *Pk*.
  - c. If *kPresent* is **true**, then
    - i. Let *kValue* be the result of calling the **[[Get]]** internal method of *O* with argument *Pk*.
    - ii. **ReturnIfAbrupt(*kValue*)**.
    - iii. Let *funcResult* be the result of calling **[[Call]]** internal method of *callbackfn* with *T* as the **this** value and argument list containing *kValue*, *k*, and *O*.
    - iv. **ReturnIfAbrupt(*funcResult*)**.
  - d. Increase *k* by 1.
24. Return **undefined**.

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

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

#### 15.4.4.19 Array.prototype.map ( callbackfn [ , thisArg ] )

*callbackfn* should be a function that accepts three arguments. **map** calls *callbackfn* once for each element in the array, in ascending order, and constructs a new Array from the results. *callbackfn* is called only for elements of the array which actually exist; it is not called for missing elements of the array.

If a *thisArg* parameter is provided, it will be used as the **this** value for each invocation of *callbackfn*. If it is not provided, **undefined** is used instead.

*callbackfn* is called with three arguments: the value of the element, the index of the element, and the object being traversed.

**Deleted:** If *kValue* is an abrupt completion, return *kValue*.

**Deleted:** Call

**Deleted:** If *funcResult* is an abrupt completion, return *funcResult*.

`map` does not directly mutate the object on which it is called but the object may be mutated by the calls to `callbackfn`.

The range of elements processed by `map` is set before the first call to `callbackfn`. Elements which are appended to the array after the call to `map` begins will not be visited by `callbackfn`. If existing elements of the array are changed, their value as passed to `callbackfn` will be the value at the time `map` visits them; elements that are deleted after the call to `map` begins and before being visited are not visited.

When the `map` method is called with one or two arguments, the following steps are taken:

1. Let  $O$  be the result of calling `ToObject` passing the `this` value as the argument.
2. [ReturnIfAbrupt\( \$O\$ \)](#)
3. Let  $lenValue$  be the result of calling the `[[Get]]` internal method of  $O$  with the argument `"length"`.
4. Let  $len$  be `ToUint32(lenValue)`.
5. [ReturnIfAbrupt\(len\)](#)
6. If `IsCallable(callbackfn)` is `false`, throw a `TypeError` exception.
7. If `thisArg` was supplied, let  $T$  be `thisArg`; else let  $T$  be `undefined`.
8. Let  $A$  be [the result of the abstraction operation `ArrayCreate` \(15.4\) with argument len](#).
9. Let  $k$  be 0.
10. Repeat, while  $k < len$ 
  - a. Let  $Pk$  be `ToString(k)`.
  - b. Let  $kPresent$  be the result of calling the `[[HasProperty]]` internal method of  $O$  with argument  $Pk$ .
  - c. If  $kPresent$  is `true`, then
    - i. Let  $kValue$  be the result of calling the `[[Get]]` internal method of  $O$  with argument  $Pk$ .
    - ii. [ReturnIfAbrupt\(kValue\)](#)
    - iii. Let  $mappedValue$  be the result of calling the `[[Call]]` internal method of `callbackfn` with  $T$  as the `this` value and argument [List](#) containing  $kValue$ ,  $k$ , and  $O$ .
    - iv. [ReturnIfAbrupt\(mappedValue\)](#)
    - v. Let [defineStatus](#) be the result of calling the `[[DefineOwnProperty]]` internal method of  $A$  with arguments  $Pk$ , Property Descriptor `{[[Value]]: mappedValue, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, and `false`.
    - vi. [ReturnIfAbrupt\(defineStatus\)](#)
  - d. Increase  $k$  by 1.
11. Return  $A$ .

The `length` property of the `map` method is 1.

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

#### 15.4.4.20 `Array.prototype.filter ( callbackfn [ , thisArg ] )`

`callbackfn` should be a function that accepts three arguments and returns a value that is coercible to the Boolean value `true` or `false`. `filter` calls `callbackfn` once for each element in the array, in ascending order, and constructs a new array of all the values for which `callbackfn` returns `true`. `callbackfn` is called only for elements of the array which actually exist; it is not called for missing elements of the array.

If a `thisArg` parameter is provided, it will be used as the `this` value for each invocation of `callbackfn`. If it is not provided, `undefined` is used instead.

`callbackfn` is called with three arguments: the value of the element, the index of the element, and the object being traversed.

`filter` does not directly mutate the object on which it is called but the object may be mutated by the calls to `callbackfn`.

The range of elements processed by `filter` is set before the first call to `callbackfn`. Elements which are appended to the array after the call to `filter` begins will not be visited by `callbackfn`. If existing elements of

**Deleted:** a new array created as if by the expression `new Array(len)` where `Array` is the standard built-in constructor with that name and `len` is the value of

**Deleted:** list

**Deleted:** Call

**Deleted:** defineResult

**Deleted:** defineResult

the array are changed their value as passed to `callbackfn` will be the value at the time `filter` visits them; elements that are deleted after the call to `filter` begins and before being visited are not visited.

When the `filter` method is called with one or two arguments, the following steps are taken:

1. Let  $O$  be the result of calling `ToObject` passing the `this` value as the argument.  
[2. `ReturnIfAbrupt\(O\)`](#).
3. Let  $lenValue$  be the result of calling the `[[Get]` internal method of  $O$  with the argument `"length"`.
4. Let  $len$  be `ToInt32(lenValue).`
- [5. `ReturnIfAbrupt\(len\)`](#).
6. If `IsCallable(callbackfn)` is `false`, throw a `TypeError` exception.
7. If `thisArg` was supplied, let  $T$  be `thisArg`; else let  $T$  be `undefined`.
8. Let  $A$  be [the result of the abstraction operation `ArrayCreate` \(15.4\) with argument 0](#).
9. Let  $k$  be 0.
10. Let  $to$  be 0.
11. Repeat, while  $k < len$ 
  - a. Let  $Pk$  be `ToString(k).`
  - b. Let  $kPresent$  be the result of calling the `[[HasProperty]` internal method of  $O$  with argument  $Pk$ .
  - c. If  $kPresent$  is `true`, then
    - i. Let  $kValue$  be the result of calling the `[[Get]` internal method of  $O$  with argument  $Pk$ .
    - [ii. If  \$kValue\$  is an abrupt completion, return  \$kValue\$ .](#)
    - iii. Let  $selected$  be the result of calling the `[[Call]` internal method of `callbackfn` with  $T$  as the `this` value and argument list containing  $kValue$ ,  $k$ , and  $O$ .
    - [iv. `ReturnIfAbrupt\(selected\)`](#).
    - v. If `ToBoolean(selected)` is `true`, then
      1. [Let `defineStatus` be the result of calling the `\[\[DefineOwnProperty\]` internal method of  \$A\$  with arguments `ToString\(to\)`, Property Descriptor `{\[\[Value\]\]: kValue, \[\[Writable\]\]: true, \[\[Enumerable\]\]: true, \[\[Configurable\]\]: true}`, and `false`.](#)
      - [2. `ReturnIfAbrupt\(defineStatus\)`](#).
      3. Increase  $to$  by 1.
    - d. Increase  $k$  by 1.
12. Return  $A$ .

The `length` property of the `filter` method is 1.

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

#### 15.4.4.21 `Array.prototype.reduce ( callbackfn [ , initialValue ] )`

`callbackfn` should be a function that takes four arguments. `reduce` calls the callback, as a function, once for each element present in the array, in ascending order.

`callbackfn` is called with four arguments: the `previousValue` (or value from the previous call to `callbackfn`), the `currentValue` (value of the current element), the `currentIndex`, and the object being traversed. The first time that callback is called, the `previousValue` and `currentValue` can be one of two values. If an `initialValue` was provided in the call to `reduce`, then `previousValue` will be equal to `initialValue` and `currentValue` will be equal to the first value in the array. If no `initialValue` was provided, then `previousValue` will be equal to the first value in the array and `currentValue` will be equal to the second. It is a `TypeError` if the array contains no elements and `initialValue` is not provided.

`reduce` does not directly mutate the object on which it is called but the object may be mutated by the calls to `callbackfn`.

The range of elements processed by `reduce` is set before the first call to `callbackfn`. Elements that are appended to the array after the call to `reduce` begins will not be visited by `callbackfn`. If existing elements of the array are changed, their value as passed to `callbackfn` will be the value at the time `reduce` visits them; elements that are deleted after the call to `reduce` begins and before being visited are not visited.

**Deleted:** a new array created as if by the expression `new Array()` where `Array` is the standard built-in constructor with that name

**Deleted:** If `selected` is an abrupt completion, return `selected`

**Deleted:** Call

When the `reduce` method is called with one or two arguments, the following steps are taken:

1. Let  $O$  be the result of calling `ToObject` passing the `this` value as the argument.
- ~~2. `ReturnIfAbrupt( $O$ )`.~~
3. Let  $lenValue$  be the result of calling the `[[Get]]` internal method of  $O$  with the argument `"length"`.
4. Let  $len$  be `ToUint32(lenValue)`.
- ~~5. `ReturnIfAbrupt(len)`.~~
6. If `IsCallable(callbackfn)` is `false`, throw a `TypeError` exception.
7. If  $len$  is 0 and  $initialValue$  is not present, throw a `TypeError` exception.
8. Let  $k$  be 0.
9. If  $initialValue$  is present, then
  - a. Set  $accumulator$  to  $initialValue$ .
10. Else,  $initialValue$  is not present
  - a. Let  $kPresent$  be `false`.
  - b. Repeat, while  $kPresent$  is `false` and  $k < len$ 
    - i. Let  $Pk$  be `ToString(k)`.
    - ii. Let  $kPresent$  be the result of calling the `[[HasProperty]]` internal method of  $O$  with argument  $Pk$ .
    - iii. If  $kPresent$  is `true`, then
      1. Let  $accumulator$  be the result of calling the `[[Get]]` internal method of  $O$  with argument  $Pk$ .
      - ~~2. `ReturnIfAbrupt(accumulator)`.~~
      - iv. Increase  $k$  by 1.
    - c. If  $kPresent$  is `false`, throw a `TypeError` exception.
11. Repeat, while  $k < len$ 
  - a. Let  $Pk$  be `ToString(k)`.
  - b. Let  $kPresent$  be the result of calling the `[[HasProperty]]` internal method of  $O$  with argument  $Pk$ .
  - c. If  $kPresent$  is `true`, then
    - i. Let  $kValue$  be the result of calling the `[[Get]]` internal method of  $O$  with argument  $Pk$ .
    - ~~ii. `ReturnIfAbrupt(kValue)`.~~
    - iii. Let  $accumulator$  be the result of calling the `[[Call]]` internal method of  $callbackfn$  with `undefined` as the `this` value and argument list containing  $accumulator$ ,  $kValue$ ,  $k$ , and  $O$ .
    - ~~iv. `ReturnIfAbrupt(accumulator)`.~~
  - d. Increase  $k$  by 1.
12. Return  $accumulator$ .

The `length` property of the `reduce` method is 1.

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

#### 15.4.4.22 `Array.prototype.reduceRight ( callbackfn [ , initialValue ] )`

`callbackfn` should be a function that takes four arguments. `reduceRight` calls the callback, as a function, once for each element present in the array, in descending order.

`callbackfn` is called with four arguments: the `previousValue` (or `value` from the previous call to `callbackfn`), the `currentValue` (value of the current element), the `currentIndex`, and the object being traversed. The first time the function is called, the `previousValue` and `currentValue` can be one of two values. If an `initialValue` was provided in the call to `reduceRight`, then `previousValue` will be equal to `initialValue` and `currentValue` will be equal to the last value in the array. If no `initialValue` was provided, then `previousValue` will be equal to the last value in the array and `currentValue` will be equal to the second-to-last value. It is a `TypeError` if the array contains no elements and `initialValue` is not provided.

`reduceRight` does not directly mutate the object on which it is called but the object may be mutated by the calls to `callbackfn`.

The range of elements processed by `reduceRight` is set before the first call to `callbackfn`. Elements that are appended to the array after the call to `reduceRight` begins will not be visited by `callbackfn`. If existing

**Deleted:** If `accumulator` is an abrupt completion  
return `accumulator`.

**Deleted:** If `kValue` is an abrupt completion,  
`kValue`

**Deleted:** If `accumulator` is an abrupt completion  
return `accumulator`

elements of the array are changed by `callbackfn`, their value as passed to `callbackfn` will be the value at the time `reduceRight` visits them; elements that are deleted after the call to `reduceRight` begins and before being visited are not visited.

When the `reduceRight` method is called with one or two arguments, the following steps are taken:

1. Let  $O$  be the result of calling `ToObject` passing the `this` value as the argument.
2. `ReturnIfAbrupt( $O$ )`.
3. Let  $lenValue$  be the result of calling the `[[Get]]` internal method of  $O$  with the argument "`length`".
4. Let  $len$  be `ToUint32(lenValue)`.
5. `ReturnIfAbrupt(len)`.
6. If `IsCallable(callbackfn)` is `false`, throw a `TypeError` exception.
7. If  $len$  is 0 and  $initialValue$  is not present, throw a `TypeError` exception.
8. Let  $k$  be  $len - 1$ .
9. If  $initialValue$  is present, then
  - a. Set `accumulator` to  $initialValue$ .
10. Else,  $initialValue$  is not present
  - a. Let  $kPresent$  be `false`.
  - b. Repeat, while  $kPresent$  is `false` and  $k \geq 0$ 
    - i. Let  $P_k$  be `ToString(k)`.
    - ii. Let  $kPresent$  be the result of calling the `[[HasProperty]]` internal method of  $O$  with argument  $P_k$ .
    - iii. If  $kPresent$  is `true`, then
      1. Let `accumulator` be the result of calling the `[[Get]]` internal method of  $O$  with argument  $P_k$ .
      2. `ReturnIfAbrupt(accumulator)`.
    - iv. Decrease  $k$  by 1.
  - c. If  $kPresent$  is `false`, throw a `TypeError` exception.
11. Repeat, while  $k \geq 0$ 
  - a. Let  $P_k$  be `ToString(k)`.
  - b. Let  $kPresent$  be the result of calling the `[[HasProperty]]` internal method of  $O$  with argument  $P_k$ .
  - c. If  $kPresent$  is `true`, then
    - i. Let  $kValue$  be the result of calling the `[[Get]]` internal method of  $O$  with argument  $P_k$ .
    - ii. `ReturnIfAbrupt(kValue)`.
    - iii. Let `accumulator` be the result of calling the `[[Call]]` internal method of `callbackfn` with `undefined` as the `this` value and argument list containing `accumulator`,  $kValue$ ,  $k$ , and  $O$ .
    - iv. `ReturnIfAbrupt(accumulator)`.
  - d. Decrease  $k$  by 1.
12. Return `accumulator`.

The `length` property of the `reduceRight` method is 1.

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

#### 15.4.5 Properties of Array Instances

Array instances inherit properties from the Array prototype object and have the `[[NativeBrand]]` internal property `with value NativeArray`. Array instances also have the following properties.

##### 15.4.5.1 `[[DefineOwnProperty]] ( P, Desc, Throw )`

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

Assume  $A$  is an Array object,  $Desc$  is a Property Descriptor, and  $Throw$  is a Boolean flag.

In the following algorithm, the term "Reject" means "If  $Throw$  is `true`, then throw a `TypeError` exception, otherwise return `false`."

**Deleted:** If `accumulator` is an abrupt completion, return `accumulator`

**Deleted:** If  $kValue$  is an abrupt completion, return  $kValue$

**Deleted:** If `accumulator` is an abrupt completion, return `accumulator`

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  $oldLenDesc$  be the result of calling the `[[GetOwnProperty]]` internal method of  $A$  passing `"length"` as the argument. The result will never be **undefined** or an accessor descriptor because Array objects are created with a length data property that cannot be deleted or reconfigured.
2. Let  $oldLen$  be  $oldLenDesc.[[Value]]$ .
3. If  $P$  is `"length"`, then
  - a. If the `[[Value]]` field of  $Desc$  is absent, then
    - i. Return the result of calling the default `[[DefineOwnProperty]]` internal method (8.12.9) on  $A$  passing `"length"`,  $Desc$ , and  $Throw$  as arguments.
  - b. Let  $newLenDesc$  be a copy of  $Desc$ .
  - c. Let  $newLen$  be  $\text{ToUint32}(Desc.[[Value]])$ .
  - d. If  $newLen$  is not equal to  $\text{ToNumber}(Desc.[[Value]])$ , throw a **RangeError** exception.
  - e. Set  $newLenDesc.[[Value]]$  to  $newLen$ .
  - f. If  $newLen \geq oldLen$ , then
    - i. Return the result of calling the default `[[DefineOwnProperty]]` internal method (8.12.9) on  $A$  passing `"length"`,  $newLenDesc$ , and  $Throw$  as arguments.
  - g. Reject if  $oldLenDesc.[[Writable]]$  is **false**.
  - h. If  $newLenDesc.[[Writable]]$  is absent or has the value **true**, let  $newWritable$  be **true**.
  - i. Else,
    - i. Need to defer setting the `[[Writable]]` attribute to **false** in case any elements cannot be deleted.
    - ii. Let  $newWritable$  be **false**.
    - iii. Set  $newLenDesc.[[Writable]]$  to **true**.
  - j. Let  $succeeded$  be the result of calling the default `[[DefineOwnProperty]]` internal method (8.12.9) on  $A$  passing `"length"`,  $newLenDesc$ , and  $Throw$  as arguments.
  - k. If  $succeeded$  is **false**, return **false**.
  - l. While  $newLen < oldLen$  repeat,
    - i. Set  $oldLen$  to  $oldLen - 1$ .
    - ii. Let  $deleteSucceeded$  be the result of calling the `[[Delete]]` internal method of  $A$  passing  $\text{ToString}(oldLen)$  and **false** as arguments.
    - iii. If  $deleteSucceeded$  is **false**, then
      1. Set  $newLenDesc.[[Value]]$  to  $oldLen + 1$ .
      2. If  $newWritable$  is **false**, set  $newLenDesc.[[Writable]]$  to **false**.
      3. Call the default `[[DefineOwnProperty]]` internal method (8.12.9) on  $A$  passing `"length"`,  $newLenDesc$ , and **false** as arguments.
      4. Reject.
    - m. If  $newWritable$  is **false**, then
      - i. Call the default `[[DefineOwnProperty]]` internal method (8.12.9) on  $A$  passing `"length"`, Property Descriptor `{[[Writable]]: false}`, and **false** as arguments. This call will always return **true**.
    - n. Return **true**.
  4. Else if  $P$  is an array index (15.4), then
    - a. Let  $index$  be  $\text{ToUint32}(P)$ .
    - b. **ReturnIfAbrupt(index)**.
    - c. Reject if  $index \geq oldLen$  and  $oldLenDesc.[[Writable]]$  is **false**.
    - d. Let  $succeeded$  be the result of calling the default `[[DefineOwnProperty]]` internal method (8.12.9) on  $A$  passing  $P$ ,  $Desc$ , and **false** as arguments.
    - e. Reject if  $succeeded$  is **false**.
    - f. If  $index \geq oldLen$ 
      - i. Set  $oldLenDesc.[[Value]]$  to  $index + 1$ .
      - ii. Call the default `[[DefineOwnProperty]]` internal method (8.12.9) on  $A$  passing `"length"`,  $oldLenDesc$ , and **false** as arguments. This call will always return **true**.
    - g. Return **true**.
  5. Return the result of calling the default `[[DefineOwnProperty]]` internal method (8.12.9) on  $A$  passing  $P$ ,  $Desc$ , and  $Throw$  as arguments.

**Deleted:** `<#>ReturnIfAbrupt(A)`.

**Formatted:** Font: Italic

#### 15.4.5.2 `length`

The `length` property of this Array object is a data property whose value is always numerically greater than the name of every deletable property whose name is an array index.

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

**NOTE** Attempting to set the `length` property of an Array object to a value that is numerically less than or equal to the largest numeric property name of an existing array indexed non-deletable property of the array will result in the `length` being set to a numeric value that is one greater than that largest numeric property name. See 15.4.5.1.

### 15.5 String Objects

#### 15.5.1 The String Constructor Called as a Function

When `String` is called as a function rather than as a constructor, it performs a type conversion.

##### 15.5.1.1 `String ( [ value ] )`

Returns a String value (not a String object) computed by `ToString(value)`. If `value` is not supplied, the empty String `""` is returned.

#### 15.5.2 The String Constructor

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

##### 15.5.2.1 `new String ( [ value ] )`

The `[[Prototype]]` internal property of the newly constructed object is set to the standard built-in String prototype object that is the initial value of `String.prototype` (15.5.3.1).

The newly constructed object has the `[[NativeBrand]]` internal property with value `StringWrapper`.

The `[[Extensible]]` internal property of the newly constructed object is set to `true`.

The `[[PrimitiveValue]]` internal property of the newly constructed object is set to `ToString(value)`, or to the empty String if `value` is not supplied.

- Deleted:** `[[Class]]` internal property of the
- Deleted:** is set to `"String"`
- Deleted:** `IsString`

#### 15.5.3 Properties of the String Constructor

The value of the `[[Prototype]]` internal property of the String constructor is the standard built-in Function prototype object (15.3.4).

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

##### 15.5.3.1 `String.prototype`

The initial value of `String.prototype` is the standard built-in String prototype object (15.5.4).

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

##### 15.5.3.2 `String.fromCharCode ( [ char0 [, char1 [, ... ] ] ] )`

Returns a String value containing as many characters as the number of arguments. Each argument specifies one character of the resulting String, with the first argument specifying the first character, and so on, from left to right. An argument is converted to a character by applying the operation `ToUint16` (9.7) and regarding the

resulting 16-bit integer as the code unit value of a character. If no arguments are supplied, the result is the empty String.

The `length` property of the `fromCharCode` function is 1.

#### 15.5.4 Properties of the String Prototype Object

The String prototype object is itself a String object whose value is an empty String. [The String prototype object has the `\[\[NativeBrand\]\]` internal property with value `StringWrapper`.](#)

**Deleted:** (its `[[Class]]` is "string")

**Deleted:** `IsString`

The value of the `[[Prototype]]` internal property of the String prototype object is the standard built-in Object prototype object (15.2.4).

##### 15.5.4.1 `String.prototype.constructor`

The initial value of `String.prototype.constructor` is the built-in `String` constructor.

##### 15.5.4.2 `String.prototype.toString()`

Returns this String value. (Note that, for a String object, the `toString` method happens to return the same thing as the `valueOf` method.)

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

##### 15.5.4.3 `String.prototype.valueOf()`

Returns this String value.

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

##### 15.5.4.4 `String.prototype.charAt(pos)`

Returns a String containing the character at position `pos` in the String resulting from converting this object to a String. If there is no character at that position, the result is the empty String. The result is a String value, not a String object.

If `pos` is a value of Number type that is an integer, then the result of `x.charAt(pos)` is equal to the result of `x.substring(pos, pos+1)`.

When the `charAt` method is called with one argument `pos`, the following steps are taken:

1. [ReturnIfAbrupt\(CheckObjectCoercible\(this value\)\)](#).
2. Let `S` be the result of calling `ToString`, giving it the `this` value as its argument.
3. [ReturnIfAbrupt\(s\)](#).
4. Let `position` be `ToInteger(pos)`.
5. [ReturnIfAbrupt\(position\)](#).
6. Let `size` be the number of characters in `S`.
7. If `position < 0` or `position ≥ size`, return the empty String.
8. Return a String of length 1, containing one character from `S`, namely the character at position `position`, where the first (leftmost) character in `S` is considered to be at position 0, the next one at position 1, and so on.

**Deleted:** Call

**Deleted:** passing the

**Deleted:** as its argument

NOTE The `charAt` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

#### 15.5.4.5 String.prototype.charCodeAt(pos)

Returns a Number (a nonnegative integer less than  $2^{16}$ ) representing the code unit value of the character at position *pos* in the String resulting from converting this object to a String. If there is no character at that position, the result is **NaN**.

When the `charCodeAt` method is called with one argument *pos*, the following steps are taken:

1. `ReturnIfAbrupt(CheckObjectCoercible(this value))`
2. Let *S* be the result of calling `ToString`, giving it the `this` value as its argument.
3. `ReturnIfAbrupt(S)`
4. Let *position* be `ToInteger(pos)`.
5. `ReturnIfAbrupt(position)`
6. Let *size* be the number of characters in *S*.
7. If *position* < 0 or *position* ≥ *size*, return **NaN**.
8. Return a value of Number type, whose value is the code unit value of the character at position *position* in the String *S*, where the first (leftmost) character in *S* is considered to be at position 0, the next one at position 1, and so on.

NOTE The `charCodeAt` function is intentionally generic; it does not require that its `this` value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

#### 15.5.4.6 String.prototype.concat ([ string1 [, string2 [, ... ]]])

When the `concat` method is called with zero or more arguments *string1*, *string2*, etc., it returns a String consisting of the characters of this object (converted to a String) followed by the characters of each of *string1*, *string2*, etc. (where each argument is converted to a String). The result is a String value, not a String object. The following steps are taken:

1. `ReturnIfAbrupt(CheckObjectCoercible(this value))`
2. Let *S* be the result of calling `ToString`, giving it the `this` value as its argument.
3. `ReturnIfAbrupt(S)`
4. Let *args* be an internal list that is a copy of the argument list passed to this function.
5. Let *R* be *S*.
6. Repeat, while *args* is not empty
  - a. Remove the first element from *args* and let *next* be the value of that element.
  - b. `Let nextString be ToString(next)`
  - c. `ReturnIfAbrupt(nextString)`
  - d. Let *R* be the String value consisting of the characters in the previous value of *R* followed by the characters of *nextString*
7. Return *R*.

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

NOTE The `concat` function is intentionally generic; it does not require that its `this` value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

#### 15.5.4.7 String.prototype.indexOf(searchString, position)

If *searchString* appears as a substring of the result of converting this object to a String, at one or more positions that are greater than or equal to *position*, then the index of the smallest such position is returned; otherwise, -1 is returned. If *position* is **undefined**, 0 is assumed, so as to search all of the String.

The `indexOf` method takes two arguments, *searchString* and *position*, and performs the following steps:

1. `ReturnIfAbrupt(CheckObjectCoercible(this value))`
2. Let *S* be the result of calling `ToString`, giving it the `this` value as its argument.
3. `ReturnIfAbrupt(S)`
4. Let *searchStr* be `ToString(searchString)`.

**Deleted:** Call `CheckObjectCoercible` passing the `this` value as its argument.

**Deleted:** Call `CheckObjectCoercible` passing the `this` value as its argument.

**Deleted:** `ToString()`

**Deleted:** `)`

## 5. ReturnIfAbrupt(searchString).

6. Let *pos* be *ToInteger(position)*. (If *position* is **undefined**, this step produces the value **0**).
7. ReturnIfAbrupt(*pos*).
8. Let *len* be the number of characters in *S*.
9. Let *start* be *min(max(pos, 0), len)*.
10. Let *searchLen* be the number of characters in *searchStr*.
11. Return the smallest possible integer *k* not smaller than *start* such that *k+ searchLen* is not greater than *len*, and for all nonnegative integers *j* less than *searchLen*, the character at position *k+j* of *S* is the same as the character at position *j* of *searchStr*; but if there is no such integer *k*, then return the value **-1**.

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

NOTE The **indexOf** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

### 15.5.4.8 String.prototype.lastIndexOf (searchString, position)

If *searchString* appears as a substring of the result of converting *this* object to a String at one or more positions that are smaller than or equal to *position*, then the index of the greatest such position is returned; otherwise, **-1** is returned. If *position* is **undefined**, the length of the String value is assumed, so as to search all of the String.

The **lastIndexOf** method takes two arguments, *searchString* and *position*, and performs the following steps:

1. ReturnIfAbrupt(CheckObjectCoercible(this value)).
2. Let *S* be the result of calling *ToString*, giving it the **this** value as its argument.
3. ReturnIfAbrupt(*S*).
4. Let *searchStr* be *ToString(searchString)*.
5. ReturnIfAbrupt(*searchString*).
6. Let *numPos* be *ToNumber(position)*. (If *position* is **undefined**, this step produces the value **NaN**).
7. ReturnIfAbrupt(*numPos*).
8. If *numPos* is **NaN**, let *pos* be **+∞**; otherwise, let *pos* be *ToInteger(numPos)*.
9. Let *len* be the number of characters in *S*.
10. Let *start* *min(max(pos, 0), len)*.
11. Let *searchLen* be the number of characters in *searchStr*.
12. Return the largest possible nonnegative integer *k* not larger than *start* such that *k+ searchLen* is not greater than *len*, and for all nonnegative integers *j* less than *searchLen*, the character at position *k+j* of *S* is the same as the character at position *j* of *searchStr*; but if there is no such integer *k*, then return the value **-1**.

**Deleted:** Call *CheckObjectCoercible* passing the **this** value as its argument.

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

NOTE The **lastIndexOf** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

### 15.5.4.9 String.prototype.localeCompare (that)

When the **localeCompare** method is called with one argument *that*, it returns a Number other than **NaN** that represents the result of a locale-sensitive String comparison of the *this* value (converted to a String) with *that* (converted to a String). The two Strings are *S* and *That*. The two Strings are compared in an implementation-defined fashion. The result is intended to order String values in the sort order specified by the system default locale, and will be negative, zero, or positive, depending on whether *S* comes before *That* in the sort order, the Strings are equal, or *S* comes after *That* in the sort order, respectively.

Before perform the comparisons the following steps are performed to prepare the Strings:

1. ReturnIfAbrupt(CheckObjectCoercible(this value)).
2. Let *S* be the result of calling *ToString*, giving it the **this** value as its argument.
3. ReturnIfAbrupt(*S*).
4. Let *That* be *ToString(that)*.

**Deleted:** Call *CheckObjectCoercible* passing the **this** value as its argument.

## 5. ReturnIfAbrupt(That).

The `localeCompare` method, if considered as a function of two arguments `this` and `that`, is a consistent comparison function (as defined in 15.4.4.11) on the set of all Strings.

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

If no language-sensitive comparison at all is available from the host environment, this function may perform a bitwise comparison.

**NOTE 1** The `localeCompare` method itself is not directly suitable as an argument to `Array.prototype.sort` because the latter requires a function of two arguments.

**NOTE 2** This function is intended to rely on whatever language-sensitive comparison functionality is available to the ECMAScript environment from the host environment, and to compare according to the rules of the host environment's current locale. It is strongly recommended that this function treat Strings that are canonically equivalent according to the Unicode standard as identical (in other words, compare the Strings as if they had both been converted to Normalised Form C or D first). It is also recommended that this function not honour Unicode compatibility equivalences or decompositions.

**NOTE 3** The second parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

**NOTE 4** The `localeCompare` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

### 15.5.4.10 String.prototype.match (regexp)

When the `match` method is called with argument `regexp`, the following steps are taken:

1. ReturnIfAbrupt(CheckObjectCoercible(this value)).
2. Let `S` be the result of calling `ToString`, giving it the `this` value as its argument.
3. ReturnIfAbrupt(`S`).
4. If `Type(regexp)` is `Object` and `regexp` has a `[[NativeBrand]]` internal property whose value is `NativeRegExp`, then let `rx` be `regexp`;
5. Else, let `rx` be the result of the abstraction operation `RegExpCreate` (15.10.4.1) with arguments `regexp` and `undefined`.
6. ReturnIfAbrupt(`rx`).
7. Let `global` be the result of calling the `[[Get]]` internal method of `rx` with argument `"global"`.
8. ReturnIfAbrupt(`global`).
9. If `global` is not `true`, then
  - a. Return the result of calling the abstract operation `RegExpExec` (see 15.10.6.2) with arguments `rx` and `S`.
10. Else, `global` is `true`
  - a. Let `putStatus` be the result of calling the `[[Put]]` internal method of `rx` with arguments `"lastIndex"` and `0`.
  - b. ReturnIfAbrupt(`putStatus`).
  - c. Let `A` be the result of the abstraction operation `ArrayCreate` (15.4) with argument `0`.
  - d. Let `previousLastIndex` be `0`.
  - e. Let `n` be `0`.
  - f. Let `lastMatch` be `true`.
  - g. Repeat, while `lastMatch` is `true`
    - i. Let `result` be the result of calling the abstract operation `RegExpExec` (see 15.10.6.2) with arguments `rx` and `S`.
    - ii. ReturnIfAbrupt(`result`).
    - iii. If `result` is `null`, then set `lastMatch` to `false`.
    - iv. Else, `result` is not `null`

**Deleted:** Call `CheckObjectCoercible` passing the `this` value as its argument.

**Deleted:** the value of the `[[Class]]` internal property of

**Deleted:** is `"RegExp"`

**Deleted:** the

**Deleted:** `IsRegExp`

**Deleted:** a new `RegExp` object created as if by the expression `new RegExp (regexp)` where `RegExp` is the standard built-in constructor with that name.

**Deleted:** <#> Let `exec` be the standard built-in function `RegExp.prototype.exec` (see 15.10.6.2)¶

**Deleted:** `[[Call]]` internal method of `exec`

**Deleted:** as

**Deleted:** the `this` value and argument list containing

**Deleted:** `Call`

**Deleted:** a new array created as if by the expression `new Array ()` where `Array` is the standard built-in constructor with that name

**Deleted:** `[[Call]]` internal method of `exec`

**Deleted:** as the `this` value and argument list containing

1. Let *thisIndex* be the result of calling the [[Get]] internal method of *rx* with argument "lastIndex".
2. ReturnIfAbrupt(thisIndex)
3. If *thisIndex* = *previousLastIndex* then
  - a Let *putStatus* be the result of calling the [[Put]] internal method of *rx* with arguments "lastIndex" and *thisIndex*+1.
  - b ReturnIfAbrupt(putStatus).
  - c Set *previousLastIndex* to *thisIndex*+1.
4. Else, set *previousLastIndex* to *thisIndex*.
5. Let *matchStr* be the result of calling the [[Get]] internal method of *result* with argument "0".
6. ReturnIfAbrupt(matchStr).
7. Let *defineStatus* be the result of calling the [[DefineOwnProperty]] internal method of *A* with arguments *ToString(n)*, the Property Descriptor {[[Value]]: *matchStr*, [[Writable]]: true, [[Enumerable]]: true, [[configurable]]: true}, and false.
8. ReturnIfAbrupt(defineStatus).
9. Increment *n*.
  - h. If *n* = 0, then return null.
  - i. Return *A*.

NOTE The *match* function is intentionally generic; it does not require that its *this* value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

#### 15.5.4.11 String.prototype.replace (searchValue, replaceValue)

First set *string* according to the following steps:

1. ReturnIfAbrupt(CheckObjectCoercible(this value)).
2. Let *string* be the result of calling *ToString*, giving it the *this* value as its argument.
3. ReturnIfAbrupt(string).

If *searchValue* is a regular expression (an object that has a [[NativeBrand]] internal property whose value is NativeRegExp), do the following: If *searchValue.global* is false, then search *string* for the first match of the regular expression *searchValue*. If *searchValue.global* is true, then search *string* for all matches of the regular expression *searchValue*. Do the search in the same manner as in *String.prototype.match*, including the update of *searchValue.lastIndex*. Let *m* be the number of left capturing parentheses in *searchValue* (using *NcapturingParens* as specified in 15.10.2.1).

If *searchValue* is not a regular expression, let *searchString* be *ToString(searchValue)* and search *string* for the first occurrence of *searchString*. Let *m* be 0.

If *replaceValue* is a function, then for each matched substring, call the function with the following *m* + 3 arguments. Argument 1 is the substring that matched. If *searchValue* is a regular expression, the next *m* arguments are all of the captures in the MatchResult (see 15.10.2.1). Argument *m* + 2 is the offset within *string* where the match occurred, and argument *m* + 3 is *string*. The result is a String value derived from the original input by replacing each matched substring with the corresponding return value of the function call, converted to a String if need be.

Otherwise, let *newstring* denote the result of converting *replaceValue* to a String. The result is a String value derived from the original input String by replacing each matched substring with a String derived from *newstring* by replacing characters in *newstring* by replacement text as specified in Table 24. These \$ replacements are done left-to-right, and, once such a replacement is performed, the new replacement text is not subject to further replacements. For example, "\$1,\$2".replace(/(\\$(\d))/g, "\$\$1-\$1\$2") returns "\$1-\$11,\$1-\$22". A \$ in *newstring* that does not match any of the forms below is left as is.

Table 24 — Replacement Text Symbol Substitutions

Characters	Replacement text
\$\$	\$

<b>\$&amp;</b>	The matched substring.
<b>\$`</b>	The portion of <i>string</i> that precedes the matched substring.
<b>\$'</b>	The portion of <i>string</i> that follows the matched substring.
<b>\$n</b>	The <i>n</i> <sup>th</sup> capture, where <i>n</i> is a single digit in the range 1 to 9 and <b>\$n</b> is not followed by a decimal digit. If <i>n</i> ≤ <i>m</i> and the <i>n</i> <sup>th</sup> capture is <b>undefined</b> , use the empty String instead. If <i>n</i> > <i>m</i> , the result is implementation-defined.
<b>\$nn</b>	The <i>nn</i> <sup>th</sup> capture, where <i>nn</i> is a two-digit decimal number in the range 01 to 99. If <i>nn</i> ≤ <i>m</i> and the <i>nn</i> <sup>th</sup> capture is <b>undefined</b> , use the empty String instead. If <i>nn</i> > <i>m</i> , the result is implementation-defined.

NOTE The `replace` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

#### 15.5.4.12 String.prototype.search (regexp)

When the search method is called with argument *regexp*, the following steps are taken:

1. **ReturnIfAbrupt(CheckObjectCoercible(this value)).**
2. Let *string* be the result of calling `ToString`, giving it the `this` value as its argument.
3. **ReturnIfAbrupt(S).**
4. If `Type(regexp)` is Object and *regexp* has a `[[NativeBrand]]` internal property whose value is `NativeRegExp`, then let *rx* be *regexp*;
5. Else, let *rx* be the result of the abstraction operation `RegExpCreate` (15.10.4.1) with arguments *regexp* and **undefined**.
6. **ReturnIfAbrupt(rx).**
7. Search the value *string* from its beginning for an occurrence of the regular expression pattern *rx*. Let *result* be a Number indicating the offset within *string* where the pattern matched, or -1 if there was no match. If an abrupt completion occurs during the search, *result* is that Completion Record. The `lastIndex` and `global` properties of *regexp* are ignored when performing the search. The `lastIndex` property of *regexp* is left unchanged.
8. Return *result*.

NOTE The `search` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

#### 15.5.4.13 String.prototype.slice (start, end)

The `slice` method takes two arguments, *start* and *end*, and returns a substring of the result of converting this object to a String, starting from character position *start* and running to, but not including, character position *end* (or through the end of the String if *end* is **undefined**). If *start* is negative, it is treated as *sourceLength*+*start* where *sourceLength* is the length of the String. If *end* is negative, it is treated as *sourceLength*+*end* where *sourceLength* is the length of the String. The result is a String value, not a String object. The following steps are taken:

1. **ReturnIfAbrupt(CheckObjectCoercible(this value)).**
2. Let *S* be the result of calling `ToString`, giving it the `this` value as its argument.
3. **ReturnIfAbrupt(S).**
4. Let *len* be the number of characters in *S*.
5. Let *intStart* be `ToInteger(start)`.
6. If *end* is **undefined**, let *intEnd* be *len*; else let *intEnd* be `ToInteger(end)`.
7. If *intStart* is negative, let *from* be  $\max(len + intStart, 0)$ ; else let *from* be  $\min(intStart, len)$ .
8. If *intEnd* is negative, let *to* be  $\max(len + intEnd, 0)$ ; else let *to* be  $\min(intEnd, len)$ .
9. Let *span* be  $\max(to - from, 0)$ .
10. Return a String *value* containing *span* consecutive characters from *S* beginning with the character at position *from*.

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

NOTE The `slice` function is intentionally generic; it does not require that its `this` value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

**Deleted:** Call `CheckObjectCoercible` passing the `this` value as its argument.

**Deleted:** the value of the `[[Class]]` internal property of

**Deleted:** the

**Deleted:** `IsRegExp`

**Deleted:** is "RegExp"

**Deleted:** a new `RegExp` object created as if by the expression `new RegExp(regexp)` where `RegExp` is the standard built-in constructor with that name

**Comment [AWB794]:** Needs better spec language

**Deleted:** Call `CheckObjectCoercible` passing the `this` value as its argument.

#### 15.5.4.14 String.prototype.split (separator, limit)

Returns an Array object into which substrings of the result of converting this object to a String have been stored. The substrings are determined by searching from left to right for occurrences of *separator*; these occurrences are not part of any substring in the returned array, but serve to divide up the String value. The value of *separator* may be a String of any length or it may be a RegExp object (i.e., an object with a [[NativeBrand]] internal property whose value is NativeRegExp; see 15.10).

The value of *separator* may be an empty String, an empty regular expression, or a regular expression that can match an empty String. In this case, *separator* does not match the empty substring at the beginning or end of the input String, nor does it match the empty substring at the end of the previous separator match. (For example, if *separator* is the empty String, the String is split up into individual characters; the length of the result array equals the length of the String, and each substring contains one character.) If *separator* is a regular expression, only the first match at a given position of the **this** String is considered, even if backtracking could yield a non-empty-substring match at that position. (For example, "**ab**".split(/**a**\*?/) evaluates to the array ["**a**", "**b**"], while "**ab**".split(/**a**\*/) evaluates to the array ["**a**", "**b**"].)

If the **this** object is (or converts to) the empty String, the result depends on whether *separator* can match the empty String. If it can, the result array contains no elements. Otherwise, the result array contains one element, which is the empty String.

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

`"A<B>bold</B>and<CODE>coded</CODE>".split(/<(\w)/ ? ([^<>]+)>/)`

evaluates to the array

```
["A", undefined, "B", "bold", "/", "B", "and", undefined,  
"CODE", "coded", "/", "CODE", ""]
```

If *separator* is **undefined**, then the result array contains just one String, which is the **this** value (converted to a String). If *limit* is not **undefined**, then the output array is truncated so that it contains no more than *limit* elements.

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

1. ReturnIfAbrupt(CheckObjectCoercible(this value)).
2. Let *S* be the result of calling **ToString**, giving it the **this** value as its argument.
3. ReturnIfAbrupt(*S*).
4. Let *A* be the result of the abstract operation **ArrayCreate** with argument 0.
5. Let *lengthA* be 0.
6. If *limit* is **undefined**, let *lim* =  $2^{32}-1$ ; else let *lim* = **ToInt32**(*limit*).
7. Let *s* be the number of characters in *S*.
8. Let *p* = 0.
9. If *separator* has a [[NativeBrand]] internal property whose value is NativeRegExp, let *R* be *separator*; otherwise let *R* be **ToString(separator)**.
10. ReturnIfAbrupt(*separator*).
11. If *lim* = 0, return *A*.
12. If *separator* is **undefined**, then
  - a. Call the [[DefineOwnProperty]] internal method of *A* with arguments "**0**", Property Descriptor {[[Value]]: *S*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
  - b. Assert: the previous step will never result in an abrupt completion.
  - c. Return *A*.
13. If *s* = 0, then
  - a. Call **SplitMatch**(*S*, 0, *R*) and let *z* be its MatchResult result.
  - b. If *z* is not **failure**, return *A*.
  - c. Call the [[DefineOwnProperty]] internal method of *A* with arguments "**0**", Property Descriptor {[[Value]]: *S*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
  - d. Assert: the previous step will never result in an abrupt completion.

**Deleted:** the

**Deleted:** IsRegExp

**Deleted:** whose [[Class]] internal property is "RegExp"

- e. Return  $A$ .
- 14. Let  $q = p$ .
- 15. Repeat, while  $q \neq s$ 
  - a. Call  $\text{SplitMatch}(S, q, R)$  and let  $z$  be its MatchResult result.
  - b. If  $z$  is **failure**, then let  $q = q+1$ .
  - c. Else,  $z$  is not **failure**
    - i.  $z$  must be a State. Let  $e$  be  $z$ 's *endIndex* and let  $cap$  be  $z$ 's *captures* array.
    - ii. If  $e = p$ , then let  $q = q+1$ .
    - iii. Else,  $e \neq p$ 
      1. Let  $T$  be a String value equal to the substring of  $S$  consisting of the characters at positions  $p$  (inclusive) through  $q$  (exclusive).
      2. Call the **[[DefineOwnProperty]]** internal method of  $A$  with arguments  $\text{ToString}(lengthA)$ , Property Descriptor **{[[Value]]:  $T$ , [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}**, and **false**.
      - 3. Assert: the previous step will never result in an abrupt completion.**
      4. Increment  $lengthA$  by 1.
      5. If  $lengthA = lim$ , return  $A$ .
      6. Let  $p = e$ .
      7. Let  $i = 0$ .
      8. Repeat, while  $i$  is not equal to the number of elements in  $cap$ .
        - a. Let  $i = i+1$ .
        - b. Call the **[[DefineOwnProperty]]** internal method of  $A$  with arguments  $\text{ToString}(lengthA)$ , Property Descriptor **{[[Value]]:  $cap[i]$ , [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}**, and **false**.
        - c. Assert: the previous step will never result in an abrupt completion.**
        - d. Increment  $lengthA$  by 1.
        - e. If  $lengthA = lim$ , return  $A$ .
      9. Let  $q = p$ .
  16. Let  $T$  be a String value equal to the substring of  $S$  consisting of the characters at positions  $p$  (inclusive) through  $s$  (exclusive).
  17. Call the **[[DefineOwnProperty]]** internal method of  $A$  with arguments  $\text{ToString}(lengthA)$ , Property Descriptor **{[[Value]]:  $T$ , [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}**, and **false**.
  - 18. Assert: the previous step will never result in an abrupt completion.**
  19. Return  $A$ .

**Formatted:** Don't add space between paragraphs of the same style

The abstract operation *SplitMatch* takes three parameters, a String  $S$ , an integer  $q$ , and a String or RegExp  $R$ , and performs the following in order to return a MatchResult (see 15.10.2.1):

1. If  $R$  has a **[[NativeBrand]]** internal property whose value is NativeRegExp, then
  - a. Call the **[[Match]]** internal method of  $R$  giving it the arguments  $S$  and  $q$ , and return the MatchResult result.
2. Type( $R$ ) must be String. Let  $r$  be the number of characters in  $R$ .
3. Let  $s$  be the number of characters in  $S$ .
4. If  $q+r > s$  then return the MatchResult **failure**.
5. If there exists an integer  $i$  between 0 (inclusive) and  $r$  (exclusive) such that the character at position  $q+i$  of  $S$  is different from the character at position  $i$  of  $R$ , then return **failure**.
6. Let  $cap$  be an empty array of captures (see 15.10.2.1).
7. Return the State  $(q+r, cap)$ . (see 15.10.2.1)

**Deleted:** the

**Deleted:** IsRegExp

**Deleted:** is a RegExp object (its **[[Class]]** is "RegExp")

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

NOTE 1 The **split** method ignores the value of separator **.global** for separators that are RegExp objects.

NOTE 2 The **split** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

#### 15.5.4.15 String.prototype.substring (start, end)

The **substring** method takes two arguments, *start* and *end*, and returns a substring of the result of converting this object to a String, starting from character position *start* and running to, but not including, character position

**Comment [AWB795]:** TODO: Probably needs some Completion Record work, assume that **[[Match]]** can fail

*end* of the String (or through the end of the String if *end* is **undefined**). The result is a String value, not a String object.

If either argument is **NaN** or negative, it is replaced with zero; if either argument is larger than the length of the String, it is replaced with the length of the String.

If *start* is larger than *end*, they are swapped.

The following steps are taken:

1. ReturnIfAbrupt(CheckObjectCoercible(this value)).
2. Let *S* be the result of calling `ToString`, giving it the **this** value as its argument.
3. ReturnIfAbrupt(*S*).
4. Let *len* be the number of characters in *S*.
5. Let *intStart* be `ToInteger(start)`.
6. If *end* is **undefined**, let *intEnd* be *len*; else let *intEnd* be `ToInteger(end)`.
7. Let *finalStart* be `min(max(intStart, 0), len)`.
8. Let *finalEnd* be `min(max(intEnd, 0), len)`.
9. Let *from* be `min(finalStart, finalEnd)`.
10. Let *to* be `max(finalStart, finalEnd)`.
11. Return a String whose length is *to* - *from*, containing characters from *S*, namely the characters with indices *from* through *to* - 1, in ascending order.

**Deleted:** Call `CheckObjectCoercible` passing the **this** value as its argument.

The `length` property of the `substring` method is **2**.

**NOTE** The `substring` function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

#### 15.5.4.16 `String.prototype.toLowerCase ()`

The following steps are taken:

1. ReturnIfAbrupt(CheckObjectCoercible(this value)).
2. Let *S* be the result of calling `ToString`, giving it the **this** value as its argument.
3. ReturnIfAbrupt(*S*).
4. Let *L* be a String where each character of *L* is either the Unicode lowercase equivalent of the corresponding character of *S* or the actual corresponding character of *S* if no Unicode lowercase equivalent exists.
5. Return *L*.

**Deleted:** Call `CheckObjectCoercible` passing the **this** value as its argument.

For the purposes of this operation, the 16-bit code units of the Strings are treated as code points in the Unicode Basic Multilingual Plane. Surrogate code points are directly transferred from *S* to *L* without any mapping.

**NOTE 1** The case mapping of some characters may produce multiple characters. In this case the result String may not be the same length as the source String. Because both `toUpperCase` and `toLowerCase` have context-sensitive behaviour, the functions are not symmetrical. In other words, `s.toUpperCase().toLowerCase()` is not necessarily equal to `s.toLowerCase().toUpperCase()`.

**NOTE 2** The `toLowerCase` function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

#### 15.5.4.17 `String.prototype.toLocaleLowerCase ()`

This function works exactly the same as `toLowerCase` except that its result is intended to yield the correct result for the host environment's current locale, rather than a locale-independent result. There will only be a

difference in the few cases (such as Turkish) where the rules for that language conflict with the regular Unicode case mappings.

NOTE 1 The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

NOTE 2 The `toLocaleLowerCase` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

#### **15.5.4.18 String.prototype.toUpperCase ( )**

This function behaves in exactly the same way as `String.prototype.toLowerCase`, except that characters are mapped to their *uppercase* equivalents as specified in the Unicode Character Database.

NOTE The `toUpperCase` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

#### **15.5.4.19 String.prototype.toLocaleUpperCase ( )**

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

NOTE 1 The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

NOTE 2 The `toLocaleUpperCase` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

#### **15.5.4.20 String.prototype.trim ( )**

The following steps are taken:

1. ReturnIfAbrupt(CheckObjectCoercible(this value)).
2. Let  $S$  be the result of calling `ToString`, giving it the `this` value as its argument.
3. ReturnIfAbrupt( $S$ ).
4. Let  $T$  be a String value that is a copy of  $S$  with both leading and trailing white space removed. The definition of white space is the union of `WhiteSpace` and `LineTerminator`.
5. Return  $T$ .

**Deleted:** Call `CheckObjectCoercible` passing the `this` value as its argument.

NOTE The `trim` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

#### **15.5.4.21 String.prototype.repeat (count)**

The following steps are taken:

1. ReturnIfAbrupt(CheckObjectCoercible(this value)).
2. Let  $S$  be the result of calling `ToString`, giving it the `this` value as its argument.
3. ReturnIfAbrupt( $S$ ).
4. Let  $n$  be the result of calling `ToInteger(count)`.
5. ReturnIfAbrupt( $n$ ).
6. If  $n < 0$ , then throw a `RangeError` exception.
7. Let  $T$  be a String value that is made from  $n$  copies of  $S$  appended together.
8. Return  $T$ .

**Comment [AWB796]:** Is this the right error handling?

NOTE 1 This method creates a String consisting of the characters of this object (converted to String) repeated  $count$  time.

NOTE 2 The `repeat` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

#### 15.5.4.22 String.prototype.startsWith (searchString [, position])

The following steps are taken:

1. `ReturnIfAbrupt(CheckObjectCoercible(this value))`.
2. Let  $S$  be the result of calling `ToString`, giving it the `this` value as its argument.
3. `ReturnIfAbrupt(S)`.
4. Let  $searchStr$  be `ToString(searchString)`.
5. `ReturnIfAbrupt(searchStr)`.
6. Let  $pos$  be `ToInteger(position)`. (If `position` is `undefined`, this step produces the value `0`).
7. `ReturnIfAbrupt(pos)`.
8. Let  $len$  be the number of characters in  $S$ .
9. Let  $start$  be  $\min(\max(pos, 0), len)$ .
10. Let  $searchLength$  be the number of characters in `searchString`.
11. If  $searchLength + start$  is greater than  $len$ , return `false`.
12. If the `searchLength` sequence of characters of  $S$  starting at  $start$  is the same as the full character sequence of `searchString`, return `true`.
13. Otherwise, return `false`.

The `length` property of the `startsWith` method is `1`.

NOTE 1 This method returns `true` if the sequence of characters of `searchString` converted to a String is the same as the corresponding characters of this object (converted to a String) starting at `position`. Otherwise returns `false`.

NOTE 2 The `startsWith` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

#### 15.5.4.23 String.prototype.endsWith (searchString [, endPosition])

The following steps are taken:

1. `ReturnIfAbrupt(CheckObjectCoercible(this value))`.
2. Let  $S$  be the result of calling `ToString`, giving it the `this` value as its argument.
3. `ReturnIfAbrupt(S)`.
4. Let  $searchStr$  be `ToString(searchString)`.
5. `ReturnIfAbrupt(searchStr)`.
6. Let  $len$  be the number of characters in  $S$ .
7. If `endPosition` is `undefined`, let  $pos$  be  $len$ , else let  $pos$  be `ToInteger(endPosition)`.
8. `ReturnIfAbrupt(pos)`.
9. Let  $end$  be  $\min(\max(pos, 0), len)$ .
10. Let  $searchLength$  be the number of characters in `searchString`.
11. Let  $start$  be  $end - searchLength$ .
12. If  $start$  is less than `0`, return `false`.
13. If the `searchLength` sequence of characters of  $S$  starting at  $start$  is the same as the full character sequence of `searchString`, return `true`.
14. Otherwise, return `false`.

The `length` property of the `endsWith` method is `1`.

NOTE 1 Returns `true` if the sequence of characters of `searchString` converted to a String is the same as the corresponding characters of this object (converted to a String) starting at `endPosition - length(this)`. Otherwise returns `false`.

NOTE 2 The `endsWith` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

#### 15.5.4.24 String.prototype.contains (searchString [, position])

The `contains` method takes two arguments, `searchString` and `position`, and performs the following steps:

1. `ReturnIfAbrupt(CheckObjectCoercible(this value)).`
2. Let `S` be the result of calling `ToString`, giving it the `this` value as its argument.
3. `ReturnIfAbrupt(S).`
4. Let `searchStr` be `ToString(searchString)`.
5. `ReturnIfAbrupt(searchStr).`
6. Let `pos` be `ToInteger(position)`. (If `position` is `undefined`, this step produces the value `0`).
7. `ReturnIfAbrupt(pos).`
8. Let `len` be the number of characters in `S`.
9. Let `start` be `min(max(pos, 0), len)`.
10. Let `searchLen` be the number of characters in `searchStr`.
11. If there exists any integer `k` not smaller than `start` such that `k + searchLen` is not greater than `len`, and for all nonnegative integers `j` less than `searchLen`, the character at position `k+j` of `S` is the same as the character at position `j` of `searchStr`, return `true`; but if there is no such integer `k`, return `false`.

The `length` property of the `contains` method is `1`.

NOTE 1 If `searchString` appears as a substring of the result of converting this object to a String, at one or more positions that are greater than or equal to `position`, then return `true`; otherwise, returns `false`. If `position` is `undefined`, `0` is assumed, so as to search all of the String.

NOTE 2 The `contains` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

#### 15.5.4.25 String.prototype.toArray()

The following steps are taken:

1. `ReturnIfAbrupt(CheckObjectCoercible(this value)).`
2. Let `S` be the result of calling `ToString`, giving it the `this` value as its argument.
3. `ReturnIfAbrupt(S).`
4. Let `len` be the number of characters in `S`.
5. Let `array` be the result of the abstraction operation `ArrayCreate` (15.4) with argument `len`.
6. Let `n` be `0`
7. Repeat, while `n < len`:
  - a. Let `c` be the character at position `n` in `S`.
  - b. Call the `[[DefineOwnProperty]]` internal method of `array` with arguments `ToString(n)`, the `PropertyDescriptor` `{[[Value]]: c, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, and `false`.
  - c. Increment `n` by 1.
8. Return `array`.

The `length` property of the `toArray` method is `0`.

NOTE 1 Returns an Array object with elements corresponding to the characters of this object (converted to a String).

NOTE 2 The `toArray` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

#### **15.5.5 Properties of String Instances**

String instances inherit properties from the String prototype object and have a `[[NativeBrand]]` internal property with value `StringWrapper`. String instances also have a `[[PrimitiveValue]]` internal property, a `length` property, and a set of enumerable properties with array index names.

**Deleted:** their `[[Class]]`

**Deleted:** `IsString`

**Deleted:** value is "String"

The `[[PrimitiveValue]]` internal property is the String value represented by this String object. The array index named properties correspond to the individual characters of the String value. A special `[[GetOwnProperty]]` internal method is used to specify the number, values, and attributes of the array index named properties.

#### 15.5.5.1 `length`

The number of characters in the String value represented by this String object.

Once a String object is created, this property is unchanging. It has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }`.

#### 15.5.5.2 `[[GetOwnProperty]](P)`

String 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 characters of String objects.

Assume `S` is a String object and `P` is a String.

When the `[[GetOwnProperty]]` internal method of `S` 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 `S` with argument `P`.
2. If `desc` is not `undefined` return `desc`.
3. If `ToString(abs(TolInteger(P)))` is not the same value as `P`, return `undefined`.
4. Let `str` be the String value of the `[[PrimitiveValue]]` internal property of `S`.
5. Let `index` be `TolInteger(P)`.
6. Let `len` be the number of characters in `str`.
7. If `len ≤ index`, return `undefined`.
8. Let `resultStr` be a String of length 1, containing one character from `str`, specifically the character at position `index`, where the first (leftmost) character in `str` is considered to be at position 0, the next one at position 1, and so on.
9. Return a Property Descriptor `{ [[Value]]: resultStr, [[Enumerable]]: true, [[Writable]]: false, [[Configurable]]: false }`.

### 15.6 Boolean Objects

#### 15.6.1 The Boolean Constructor Called as a Function

When `Boolean` is called as a function rather than as a constructor, it performs a type conversion.

##### 15.6.1.1 `Boolean(value)`

Returns a Boolean value (not a Boolean object) computed by `ToBoolean(value)`.

#### 15.6.2 The Boolean Constructor

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

##### 15.6.2.1 `new Boolean(value)`

The `[[Prototype]]` internal property of the newly constructed object is set to the original Boolean prototype object, the one that is the initial value of `Boolean.prototype` (15.6.3.1).

The newly constructed Boolean object has a `[[NativeBrand]]` internal property with value `BooleanWrapper`.

The `[[PrimitiveValue]]` internal property of the newly constructed Boolean object is set to `ToBoolean(value)`.

**Deleted:** `[[Class]]` internal property of the  
**Deleted:** is set to "Boolean"  
**Deleted:** `IsBoolean`

The `[[Extensible]]` internal property of the newly constructed object is set to `true`.

### 15.6.3 Properties of the Boolean Constructor

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

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

#### 15.6.3.1 Boolean.prototype

The initial value of `Boolean.prototype` is the Boolean prototype object (15.6.4).

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

### 15.6.4 Properties of the Boolean Prototype Object

The Boolean prototype object is itself a Boolean object whose value is `false`. [The Boolean prototype object has a `\[\[NativeBrand\]\]` internal property whose value is BooleanWrapper.](#)

**Deleted:** (its `[[Class]]` is "Boolean")

**Deleted:** IsBoolean

The value of the `[[Prototype]]` internal property of the Boolean prototype object is the standard built-in Object prototype object (15.2.4).

#### 15.6.4.1 Boolean.prototype.constructor

The initial value of `Boolean.prototype.constructor` is the built-in `Boolean` constructor.

#### 15.6.4.2 Boolean.prototype.toString ()

The following steps are taken:

1. Let  $B$  be the `this` value.
2. If `Type(B)` is Boolean, then let  $b$  be  $B$ .
3. Else if `Type(B)` is Object and  $B$  has a `[[NativeWrapper]]` internal property whose value is BooleanWrapper, then let  $b$  be the value of the `[[PrimitiveValue]]` internal property of  $B$ .
4. Else throw a `TypeError` exception.
5. If  $b$  is `true`, then return "`true`"; else return "`false`".

**Deleted:** the value of the `[[Class]]` internal property of

**Deleted:** is "Boolean"

**Deleted:** IsBoolean

#### 15.6.4.3 Boolean.prototype.valueOf ()

The following steps are taken:

1. Let  $B$  be the `this` value.
2. If `Type(B)` is Boolean, then let  $b$  be  $B$ .
3. Else if `Type(B)` is Object and  $B$  has a `[[NativeWrapper]]` internal property whose value is BooleanWrapper, then let  $b$  be the value of the `[[PrimitiveValue]]` internal property of  $B$ .
4. Else throw a `TypeError` exception.
5. Return  $b$ .

**Deleted:** the value of the `[[Class]]` internal property of

**Deleted:** is "Boolean"

**Deleted:** IsBoolean

### 15.6.5 Properties of Boolean Instances

Boolean instances inherit properties from the Boolean prototype object and [have a `\[\[NativeBrand\]\]` internal property whose value is BooleanWrapper](#). Boolean instances also have a `[[PrimitiveValue]]` internal property.

**Deleted:** their

**Deleted:** Class

**Deleted:** IsBoolean

**Deleted:** value is "Boolean"

## 15.7 Number Objects

### 15.7.1 The Number Constructor Called as a Function

When `Number` is called as a function rather than as a constructor, it performs a type conversion.

#### 15.7.1.1 `Number( [ value ] )`

Returns a Number value (not a Number object) computed by `ToNumber(value)` if `value` was supplied, else returns `+0`.

### 15.7.2 The Number Constructor

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

#### 15.7.2.1 `new Number( [ value ] )`

The `[[Prototype]]` internal property of the newly constructed object is set to the original Number prototype object, the one that is the initial value of `Number.prototype` (15.7.3.1).

The newly constructed object is has a `[[NativeBrand]]` internal property whose value is `NumberWrapper`.

The `[[PrimitiveValue]]` internal property of the newly constructed object is set to `ToNumber(value)` if `value` was supplied, else to `+0`.

The `[[Extensible]]` internal property of the newly constructed object is set to `true`.

**Deleted:** `[[Class]]` internal property of the  
**Deleted:** set to "Number"  
**Deleted:** `IsNumber`

### 15.7.3 Properties of the Number Constructor

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

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

#### 15.7.3.1 `Number.prototype`

The initial value of `Number.prototype` is the Number prototype object (15.7.4).

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

#### 15.7.3.2 `Number.MAX_VALUE`

The value of `Number.MAX_VALUE` is the largest positive finite value of the Number type, which is approximately  $1.7976931348623157 \times 10^{308}$ .

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

#### 15.7.3.3 `Number.MIN_VALUE`

The value of `Number.MIN_VALUE` is the smallest positive value of the Number type, which is approximately  $5 \times 10^{-324}$ .

In the IEEE-764 double precision binary representation, the smallest possible value is a denormalized number. If an implementation does not support denormalized values, the value of `Number.MIN_VALUE` must be the smallest non-zero positive value that can actually be represented by the implementation.

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

#### 15.7.3.4 Number.NaN

The value of `Number.NaN` is **NaN**.

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

#### 15.7.3.5 Number.NEGATIVE\_INFINITY

The value of `Number.NEGATIVE_INFINITY` is  $-\infty$ .

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

#### 15.7.3.6 Number.POSITIVE\_INFINITY

The value of `Number.POSITIVE_INFINITY` is  $+\infty$ .

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

#### 15.7.3.7 Number.EPSILON

The value of `Number.EPSILON` is the difference between 1 and the smallest value greater than 1 that is representable as a `Number` value, which is approximately  $2.2204460492503130808472633361816 \times 10^{-16}$ .

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

#### 15.7.3.7 Number.MAX\_INTEGER

The value of `Number.MAX_INTEGER` is the largest integer value that can be represented as a `Number` value without losing precision, which is 9007199254740991.

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

#### 15.7.3.8 Number.parseInt (string, radix)

Same as 15.1.2.2.

**Comment [AWB797]:** Same specification or same value?

#### 15.7.3.9 Number.parseFloat (string)

Same as 15.1.2.3.

**Comment [AWB798]:** Same specification or same value?

#### 15.7.3.10 Number.isNaN (number)

When the `Number.isNaN` is called with one argument `number`, the following steps are taken:

1. If `Type(number)` is not `Number`, return `false`.
2. If `number` is `NaN`, return `true`.
3. Otherwise, return `false`.

**NOTE** This function differs from the global `isNaN` function (15.1.2.4) is that it does not convert its argument to a `Number` before determining whether it is `NaN`.

#### 15.7.3.11 Number.isFinite (number)

When the `Number.isFinite` is called with one argument `number`, the following steps are taken:

1. If `Type(number)` is not `Number`, return `false`.
2. If `number` is `NaN`,  $+\infty$ , or  $-\infty$ , return `false`.
3. Otherwise, return `true`.

### 15.7.3.12 Number.isInteger (number)

When the `Number.isInteger` is called with one argument `number`, the following steps are taken:

1. If `Type(number)` is not `Number`, return `false`.
2. Let `integer` be `ToInteger(number)`.
3. If `integer` is not equal to `number`, return `false`.
4. Otherwise, return `true`.

### 15.7.3.13 Number.toInt (number)

When the `Number.toInt` is called with one argument `number`, the following steps are taken:

1. Return `ToInteger(number)`.

**Comment [AWB799]:** Name changed from `toInteger` at Marcdh 29 TC39 meeting

## 15.7.4 Properties of the Number Prototype Object

The Number prototype object is itself a Number object with a `[[NativeBrand]]` internal property whose value is `NumberWrapper`. Its value is `+0`.

**Deleted:** (its `[[Class]]` is "Number")  
**Deleted:** IsNumber  
**Deleted:** whose

The value of the `[[Prototype]]` internal property of the Number prototype object is the standard built-in Object prototype object (15.2.4).

**Deleted:** Object  
**Deleted:** for which the value of  
**Deleted:** the  
**Deleted:** Class  
**Deleted:** IsNumber  
**Deleted:** is "Number"

Unless explicitly stated otherwise, the methods of the Number prototype object defined below are not generic and the `this` value passed to them must be either a Number value or an `object that has a [[NativeBrand]] internal property whose value is NumberWrapper`.

**Deleted:** for which the value of the  
**Deleted:** n  
**Deleted:** Class  
**Deleted:** IsNumber  
**Deleted:** is "Number"

In the following descriptions of functions that are properties of the Number prototype object, the phrase "this Number object" refers to either the object that is the `this` value for the invocation of the function or, if `Type(this value)` is `Number`, an object that is created as if by the expression `new Number(this value)` where `Number` is the standard built-in constructor with that name. Also, the phrase "this Number value" refers to either the Number value represented by this Number object, that is, the value of the `[[PrimitiveValue]]` internal property of this Number object or the `this` value if its type is Number. A `TypeError` exception is thrown if the `this` value is neither an object `that has a [[NativeBrand]] internal property whose value is NumberWrapper` or a value whose type is Number.

**Deleted:** for which the value of the  
**Deleted:** n  
**Deleted:** Class  
**Deleted:** IsNumber  
**Deleted:** is "Number"

### 15.7.4.1 Number.prototype.constructor

The initial value of `Number.prototype.constructor` is the built-in `Number` constructor.

**Comment [AWB7100]:** TODO: need to private algorithm that ordered abnormal completion detection

### 15.7.4.2 Number.prototype.toString ( [ radix ] )

The optional `radix` should be an integer value in the inclusive range 2 to 36. If `radix` not present or is `undefined` the Number 10 is used as the value of `radix`. If `ToInteger(radix)` is the Number 10 then this Number value is given as an argument to the `ToString` abstract operation; the resulting String value is returned.

If `ToInteger(radix)` is not an integer between 2 and 36 inclusive throw a `RangeError` exception. If `ToInteger(radix)` is an integer from 2 to 36, but not 10, the result is a String representation of this Number value using the specified radix. Letters `a-z` are used for digits with values 10 through 35. The precise algorithm is implementation-dependent if the radix is not 10, however the algorithm should be a generalisation of that specified in 9.8.1.

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

#### 15.7.4.3 Number.prototype.toLocaleString()

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

**NOTE** The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

#### 15.7.4.4 Number.prototype.valueOf()

1. Let  $x$  be this Number value.
2. Return  $x$ .

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

**Formatted:** Space After: 11 pt

**Deleted:** Returns this Number value.¶

#### 15.7.4.5 Number.prototype.toFixed (fractionDigits)

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

3. Let  $f$  be `ToInteger(fractionDigits)`. (If *fractionDigits* is **undefined**, this step produces the value 0).
4. ReturnIfAbrupt(f).
5. If  $f < 0$  or  $f > 20$ , throw a **RangeError** exception.
6. Let  $x$  be this Number value.
7. ReturnIfAbrupt(x).
8. If  $x$  is **NaN**, return the String "**NaN**".
9. Let  $s$  be the empty String.
10. If  $x < 0$ , then
  - a. Let  $s$  be "-".
  - b. Let  $x = -x$ .
11. If  $x \geq 10^{21}$ , then
  - a. Let  $m = \text{ToString}(x)$ .
12. Else,  $x < 10^{21}$ 
  - a. Let  $n$  be an integer for which the exact mathematical value of  $n \div 10^f - x$  is as close to zero as possible. If there are two such  $n$ , pick the larger  $n$ .
  - b. If  $n = 0$ , let  $m$  be the String "**0**". Otherwise, let  $m$  be the String consisting of the digits of the decimal representation of  $n$  (in order, with no leading zeroes).
  - c. If  $f \neq 0$ , then
    - i. Let  $k$  be the number of characters in  $m$ .
    - ii. If  $k \leq f$ , then
      1. Let  $z$  be the String consisting of  $f+1-k$  occurrences of the character '**0**'.
      2. Let  $m$  be the concatenation of Strings  $z$  and  $m$ .
      3. Let  $k = f + 1$ .
    - iii. Let  $a$  be the first  $k-f$  characters of  $m$ , and let  $b$  be the remaining  $f$  characters of  $m$ .
    - iv. Let  $m$  be the concatenation of the three Strings  $a$ , ". ", and  $b$ .
13. Return the concatenation of the Strings  $s$  and  $m$ .

The `length` property of the `toFixed` method is **1**.

If the `toFixed` method is called with more than one argument, then the behaviour is undefined (see clause 15).

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

**NOTE** The output of `toFixed` may be more precise than `toString` for some values because `toString` only prints enough significant digits to distinguish the number from adjacent number values. For example, `(1000000000000000128).toString()` returns "1000000000000000100", while `(1000000000000000128).toFixed(0)` returns "1000000000000000128".

#### 15.7.4.6 Number.prototype.toExponential (fractionDigits)

Return a String containing this Number value represented in decimal exponential notation with one digit before the significand's decimal point and *fractionDigits* digits after the significand's decimal point. If *fractionDigits* is **undefined**, include as many significand digits as necessary to uniquely specify the Number (just like in `ToString` except that in this case the Number is always output in exponential notation). Specifically, perform the following steps:

1. Let *x* be this Number value.
2. ReturnIfAbrupt(*x*).
3. Let *f* be `TolInteger(fractionDigits)`.
4. ReturnIfAbrupt(*f*).
5. If *x* is `NaN`, return the String "`NaN`".
6. Let *s* be the empty String.
7. If *x* < 0, then
  - a. Let *s* be `"-"`.
  - b. Let *x* = *-x*.
8. If *x* =  $+\infty$ , then
  - a. Return the concatenation of the Strings *s* and "`Infinity`".
9. If *fractionDigits* is not **undefined** and (*f* < 0 or *f* > 20), throw a **RangeError** exception.
10. If *x* = 0, then
  - a. If *fractionalDigits* is **undefined**, then let *f* = 0.
  - b. Let *m* be the String consisting of *f*+1 occurrences of the character '`0`'.
  - c. Let *e* = 0.
11. Else, *x* ≠ 0
  - a. If *fractionDigits* is not **undefined**, then
    - i. Let *e* and *n* be integers such that  $10^f \leq n < 10^{f+1}$  and for which the exact mathematical value of  $n \times 10^{e-f} - x$  is as close to zero as possible. If there are two such sets of *e* and *n*, pick the *e* and *n* for which  $n \times 10^{e-f}$  is larger.
  - b. Else, *fractionDigits* is **undefined**
    - i. Let *e*, *n*, and *f* be integers such that *f* ≥ 0,  $10^f \leq n < 10^{f+1}$ , the number value for  $n \times 10^{e-f}$  is *x*, and *f* is as small as possible. Note that the decimal representation of *n* has *f*+1 digits, *n* is not divisible by 10, and the least significant digit of *n* is not necessarily uniquely determined by these criteria.
  - c. Let *m* be the String consisting of the digits of the decimal representation of *n* (in order, with no leading zeroes).
12. If *f* ≠ 0, then
  - a. Let *a* be the first character of *m*, and let *b* be the remaining *f* characters of *m*.
  - b. Let *m* be the concatenation of the three Strings *a*, `"."`, and *b*.
13. If *e* = 0, then
  - a. Let *c* = `"+"`.
  - b. Let *d* = `"0"`.
14. Else
  - a. If *e* > 0, then let *c* = `"+"`.
  - b. Else, *e* ≤ 0
    - i. Let *c* = `"-"`.
    - ii. Let *e* = *-e*.
  - c. Let *d* be the String consisting of the digits of the decimal representation of *e* (in order, with no leading zeroes).
15. Let *m* be the concatenation of the four Strings *m*, `"e"`, *c*, and *d*.
16. Return the concatenation of the Strings *s* and *m*.

**Deleted:** Let

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

If the `toExponential` method is called with more than one argument, then the behaviour is undefined (see clause 15).

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

**NOTE** For implementations that provide more accurate conversions than required by the rules above, it is recommended that the following alternative version of step 9.b.i be used as a guideline:

- i. Let  $e$ ,  $n$ , and  $f$  be integers such that  $f \geq 0$ ,  $10^f \leq n < 10^{f+1}$ , the number value for  $n \times 10^{e-f}$  is  $x$ , and  $f$  is as small as possible. If there are multiple possibilities for  $n$ , choose the value of  $n$  for which  $n \times 10^{e-f}$  is closest in value to  $x$ . If there are two such possible values of  $n$ , choose the one that is even.

#### 15.7.4.7 Number.prototype.toPrecision (precision)

Return a String containing this Number value represented either in decimal exponential notation with one digit before the significand's decimal point and `precision`-1 digits after the significand's decimal point or in decimal fixed notation with `precision` significant digits. If `precision` is `undefined`, call `ToString` (9.8.1) instead. Specifically, perform the following steps:

1. Let  $x$  be this Number value.
2. ReturnIfAbrupt( $x$ ).
3. If `precision` is `undefined`, return `ToString( $x$ )`.
4. Let  $p$  be `ToInteger(precision).`
5. ReturnIfAbrupt( $p$ ).
6. If  $x$  is `NaN`, return the String "`NaN`".
7. Let  $s$  be the empty String.
8. If  $x < 0$ , then
  - a. Let  $s$  be `"-"`.
  - b. Let  $x = -x$ .
9. If  $x = +\infty$ , then
  - a. Return the concatenation of the Strings  $s$  and "`Infinity`".
10. If  $p < 1$  or  $p > 21$ , throw a `RangeError` exception.
11. If  $x = 0$ , then
  - a. Let  $m$  be the String consisting of  $p$  occurrences of the character '`0`'.
  - b. Let  $e = 0$ .
12. Else  $x \neq 0$ ,
  - a. Let  $e$  and  $n$  be integers such that  $10^{p-1} \leq n < 10^p$  and for which the exact mathematical value of  $n \times 10^{e-p+1} - x$  is as close to zero as possible. If there are two such sets of  $e$  and  $n$ , pick the  $e$  and  $n$  for which  $n \times 10^{e-p+1}$  is larger.
  - b. Let  $m$  be the String consisting of the digits of the decimal representation of  $n$  (in order, with no leading zeroes).
  - c. If  $e < -6$  or  $e \geq p$ , then
    - i. Let  $a$  be the first character of  $m$ , and let  $b$  be the remaining  $p-1$  characters of  $m$ .
    - ii. Let  $m$  be the concatenation of the three Strings  $a$ , `"."`, and  $b$ .
    - iii. If  $e = 0$ , then
      1. Let  $c = "+"$  and  $d = "0"$ .
    - iv. Else  $e \neq 0$ ,
      1. If  $e > 0$ , then
        - a. Let  $c = "+"$ .
      2. Else  $e < 0$ ,
        - a. Let  $c = "-"$ .
        - b. Let  $e = -e$ .
      3. Let  $d$  be the String consisting of the digits of the decimal representation of  $e$  (in order, with no leading zeroes).
    - v. Let  $m$  be the concatenation of the five Strings  $s$ ,  $m$ , `"e"`,  $c$ , and  $d$ .
13. If  $e = p-1$ , then return the concatenation of the Strings  $s$  and  $m$ .
14. If  $e \geq 0$ , then

- a. Let  $m$  be the concatenation of the first  $e+1$  characters of  $m$ , the character ‘.’, and the remaining  $p-(e+1)$  characters of  $m$ .

15. Else  $e < 0$ ,
- Let  $m$  be the concatenation of the String "0.",  $-(e+1)$  occurrences of the character '0', and the String  $m$ .

16. Return the concatenation of the Strings  $s$  and  $m$ .

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

If the `toPrecision` method is called with more than one argument, then the behaviour is undefined (see clause 15).

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

#### 15.7.3.14 Number.prototype.clz()

When the `Number.prototype.clz` is called with one argument *number*, the following steps are taken:

1. Let  $x$  be this `Number` value.
2. Let  $n$  be `ToUint32`( $x$ ).
3. ReturnIfAbrupt( $n$ ).
4. Let  $p$  be the number of leading zero bits in the 32-bit binary representation of  $n$ .
5. Return  $p$ .

NOTE If  $n$  is 0,  $p$  will be 32. If the most significant bit the 32-bit binary encoding of  $n$  is 1,  $p$  will be 0.

#### 15.7.5 Properties of Number Instances

Number instances inherit properties from the `Number` prototype object and have a `[[NativeBrand]]` internal property whose value is `NumberWrapper`. Number instances also have a `[[PrimitiveValue]]` internal property.

The `[[PrimitiveValue]]` internal property is the `Number` value represented by this `Number` object.

### 15.8 The Math Object

The `Math` object is a single object that has some named properties, some of which are functions.

The value of the `[[Prototype]]` internal property of the `Math` object is the standard built-in `Object` prototype object (15.2.4). The `Math` object has a `[[NativeBrand]]` internal property whose value is `NativeMath`.

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

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

NOTE In this specification, the phrase “the `Number` value for  $x$ ” has a technical meaning defined in 8.5.

#### 15.8.1 Value Properties of the Math Object

##### 15.8.1.1 E

The `Number` value for  $e$ , the base of the natural logarithms, which is approximately 2.7182818284590452354.

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

**Comment [AWB7101]:** Added at March 29 TC39 meeting

**Comment [AWB7102]:** Unsigned seems most general, signed values could be converted to in a previous step.

**Deleted:** their  
**Deleted:** n  
**Deleted:** Class  
**Deleted:** IsNumber  
**Deleted:** value is "Number"

**Deleted:** value of the [[Class]] internal property of the  
**Deleted:** is "Math"  
**Deleted:** the  
**Deleted:** IsMath

#### 15.8.1.2 LN10

The Number value for the natural logarithm of 10, which is approximately 2.302585092994046.

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

#### 15.8.1.3 LN2

The Number value for the natural logarithm of 2, which is approximately 0.6931471805599453.

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

#### 15.8.1.4 LOG2E

The Number value for the base-2 logarithm of  $e$ , the base of the natural logarithms; this value is approximately 1.4426950408889634.

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

NOTE The value of `Math.LOG2E` is approximately the reciprocal of the value of `Math.LN2`.

#### 15.8.1.5 LOG10E

The Number value for the base-10 logarithm of  $e$ , the base of the natural logarithms; this value is approximately 0.4342944819032518.

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

NOTE The value of `Math.LOG10E` is approximately the reciprocal of the value of `Math.LN10`.

#### 15.8.1.6 PI

The Number value for  $\pi$ , the ratio of the circumference of a circle to its diameter, which is approximately 3.1415926535897932.

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

#### 15.8.1.7 SQRT1\_2

The Number value for the square root of  $\frac{1}{2}$ , which is approximately 0.7071067811865476.

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

NOTE The value of `Math.SQRT1_2` is approximately the reciprocal of the value of `Math.SQRT2`.

#### 15.8.1.8 SQRT2

The Number value for the square root of 2, which is approximately 1.4142135623730951.

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

#### 15.8.2 Function Properties of the Math Object

Each of the following `Math` object functions applies the `ToNumber` abstract operator to each of its arguments (in left-to-right order if there is more than one). If `ToNumber` returns an abrupt completion, that completion record is immediately returned. Otherwise, `function` performs a computation on the resulting Number value(s).

**Deleted:** and then

In the function descriptions below, the symbols `NaN`, `-0`, `+0`, `-∞` and `+∞` refer to the Number values described in 8.5.

**NOTE** The behaviour of the functions `acos`, `asin`, `atan`, `atan2`, `cos`, `exp`, `log`, `pow`, `sin`, `sqrt`, and `tan` is not precisely specified here except to require specific results for certain argument values that represent boundary cases of interest. For other argument values, these functions are intended to compute approximations to the results of familiar mathematical functions, but some latitude is allowed in the choice of approximation algorithms. The general intent is that an implementer should be able to use the same mathematical library for ECMAScript on a given hardware platform that is available to C programmers on that platform.

Although the choice of algorithms is left to the implementation, it is recommended (but not specified by this standard) that implementations use the approximation algorithms for IEEE 754 arithmetic contained in `fdlibm`, the freely distributable mathematical library from Sun Microsystems (<http://www.netlib.org/fdlibm>) .

#### 15.8.2.1 `abs (x)`

Returns the absolute value of  $x$ ; the result has the same magnitude as  $x$  but has positive sign.

- If  $x$  is `NaN`, the result is `NaN`.
- If  $x$  is `-0`, the result is `+0`.
- If  $x$  is `-∞`, the result is `+∞`.

#### 15.8.2.2 `acos (x)`

Returns an implementation-dependent approximation to the arc cosine of  $x$ . The result is expressed in radians and ranges from `+0` to `+π`.

- If  $x$  is `NaN`, the result is `NaN`.
- If  $x$  is greater than `1`, the result is `NaN`.
- If  $x$  is less than `-1`, the result is `NaN`.
- If  $x$  is exactly `1`, the result is `+0`.

#### 15.8.2.3 `asin (x)`

Returns an implementation-dependent approximation to the arc sine of  $x$ . The result is expressed in radians and ranges from `-π/2` to `+π/2`.

- If  $x$  is `NaN`, the result is `NaN`.
- If  $x$  is greater than `1`, the result is `NaN`.
- If  $x$  is less than `-1`, the result is `NaN`.
- If  $x$  is `+0`, the result is `+0`.
- If  $x$  is `-0`, the result is `-0`.

#### 15.8.2.4 `atan (x)`

Returns an implementation-dependent approximation to the arc tangent of  $x$ . The result is expressed in radians and ranges from `-π/2` to `+π/2`.

- If  $x$  is `NaN`, the result is `NaN`.
- If  $x$  is `+0`, the result is `+0`.
- If  $x$  is `-0`, the result is `-0`.
- If  $x$  is `+∞`, the result is an implementation-dependent approximation to `+π/2`.
- If  $x$  is `-∞`, the result is an implementation-dependent approximation to `-π/2`.

#### 15.8.2.5 `atan2 (y, x)`

Returns an implementation-dependent approximation to the arc tangent of the quotient  $y/x$  of the arguments  $y$  and  $x$ , where the signs of  $y$  and  $x$  are used to determine the quadrant of the result. Note that it is intentional

and traditional for the two-argument arc tangent function that the argument named  $y$  be first and the argument named  $x$  be second. The result is expressed in radians and ranges from  $-\pi$  to  $+\pi$ .

- If either  $x$  or  $y$  is NaN, the result is NaN.
- If  $y > 0$  and  $x = +0$ , the result is an implementation-dependent approximation to  $+\pi/2$ .
- If  $y > 0$  and  $x = -0$ , the result is an implementation-dependent approximation to  $+\pi/2$ .
- If  $y = +0$  and  $x > 0$ , the result is  $+0$ .
- If  $y = +0$  and  $x = +0$ , the result is  $+0$ .
- If  $y = +0$  and  $x = -0$ , the result is an implementation-dependent approximation to  $+\pi$ .
- If  $y = +0$  and  $x < 0$ , the result is an implementation-dependent approximation to  $+\pi$ .
- If  $y = -0$  and  $x > 0$ , the result is  $-0$ .
- If  $y = -0$  and  $x = +0$ , the result is  $-0$ .
- If  $y = -0$  and  $x = -0$ , the result is an implementation-dependent approximation to  $-\pi$ .
- If  $y = -0$  and  $x < 0$ , the result is an implementation-dependent approximation to  $-\pi$ .
- If  $y < 0$  and  $x = +0$ , the result is an implementation-dependent approximation to  $-\pi/2$ .
- If  $y < 0$  and  $x = -0$ , the result is an implementation-dependent approximation to  $-\pi/2$ .
- If  $y > 0$  and  $y$  is finite and  $x = +\infty$ , the result is  $+0$ .
- If  $y > 0$  and  $y$  is finite and  $x = -\infty$ , the result is an implementation-dependent approximation to  $+\pi$ .
- If  $y < 0$  and  $y$  is finite and  $x = +\infty$ , the result is  $-0$ .
- If  $y < 0$  and  $y$  is finite and  $x = -\infty$ , the result is an implementation-dependent approximation to  $-\pi$ .
- If  $y = +\infty$  and  $x$  is finite, the result is an implementation-dependent approximation to  $+\pi/2$ .
- If  $y = -\infty$  and  $x$  is finite, the result is an implementation-dependent approximation to  $-\pi/2$ .
- If  $y = +\infty$  and  $x = +\infty$ , the result is an implementation-dependent approximation to  $+\pi/4$ .
- If  $y = +\infty$  and  $x = -\infty$ , the result is an implementation-dependent approximation to  $+3\pi/4$ .
- If  $y = -\infty$  and  $x = +\infty$ , the result is an implementation-dependent approximation to  $-\pi/4$ .
- If  $y = -\infty$  and  $x = -\infty$ , the result is an implementation-dependent approximation to  $-3\pi/4$ .

#### 15.8.2.6 `ceil (x)`

Returns the smallest (closest to  $-\infty$ ) Number value that is not less than  $x$  and is equal to a mathematical integer. If  $x$  is already an integer, the result is  $x$ .

- If  $x$  is NaN, the result is NaN.
- If  $x$  is  $+0$ , the result is  $+0$ .
- If  $x$  is  $-0$ , the result is  $-0$ .
- If  $x$  is  $+\infty$ , the result is  $+\infty$ .
- If  $x$  is  $-\infty$ , the result is  $-\infty$ .
- If  $x$  is less than 0 but greater than  $-1$ , the result is  $-0$ .

The value of `Math.ceil (x)` is the same as the value of `-Math.floor (-x)`.

#### 15.8.2.7 `cos (x)`

Returns an implementation-dependent approximation to the cosine of  $x$ . The argument is expressed in radians.

- If  $x$  is NaN, the result is NaN.
- If  $x$  is  $+0$ , the result is 1.
- If  $x$  is  $-0$ , the result is 1.
- If  $x$  is  $+\infty$ , the result is NaN.
- If  $x$  is  $-\infty$ , the result is NaN.

#### 15.8.2.8 `exp (x)`

Returns an implementation-dependent approximation to the exponential function of  $x$  ( $e$  raised to the power of  $x$ , where  $e$  is the base of the natural logarithms).

- If  $x$  is NaN, the result is NaN.

- If  $x$  is  $+0$ , the result is 1.
- If  $x$  is  $-0$ , the result is 1.
- If  $x$  is  $+\infty$ , the result is  $+\infty$ .
- If  $x$  is  $-\infty$ , the result is  $+0$ .

#### 15.8.2.9 `floor (x)`

Returns the greatest (closest to  $+\infty$ ) Number value that is not greater than  $x$  and is equal to a mathematical integer. If  $x$  is already an integer, the result is  $x$ .

- If  $x$  is NaN, the result is NaN.
- If  $x$  is  $+0$ , the result is  $+0$ .
- If  $x$  is  $-0$ , the result is  $-0$ .
- If  $x$  is  $+\infty$ , the result is  $+\infty$ .
- If  $x$  is  $-\infty$ , the result is  $-\infty$ .
- If  $x$  is greater than 0 but less than 1, the result is  $+0$ .

NOTE The value of `Math.floor(x)` is the same as the value of `-Math.ceil(-x)`.

#### 15.8.2.10 `log (x)`

- Returns an implementation-dependent approximation to the natural logarithm of  $x$ .
- If  $x$  is NaN, the result is NaN.
- If  $x$  is less than 0, the result is NaN.
- If  $x$  is  $+0$  or  $-0$ , the result is  $-\infty$ .
- If  $x$  is 1, the result is  $+0$ .
- If  $x$  is  $+\infty$ , the result is  $+\infty$ .

#### 15.8.2.11 `max ([ value1 [, value2 [, ... ]]])`

Given zero or more arguments, calls `ToNumber` on each of the arguments and returns the largest of the resulting values.

- If no arguments are given, the result is  $-\infty$ .
- If any value is NaN, the result is NaN.
- The comparison of values to determine the largest value is done [using the Abstract Relational Comparison Algorithm \(11.8.1\)](#) except that  $+0$  is considered to be larger than  $-0$ .

**Deleted:** as in

**Deleted:** 5

The `length` property of the `max` method is 2.

#### 15.8.2.12 `min ([ value1 [, value2 [, ... ]]])`

Given zero or more arguments, calls `ToNumber` on each of the arguments and returns the smallest of the resulting values.

- If no arguments are given, the result is  $+\infty$ .
- If any value is NaN, the result is NaN.
- The comparison of values to determine the smallest value is done [using the Abstract Relational Comparison Algorithm \(11.8.1\)](#) except that  $+0$  is considered to be larger than  $-0$ .

**Deleted:** as in

**Deleted:** 5

The `length` property of the `min` method is 2.

#### 15.8.2.13 `pow (x, y)`

Returns an implementation-dependent approximation to the result of raising  $x$  to the power  $y$ .

- If  $y$  is NaN, the result is NaN.

- If  $y$  is  $+0$ , the result is  $1$ , even if  $x$  is  $\text{NaN}$ .
- If  $y$  is  $-0$ , the result is  $1$ , even if  $x$  is  $\text{NaN}$ .
- If  $x$  is  $\text{NaN}$  and  $y$  is nonzero, the result is  $\text{NaN}$ .
- If  $\text{abs}(x) > 1$  and  $y$  is  $+\infty$ , the result is  $+\infty$ .
- If  $\text{abs}(x) > 1$  and  $y$  is  $-\infty$ , the result is  $+0$ .
- If  $\text{abs}(x) == 1$  and  $y$  is  $+\infty$ , the result is  $\text{NaN}$ .
- If  $\text{abs}(x) == 1$  and  $y$  is  $-\infty$ , the result is  $\text{NaN}$ .
- If  $\text{abs}(x) < 1$  and  $y$  is  $+\infty$ , the result is  $+0$ .
- If  $\text{abs}(x) < 1$  and  $y$  is  $-\infty$ , the result is  $+\infty$ .
- If  $x$  is  $+\infty$  and  $y > 0$ , the result is  $+\infty$ .
- If  $x$  is  $+\infty$  and  $y < 0$ , the result is  $+0$ .
- If  $x$  is  $-\infty$  and  $y > 0$  and  $y$  is an odd integer, the result is  $-\infty$ .
- If  $x$  is  $-\infty$  and  $y > 0$  and  $y$  is not an odd integer, the result is  $+\infty$ .
- If  $x$  is  $-\infty$  and  $y < 0$  and  $y$  is an odd integer, the result is  $-0$ .
- If  $x$  is  $-\infty$  and  $y < 0$  and  $y$  is not an odd integer, the result is  $+0$ .
- If  $x$  is  $+0$  and  $y > 0$ , the result is  $+0$ .
- If  $x$  is  $+0$  and  $y < 0$ , the result is  $+\infty$ .
- If  $x$  is  $-0$  and  $y > 0$  and  $y$  is an odd integer, the result is  $-0$ .
- If  $x$  is  $-0$  and  $y > 0$  and  $y$  is not an odd integer, the result is  $+0$ .
- If  $x$  is  $-0$  and  $y < 0$  and  $y$  is an odd integer, the result is  $-\infty$ .
- If  $x$  is  $-0$  and  $y < 0$  and  $y$  is not an odd integer, the result is  $+\infty$ .
- If  $x < 0$  and  $x$  is finite and  $y$  is finite and  $y$  is not an integer, the result is  $\text{NaN}$ .

#### 15.8.2.14 random ()

Returns a Number value with positive sign, greater than or equal to  $0$  but less than  $1$ , chosen randomly or pseudo randomly with approximately uniform distribution over that range, using an implementation-dependent algorithm or strategy. This function takes no arguments.

#### 15.8.2.15 round (x)

Returns the Number value that is closest to  $x$  and is equal to a mathematical integer. If two integer Number values are equally close to  $x$ , then the result is the Number value that is closer to  $+\infty$ . If  $x$  is already an integer, the result is  $x$ .

- If  $x$  is  $\text{NaN}$ , the result is  $\text{NaN}$ .
- If  $x$  is  $+0$ , the result is  $+0$ .
- If  $x$  is  $-0$ , the result is  $-0$ .
- If  $x$  is  $+\infty$ , the result is  $+\infty$ .
- If  $x$  is  $-\infty$ , the result is  $-\infty$ .
- If  $x$  is greater than  $0$  but less than  $0.5$ , the result is  $+0$ .
- If  $x$  is less than  $0$  but greater than or equal to  $-0.5$ , the result is  $-0$ .

NOTE 1 `Math.round(3.5)` returns  $4$ , but `Math.round(-3.5)` returns  $-3$ .

NOTE 2 The value of `Math.round(x)` is the same as the value of `Math.floor(x+0.5)`, except when  $x$  is  $-0$  or is less than  $0$  but greater than or equal to  $-0.5$ ; for these cases `Math.round(x)` returns  $-0$ , but `Math.floor(x+0.5)` returns  $+0$ .

#### 15.8.2.16 sin (x)

Returns an implementation-dependent approximation to the sine of  $x$ . The argument is expressed in radians.

- If  $x$  is  $\text{NaN}$ , the result is  $\text{NaN}$ .
- If  $x$  is  $+0$ , the result is  $+0$ .
- If  $x$  is  $-0$ , the result is  $-0$ .
- If  $x$  is  $+\infty$  or  $-\infty$ , the result is  $\text{NaN}$ .

#### 15.8.2.17 `sqrt (x)`

Returns an implementation-dependent approximation to the square root of  $x$ .

- If  $x$  is NaN, the result is NaN.
- If  $x$  is less than 0, the result is NaN.
- If  $x$  is +0, the result is +0.
- If  $x$  is -0, the result is -0.
- If  $x$  is  $+\infty$ , the result is  $+\infty$ .

#### 15.8.2.18 `tan (x)`

Returns an implementation-dependent approximation to the tangent of  $x$ . The argument is expressed in radians.

- If  $x$  is NaN, the result is NaN.
- If  $x$  is +0, the result is +0.
- If  $x$  is -0, the result is -0.
- If  $x$  is  $+\infty$  or  $-\infty$ , the result is NaN.

#### 15.8.2.19 `log10 (x)`

Returns an implementation-dependent approximation to the base 10 logarithm of  $x$ .

- If  $x$  is NaN, the result is NaN.
- If  $x$  is less than 0, the result is NaN.
- If  $x$  is +0, the result is  $-\infty$ .
- If  $x$  is -0, the result is  $-\infty$ .
- If  $x$  is 1, the result is +0.
- If  $x$  is  $+\infty$ , the result is  $+\infty$ .

#### 15.8.2.20 `log2 (x)`

Returns an implementation-dependent approximation to the base 2 logarithm of  $x$ .

- If  $x$  is NaN, the result is NaN.
- If  $x$  is less than 0, the result is NaN.
- If  $x$  is +0, the result is  $-\infty$ .
- If  $x$  is -0, the result is  $-\infty$ .
- If  $x$  is 1, the result is +0.
- If  $x$  is  $+\infty$ , the result is  $+\infty$ .

#### 15.8.2.21 `log1p (x)`

Returns an implementation-dependent approximation to the natural logarithm of  $1 + x$ . The result is computed in a way that is accurate even when the value of  $x$  is close to zero.

- If  $x$  is NaN, the result is NaN.
- If  $x$  is less than -1, the result is NaN.
- If  $x$  is -1, the result is  $-\infty$ .
- If  $x$  is +0, the result is +0.
- If  $x$  is -0, the result is -0.
- If  $x$  is  $+\infty$ , the result is  $+\infty$ .

#### 15.8.2.22 expm1(x)

Returns an implementation-dependent approximation to subtracting 1 from the exponential function of  $x$  ( $e$  raised to the power of  $x$ , where  $e$  is the base of the natural logarithms). The result is computed in a way that is accurate even when the value of  $x$  is close 0.

- If  $x$  is NaN, the result is NaN.
- If  $x$  is +0, the result is +0.
- If  $x$  is -0, the result is -0.
- If  $x$  is  $+\infty$ , the result is  $+\infty$ .
- If  $x$  is  $-\infty$ , the result is -1.

#### 15.8.2.23 cosh(x)

Returns an implementation-dependent approximation to the hyperbolic cosine of  $x$ .

- If  $x$  is NaN, the result is NaN.
- If  $x$  is +0, the result is 1.
- If  $x$  is -0, the result is 1.
- If  $x$  is  $+\infty$ , the result is  $+\infty$ .
- If  $x$  is  $-\infty$ , the result is  $+\infty$ .

NOTE The value of  $\cosh(x)$  is the same as  $(\exp(x) + \exp(-x))/2$ .

#### 15.8.2.24 sinh(x)

Returns an implementation-dependent approximation to the hyperbolic sine of  $x$ .

- If  $x$  is NaN, the result is NaN.
- If  $x$  is +0, the result is +0.
- If  $x$  is -0, the result is -0.
- If  $x$  is  $+\infty$ , the result is  $+\infty$ .
- If  $x$  is  $-\infty$ , the result is  $-\infty$ .

NOTE The value of  $\sinh(x)$  is the same as  $(\exp(x) - \exp(-x))/2$ .

#### 15.8.2.25 tanh(x)

Returns an implementation-dependent approximation to the hyperbolic tangent of  $x$ .

- If  $x$  is NaN, the result is NaN.
- If  $x$  is +0, the result is +0.
- If  $x$  is -0, the result is -0.
- If  $x$  is  $+\infty$ , the result is +1.
- If  $x$  is  $-\infty$ , the result is -1.

NOTE The value of  $\tanh(x)$  is the same as  $(\exp(x) - \exp(-x))/(\exp(x) + \exp(-x))$ .

#### 15.8.2.26 acosh(x)

Returns an implementation-dependent approximation to the inverse hyperbolic cosine of  $x$ .

- If  $x$  is NaN, the result is NaN.
- If  $x$  is less than 1, the result is NaN.
- If  $x$  is 1, the result is +0.
- If  $x$  is  $+\infty$ , the result is  $+\infty$ .

### 15.8.2.27 asinh(x)

Returns an implementation-dependent approximation to the inverse hyperbolic sine of  $x$ .

- If  $x$  is NaN, the result is NaN.
- If  $x$  is +0, the result is +0.
- If  $x$  is -0, the result is -0.
- If  $x$  is  $+\infty$ , the result is  $+\infty$ .
- If  $x$  is  $-\infty$ , the result is  $-\infty$ .

### 15.8.2.28 atanh(x)

Returns an implementation-dependent approximation to the inverse hyperbolic tangent of  $x$ .

- If  $x$  is NaN, the result is NaN.
- If  $x$  is less than -1, the result is NaN.
- If  $x$  is greater than 1, the result is NaN.
- If  $x$  is -1, the result is  $-\infty$ .
- If  $x$  is +1, the result is  $+\infty$ .
- If  $x$  is +0, the result is +0.
- If  $x$  is -0, the result is -0.

### 15.8.2.29 hypot( value1 , value2 [, value3 ] )

Given two or three arguments, `hypot` returns an implementation-dependent approximation of the square root of the sum of squares of its arguments.

- If any argument is  $+\infty$ , the result is  $+\infty$ .
- If any argument is  $-\infty$ , the result is  $+\infty$ .
- If no argument is  $+\infty$  or  $-\infty$ , and any argument is NaN, the result is NaN.
- If all arguments are either +0 or -0, the result is +0.

### 15.8.2.30 hypot2( value1 , value2 [, value3 ] )

Given two or three arguments, `hypot2` returns an implementation-dependent approximation of the sum of squares of its arguments.

**Comment [AWB7103]:** Concensus at March29 TC39 meeting was to drop this function. See March29 meeting notes

- If no arguments are given, the result is +0.
- If any argument is  $+\infty$ , the result is  $+\infty$ .
- If any argument is  $-\infty$ , the result is  $+\infty$ .
- If no argument is  $+\infty$  or  $-\infty$ , and any argument is NaN, the result is NaN.
- If all arguments are either +0 or -0, the result is +0.

### 15.8.2.30 trunc(x)

Returns the integral part of the number  $x$ , removing any fractional digits. If  $x$  is already an integer, the result is  $x$ .

- If  $x$  is NaN, the result is NaN.
- If  $x$  is -0, the result is -0.
- If  $x$  is +0, the result is +0.
- If  $x$  is  $+\infty$ , the result is  $+\infty$ .
- If  $x$  is  $-\infty$ , the result is  $-\infty$ .

### 15.8.2.31 sign(x)

Returns the sign of the  $x$ , indicating whether  $x$  is positive, negative or zero.

- If  $x$  is NaN, the result is NaN.
- If  $x$  is -0, the result is -0.
- If  $x$  is +0, the result is +0.
- If  $x$  is negative, the result is -1.
- If  $x$  is positive, the result is +1.

### 15.8.2.32 cbrt(x)

Returns an implementation-dependent approximation to the cube root of  $x$ .

- If  $x$  is NaN, the result is NaN.
- If  $x$  is +0, the result is +0.
- If  $x$  is -0, the result is -0.
- If  $x$  is +∞, the result is +∞.
- If  $x$  is -∞, the result is -∞.

## 15.9 Date Objects

### 15.9.1 Overview of Date Objects and Definitions of Abstract Operators

The following functions are abstract operations that operate on time values (defined in 15.9.1.1). Note that, in every case, if any argument to one of these functions is NaN, the result will be NaN.

#### 15.9.1.1 Time Values and Time Range

A Date object contains a Number indicating a particular instant in time to within a millisecond. Such a Number is called a *time value*. A time value may also be NaN, indicating that the Date object does not represent a specific instant of time.

Time is measured in ECMAScript in milliseconds since 01 January, 1970 UTC. In time values leap seconds are ignored. It is assumed that there are exactly 86,400,000 milliseconds per day. ECMAScript Number values can represent all integers from -9,007,199,254,740,992 to 9,007,199,254,740,992; this range suffices to measure times to millisecond precision for any instant that is within approximately 285,616 years, either forward or backward, from 01 January, 1970 UTC.

The actual range of times supported by ECMAScript Date objects is slightly smaller: exactly -100,000,000 days to 100,000,000 days measured relative to midnight at the beginning of 01 January, 1970 UTC. This gives a range of 8,640,000,000,000,000 milliseconds to either side of 01 January, 1970 UTC.

The exact moment of midnight at the beginning of 01 January, 1970 UTC is represented by the value +0.

#### 15.9.1.2 Day Number and Time within Day

A given time value  $t$  belongs to day number

$$\text{Day}(t) = \text{floor}(t / \text{msPerDay})$$

where the number of milliseconds per day is

$$\text{msPerDay} = 86400000$$

The remainder is called the time within the day:

$$\text{TimeWithinDay}(t) = t \bmod \text{msPerDay}$$

### 15.9.1.3 Year Number

ECMAScript uses an extrapolated Gregorian system to map a day number to a year number and to determine the month and date within that year. In this system, leap years are precisely those which are (divisible by 4) and ((not divisible by 100) or (divisible by 400)). The number of days in year number  $y$  is therefore defined by

$$\begin{aligned}\text{DaysInYear}(y) &= 365 \text{ if } (y \bmod 4) \neq 0 \\ &= 366 \text{ if } (y \bmod 4) = 0 \text{ and } ((y \bmod 100) \neq 0) \\ &= 365 \text{ if } ((y \bmod 100) = 0) \text{ and } ((y \bmod 400) \neq 0) \\ &= 366 \text{ if } ((y \bmod 400) = 0)\end{aligned}$$

All non-leap years have 365 days with the usual number of days per month and leap years have an extra day in February. The day number of the first day of year  $y$  is given by:

$$\text{DayFromYear}(y) = 365 \times (y - 1970) + \text{floor}((y - 1969)/4) - \text{floor}((y - 1901)/100) + \text{floor}((y - 1601)/400)$$

The time value of the start of a year is:

$$\text{TimeFromYear}(y) = \text{msPerDay} \times \text{DayFromYear}(y)$$

A time value determines a year by:

$$\text{YearFromTime}(t) = \text{the largest integer } y \text{ (closest to positive infinity) such that } \text{TimeFromYear}(y) \leq t$$

The leap-year function is 1 for a time within a leap year and otherwise is zero:

$$\begin{aligned}\text{InLeapYear}(t) &= 0 \text{ if } \text{DaysInYear}(\text{YearFromTime}(t)) = 365 \\ &= 1 \text{ if } \text{DaysInYear}(\text{YearFromTime}(t)) = 366\end{aligned}$$

### 15.9.1.4 Month Number

Months are identified by an integer in the range 0 to 11, inclusive. The mapping  $\text{MonthFromTime}(t)$  from a time value  $t$  to a month number is defined by:

$$\begin{aligned}\text{MonthFromTime}(t) &= 0 \quad \text{if } 0 \leq \text{DayWithinYear}(t) < 31 \\ &= 1 \quad \text{if } 31 \leq \text{DayWithinYear}(t) < 59+\text{InLeapYear}(t) \\ &= 2 \quad \text{if } 59+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 90+\text{InLeapYear}(t) \\ &= 3 \quad \text{if } 90+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 120+\text{InLeapYear}(t) \\ &= 4 \quad \text{if } 120+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 151+\text{InLeapYear}(t) \\ &= 5 \quad \text{if } 151+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 181+\text{InLeapYear}(t) \\ &= 6 \quad \text{if } 181+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 212+\text{InLeapYear}(t) \\ &= 7 \quad \text{if } 212+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 243+\text{InLeapYear}(t) \\ &= 8 \quad \text{if } 243+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 273+\text{InLeapYear}(t) \\ &= 9 \quad \text{if } 273+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 304+\text{InLeapYear}(t) \\ &= 10 \quad \text{if } 304+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 334+\text{InLeapYear}(t) \\ &= 11 \quad \text{if } 334+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 365+\text{InLeapYear}(t)\end{aligned}$$

where

$$\text{DayWithinYear}(t) = \text{Day}(t) - \text{DayFromYear}(\text{YearFromTime}(t))$$

A month value of 0 specifies January; 1 specifies February; 2 specifies March; 3 specifies April; 4 specifies May; 5 specifies June; 6 specifies July; 7 specifies August; 8 specifies September; 9 specifies October; 10 specifies November; and 11 specifies December. Note that  $\text{MonthFromTime}(0) = 0$ , corresponding to Thursday, 01 January, 1970.

### 15.9.1.5 Date Number

A date number is identified by an integer in the range 1 through 31, inclusive. The mapping  $\text{DateFromTime}(t)$  from a time value  $t$  to a month number is defined by:

$$\begin{aligned}\text{DateFromTime}(t) &= \text{DayWithinYear}(t)+1 && \text{if } \text{MonthFromTime}(t)=0 \\ &= \text{DayWithinYear}(t)-30 && \text{if } \text{MonthFromTime}(t)=1 \\ &= \text{DayWithinYear}(t)-58-\text{InLeapYear}(t) && \text{if } \text{MonthFromTime}(t)=2\end{aligned}$$

= DayWithinYear( $t$ )–89–InLeapYear( $t$ )	if MonthFromTime( $t$ )=3
= DayWithinYear( $t$ )–119–InLeapYear( $t$ )	if MonthFromTime( $t$ )=4
= DayWithinYear( $t$ )–150–InLeapYear( $t$ )	if MonthFromTime( $t$ )=5
= DayWithinYear( $t$ )–180–InLeapYear( $t$ )	if MonthFromTime( $t$ )=6
= DayWithinYear( $t$ )–211–InLeapYear( $t$ )	if MonthFromTime( $t$ )=7
= DayWithinYear( $t$ )–242–InLeapYear( $t$ )	if MonthFromTime( $t$ )=8
= DayWithinYear( $t$ )–272–InLeapYear( $t$ )	if MonthFromTime( $t$ )=9
= DayWithinYear( $t$ )–303–InLeapYear( $t$ )	if MonthFromTime( $t$ )=10
= DayWithinYear( $t$ )–333–InLeapYear( $t$ )	if MonthFromTime( $t$ )=11

#### 15.9.1.6 Week Day

The weekday for a particular time value  $t$  is defined as

$$\text{WeekDay}(t) = (\text{Day}(t) + 4) \text{ modulo } 7$$

A weekday value of 0 specifies Sunday; 1 specifies Monday; 2 specifies Tuesday; 3 specifies Wednesday; 4 specifies Thursday; 5 specifies Friday; and 6 specifies Saturday. Note that  $\text{WeekDay}(0) = 4$ , corresponding to Thursday, 01 January, 1970.

#### 15.9.1.7 Local Time Zone Adjustment

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

#### 15.9.1.8 Daylight Saving Time Adjustment

An implementation of ECMAScript is expected to determine the daylight saving time algorithm. The algorithm to determine the daylight saving time adjustment  $\text{DaylightSavingTA}(t)$ , measured in milliseconds, must depend only on four things:

- (1) the time since the beginning of the year  
 $t - \text{TimeFromYear}(\text{YearFromTime}(t))$
- (2) whether  $t$  is in a leap year  
 $\text{InLeapYear}(t)$
- (3) the week day of the beginning of the year  
 $\text{WeekDay}(\text{TimeFromYear}(\text{YearFromTime}(t)))$
- and (4) the geographic location.

The implementation of ECMAScript should not try to determine whether the exact time was subject to daylight saving time, but just whether daylight saving time would have been in effect if the current daylight saving time algorithm had been used at the time. This avoids complications such as taking into account the years that the locale observed daylight saving time year round.

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

#### 15.9.1.9 Local Time

Conversion from UTC to local time is defined by

$$\text{LocalTime}(t) = t + \text{LocalTZA} + \text{DaylightSavingTA}(t)$$

Conversion from local time to UTC is defined by

$$\text{UTC}(t) = t - \text{LocalTZA} - \text{DaylightSavingTA}(t - \text{LocalTZA})$$

Note that  $\text{UTC}(\text{LocalTime}(t))$  is not necessarily always equal to  $t$ .

#### 15.9.1.10 Hours, Minutes, Second, and Milliseconds

The following functions are useful in decomposing time values:

$$\begin{aligned}\text{HourFromTime}(t) &= \text{floor}(t / \text{msPerHour}) \bmod \text{HoursPerDay} \\ \text{MinFromTime}(t) &= \text{floor}(t / \text{msPerMinute}) \bmod \text{MinutesPerHour} \\ \text{SecFromTime}(t) &= \text{floor}(t / \text{msPerSecond}) \bmod \text{SecondsPerMinute} \\ \text{msFromTime}(t) &= t \bmod \text{msPerSecond}\end{aligned}$$

where

$$\begin{aligned}\text{HoursPerDay} &= 24 \\ \text{MinutesPerHour} &= 60 \\ \text{SecondsPerMinute} &= 60 \\ \text{msPerSecond} &= 1000 \\ \text{msPerMinute} &= 60000 = \text{msPerSecond} \times \text{SecondsPerMinute} \\ \text{msPerHour} &= 3600000 = \text{msPerMinute} \times \text{MinutesPerHour}\end{aligned}$$

#### 15.9.1.11 MakeTime (hour, min, sec, ms)

The operator `MakeTime` calculates a number of milliseconds from its four arguments, which must be ECMAScript Number values. This operator functions as follows:

1. If `hour` is not finite or `min` is not finite or `sec` is not finite or `ms` is not finite, return `NaN`.
2. Let  $h$  be `ToInteger(hour)`.
3. Let  $m$  be `ToInteger(min)`.
4. Let  $s$  be `ToInteger(sec)`.
5. Let  $milli$  be `ToInteger(ms)`.
6. Let  $t$  be  $h * \text{msPerHour} + m * \text{msPerMinute} + s * \text{msPerSecond} + milli$ , performing the arithmetic according to IEEE 754 rules (that is, as if using the ECMAScript operators `*` and `+`).
7. Return  $t$ .

#### 15.9.1.12 MakeDay (year, month, date)

The operator `MakeDay` calculates a number of days from its three arguments, which must be ECMAScript Number values. This operator functions as follows:

1. If `year` is not finite or `month` is not finite or `date` is not finite, return `NaN`.
2. Let  $y$  be `ToInteger(year)`.
3. Let  $m$  be `ToInteger(month)`.
4. Let  $dt$  be `ToInteger(date)`.
5. Let  $ym$  be  $y + \text{floor}(m / 12)$ .
6. Let  $mn$  be  $m$  modulo 12.
7. Find a value  $t$  such that  $\text{YearFromTime}(t) == ym$  and  $\text{MonthFromTime}(t) == mn$  and  $\text{DateFromTime}(t) == 1$ ; but if this is not possible (because some argument is out of range), return `NaN`.
8. Return  $\text{Day}(t) + dt - 1$ .

#### 15.9.1.13 MakeDate (day, time)

The operator `MakeDate` calculates a number of milliseconds from its two arguments, which must be ECMAScript Number values. This operator functions as follows:

1. If `day` is not finite or `time` is not finite, return `NaN`.

2. Return  $day \times msPerDay + time$ .

#### 15.9.1.14 TimeClip (time)

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

1. If *time* is not finite, return **NaN**.
2. If  $abs(time) > 8.64 \times 10^{15}$ , return **NaN**.
3. Return an implementation-dependent choice of either `ToInteger(time)` or `ToInteger(time) + (+0)`. (Adding a positive zero converts **-0** to **+0**.)

**NOTE** The point of step 3 is that an implementation is permitted a choice of internal representations of time values, for example as a 64-bit signed integer or as a 64-bit floating-point value. Depending on the implementation, this internal representation may or may not distinguish **-0** and **+0**.

#### 15.9.1.15 Date Time String Format

ECMAScript defines a string interchange format for date-times based upon a simplification of the ISO 8601 Extended Format. The format is as follows: **YYYY-MM-DDTHH:mm:ss.sssZ**

Where the fields are as follows:

- YYYY** is the decimal digits of the year 0000 to 9999 in the Gregorian calendar.
- “-” (hyphen) appears literally twice in the string.
- MM** is the month of the year from 01 (January) to 12 (December).
- DD** is the day of the month from 01 to 31.
- T** “T” appears literally in the string, to indicate the beginning of the time element.
- HH** is the number of complete hours that have passed since midnight as two decimal digits from 00 to 24.
- :
- “:” (colon) appears literally twice in the string.
- mm** is the number of complete minutes since the start of the hour as two decimal digits from 00 to 59.
- ss** is the number of complete seconds since the start of the minute as two decimal digits from 00 to 59.
- .
- “.” (dot) appears literally in the string.
- sss** is the number of complete milliseconds since the start of the second as three decimal digits.
- Z** is the time zone offset specified as “z” (for UTC) or either “+” or “-” followed by a time expression **HH:mm**.

This format includes date-only forms:

```
YYYY
YYYY-MM
YYYY-MM-DD
```

It also includes “date-time” forms that consist of one of the above date-only forms immediately followed by one of the following time forms with an optional time zone offset appended:

```
THH:mm
THH:mm:ss
THH:mm:ss.sss
```

All numbers must be base 10. If the **MM** or **DD** fields are absent “01” is used as the value. If the **HH**, **mm**, or **ss** fields are absent “00” is used as the value and the value of an absent **sss** field is “000”. If the time zone offset is absent, the date-time is interpreted as a local time.

**Deleted:** The value of an

**Deleted:** absent

**Deleted:** is “z”

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

NOTE 1 As every day both starts and ends with midnight, the two notations `00:00` and `24:00` are available to distinguish the two midnights that can be associated with one date. This means that the following two notations refer to exactly the same point in time: `1995-02-04T24:00` and `1995-02-05T00:00`

NOTE 2 There exists no international standard that specifies abbreviations for civil time zones like CET, EST, etc. and sometimes the same abbreviation is even used for two very different time zones. For this reason, ISO 8601 and this format specifies numeric representations of date and time.

#### 15.9.1.15.1 Extended years

ECMAScript requires the ability to specify 6 digit years (extended years); approximately 285,426 years, either forward or backward, from 01 January, 1970 UTC. To represent years before 0 or after 9999, ISO 8601 permits the expansion of the year representation, but only by prior agreement between the sender and the receiver. In the simplified ECMAScript format such an expanded year representation shall have 2 extra year digits and is always prefixed with a + or – sign. The year 0 is considered positive and hence prefixed with a + sign.

NOTE Examples of extended years:

-283457-03-21T15:00:59.008Z	283458 B.C.
-000001-01-01T00:00:00Z	2 B.C.
+000000-01-01T00:00:00Z	1 B.C.
+000001-01-01T00:00:00Z	1 A.D.
+001970-01-01T00:00:00Z	1970 A.D.
+002009-12-15T00:00:00Z	2009 A.D.
+287396-10-12T08:59:00.992Z	287396 A.D.

#### 15.9.2 The Date Constructor Called as a Function

When `Date` is called as a function rather than as a constructor, it returns a String representing the current time (UTC).

NOTE The function call `Date(...)` is not equivalent to the object creation expression `new Date(...)` with the same arguments.

#### 15.9.2.1 `Date ([ year [, month [, date [, hours [, minutes [, seconds [, ms ]]]]]])`

All of the arguments are optional; any arguments supplied are accepted but are completely ignored. A String is created and returned as if by the expression `(new Date()).toString()` where `Date` is the standard built-in constructor with that name and `toString` is the standard built-in method `Date.prototype.toString`.

#### 15.9.3 The Date Constructor

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

#### 15.9.3.1 `new Date (year, month [, date [, hours [, minutes [, seconds [, ms ]]]])`

When `Date` is called with two to seven arguments, it computes the date from `year`, `month`, and (optionally) `date`, `hours`, `minutes`, `seconds` and `ms`.

The `[[Prototype]]` internal property of the newly constructed object is set to the original Date prototype object, the one that is the initial value of `Date.prototype` (15.9.4.1).

The newly constructed object has a `[[NativeBrand]]` internal property whose value is `NativeDate`.

**Deleted:** `[[Class]]` internal property of the  
**Deleted:** is set to "Date"  
**Deleted:** `IsDate`

The `[[Extensible]]` internal property of the newly constructed object is set to `true`.

The `[[PrimitiveValue]]` internal property of the newly constructed object is set as follows:

1. Let  $y$  be `ToNumber(year)`.
2. `ReturnIfAbrupt(year)`.
3. Let  $m$  be `ToNumber(month)`.
4. `ReturnIfAbrupt(month)`.
5. If  $date$  is supplied then let  $dt$  be `ToNumber(date)`; else let  $dt$  be `1`.
6. `ReturnIfAbrupt(dt)`.
7. If  $hours$  is supplied then let  $h$  be `ToNumber(hours)`; else let  $h$  be `0`.
8. `ReturnIfAbrupt(h)`.
9. If  $minutes$  is supplied then let  $min$  be `ToNumber(minutes)`; else let  $min$  be `0`.
10. `ReturnIfAbrupt(min)`.
11. If  $seconds$  is supplied then let  $s$  be `ToNumber(seconds)`; else let  $s$  be `0`.
12. `ReturnIfAbrupt(s)`.
13. If  $ms$  is supplied then let  $milli$  be `ToNumber(ms)`; else let  $milli$  be `0`.
14. `ReturnIfAbrupt(milli)`.
15. If  $y$  is not `NaN` and  $0 \leq \text{ToInteger}(y) \leq 99$ , then let  $yr$  be  $1900 + \text{ToInteger}(y)$ ; otherwise, let  $yr$  be  $y$ .
16. Let  $finalDate$  be `MakeDate(MakeDay(yr, m, dt), MakeTime(h, min, s, milli))`.
17. Set the `[[PrimitiveValue]]` internal property of the newly constructed object to `TimeClip(UTC(finalDate))`.

### 15.9.3.2 new Date (value)

The `[[Prototype]]` internal property of the newly constructed object is set to the original Date prototype object, the one that is the initial value of `Date.prototype` (15.9.4.1).

[The newly constructed object has a `\[\[NativeBrand\]\]` internal property whose value is NativeDate](#)

The `[[Extensible]]` internal property of the newly constructed object is set to `true`.

The `[[PrimitiveValue]]` internal property of the newly constructed object is set as follows:

1. Let  $v$  be `ToPrimitive(value)`.
2. If `Type(v)` is String, then
  - a. Parse  $v$  as a date, in exactly the same manner as for the `parse` method (15.9.4.2); let  $V$  be the time value for this date. If the parse resulted in an abrupt completion, return the Completion Record.
3. Else, let  $V$  be `ToNumber(v).`
4. `ReturnIfAbrupt(V)`.
5. Set the `[[PrimitiveValue]]` internal property of the newly constructed object to `TimeClip(V)` and return.

### 15.9.3.3 new Date ( )

The `[[Prototype]]` internal property of the newly constructed object is set to the original Date prototype object, the one that is the initial value of `Date.prototype` (15.9.4.1).

[The newly constructed object has a `\[\[NativeBrand\]\]` internal property whose value is NativeDate](#)

The `[[Extensible]]` internal property of the newly constructed object is set to `true`.

The `[[PrimitiveValue]]` internal property of the newly constructed object is set to the time value (UTC) identifying the current time.

### 15.9.4 Properties of the Date Constructor

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

Besides the internal properties and the `length` property (whose value is 7), the Date constructor has the following properties:

**Deleted:** `IsDate`

**Deleted:** The `[[Class]]` internal property of the newly constructed object is set to "Date".

**Deleted:** `IsDate`

**Deleted:** The `[[Class]]` internal property of the newly constructed object is set to "Date".

#### 15.9.4.1 Date.prototype

The initial value of `Date.prototype` is the built-in Date prototype object (15.9.5).

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

#### 15.9.4.2 Date.parse (string)

The `parse` function applies the `ToDateString` operator to its argument. If `ToDateString` results in an abrupt completion the Completion Record is immediately returned. Otherwise, `parse` interprets the resulting String as a date and time; it returns a Number, the UTC time value corresponding to the date and time. The String may be interpreted as a local time, a UTC time, or a time in some other time zone, depending on the contents of the String. The function first attempts to parse the format of the String according to the rules called out in Date Time String Format (15.9.1.15). If the String does not conform to that format the function may fall back to any implementation-specific heuristics or implementation-specific date formats. Unrecognisable Strings or dates containing illegal element values in the format String shall cause `Date.parse` to return `NaN`.

**Deleted:**  
**Deleted:** and

If `x` is any Date object whose milliseconds amount is zero within a particular implementation of ECMAScript, then all of the following expressions should produce the same numeric value in that implementation, if all the properties referenced have their initial values:

```
x.valueOf()
Date.parse(x.toString())
Date.parse(x.toUTCString())
Date.parse(x.toISOString())
```

However, the expression

```
Date.parse(x.toLocaleString())
```

is not required to produce the same Number value as the preceding three expressions and, in general, the value produced by `Date.parse` is implementation-dependent when given any String value that does not conform to the Date Time String Format (15.9.1.15) and that could not be produced in that implementation by the `toString` or `toUTCString` method.

#### 15.9.4.3 Date.UTC (year, month [, date [, hours [, minutes [, seconds [, ms ]]]]])

When the `UTC` function is called with fewer than two arguments, the behaviour is implementation-dependent. When the `UTC` function is called with two to seven arguments, it computes the date from `year`, `month` and (optionally) `date`, `hours`, `minutes`, `seconds` and `ms`. The following steps are taken:

1. Let `y` be `ToNumber(year)`.
2. `ReturnIfAbrupt(y)`.
3. Let `m` be `ToNumber(month)`.
4. `ReturnIfAbrupt(m)`.
5. If `date` is supplied then let `dt` be `ToNumber(date)`; else let `dt` be `1`.
6. `ReturnIfAbrupt(dt)`.
7. If `hours` is supplied then let `h` be `ToNumber(hours)`; else let `h` be `0`.
8. `ReturnIfAbrupt(h)`.
9. If `minutes` is supplied then let `min` be `ToNumber(minutes)`; else let `min` be `0`.
10. `ReturnIfAbrupt(min)`.
11. If `seconds` is supplied then let `s` be `ToNumber(seconds)`; else let `s` be `0`.
12. `ReturnIfAbrupt(s)`.
13. If `ms` is supplied then let `milli` be `ToNumber(ms)`; else let `milli` be `0`.
14. `ReturnIfAbrupt(milli)`.
15. If `y` is not `NaN` and  $0 \leq \text{ToInteger}(y) \leq 99$ , then let `yr` be  $1900 + \text{ToInteger}(y)$ ; otherwise, let `yr` be `y`.
16. Return `TimeClip(MakeDate(MakeDay(yr, m, dt), MakeTime(h, min, s, milli)))`.

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

**NOTE** The `UTC` function differs from the `Date` constructor in two ways: it returns a time value as a Number, rather than creating a Date object, and it interprets the arguments in UTC rather than as local time.

#### 15.9.4.4 Date.now()

The `now` function return a Number value that is the time value designating the UTC date and time of the occurrence of the call to `now`.

#### 15.9.5 Properties of the Date Prototype Object

The Date prototype object is itself a Date object and has a [[NativeBrand]] internal property whose value is NativeDate. Its [[PrimitiveValue]] is NaN.

The value of the [[Prototype]] internal property of the Date prototype object is the standard built-in Object prototype object (15.2.4).

In following descriptions of functions that are properties of the Date prototype object, the phrase "this Date object" refers to the object that is the `this` value for the invocation of the function. Unless explicitly noted otherwise, none of these functions are generic; a `TypeError` exception is thrown if the `this` value is not an object with a [[NativeBrand]] internal property whose value is NativeDate. Also, the phrase "this time value" refers to the Number value for the time represented by this Date object, that is, the value of the [[PrimitiveValue]] internal property of this Date object.

##### 15.9.5.1 Date.prototype.constructor

The initial value of `Date.prototype.constructor` is the built-in `Date` constructor.

##### 15.9.5.2 Date.prototype.toString()

This function returns a String value. If this time value is NaN, the String value is "Invalid Date", otherwise the contents of the String are implementation-dependent, but are intended to represent the Date in the current time zone in a convenient, human-readable form.

**NOTE** For any Date value `d` whose milliseconds amount is zero, the result of `Date.parse(d.toString())` is equal to `d.valueOf()`. See 15.9.4.2.

##### 15.9.5.3 Date.prototype.toDateString()

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

##### 15.9.5.4 Date.prototype.toTimeString()

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

##### 15.9.5.5 Date.prototype.toLocaleString()

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

**NOTE** The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

##### 15.9.5.6 Date.prototype.toLocaleDateString()

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

**Deleted:** (its [[Class]] is "Date")

**Deleted:** IsDate

**Deleted:** whose

**Deleted:** for which the value of the [[Class]]

**Deleted:** IsDate

**Deleted:** is "Date"

**Deleted:** The

**NOTE** The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

#### **15.9.5.7 Date.prototype.toLocaleTimeString ()**

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

**NOTE** The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

#### **15.9.5.8 Date.prototype.valueOf ()**

The `valueOf` function returns a Number, which is this time value.

#### **15.9.5.9 Date.prototype.getTime ()**

1. Return this time value.

#### **15.9.5.10 Date.prototype.getFullYear ()**

1. Let  $t$  be this time value.
2. If  $t$  is `Nan`, return `Nan`.
3. Return `YearFromTime(LocalTime( $t$ ))`.

#### **15.9.5.11 Date.prototype.getUTCFullYear ()**

1. Let  $t$  be this time value.
2. If  $t$  is `Nan`, return `Nan`.
3. Return `YearFromTime( $t$ )`.

#### **15.9.5.12 Date.prototype.getMonth ()**

1. Let  $t$  be this time value.
2. If  $t$  is `Nan`, return `Nan`.
3. Return `MonthFromTime(LocalTime( $t$ ))`.

#### **15.9.5.13 Date.prototype.getUTCMonth ()**

1. Let  $t$  be this time value.
2. If  $t$  is `Nan`, return `Nan`.
3. Return `MonthFromTime( $t$ )`.

#### **15.9.5.14 Date.prototype.getDate ()**

1. Let  $t$  be this time value.
2. If  $t$  is `Nan`, return `Nan`.
3. Return `DateFromTime(LocalTime( $t$ ))`.

#### **15.9.5.15 Date.prototype.getUTCDate ()**

1. Let  $t$  be this time value.
2. If  $t$  is `Nan`, return `Nan`.
3. Return `DateFromTime( $t$ )`.

#### **15.9.5.16 Date.prototype.getDay ()**

1. Let  $t$  be this time value.
2. If  $t$  is `Nan`, return `Nan`.

3. Return WeekDay(LocalTime( $t$ )).

#### **15.9.5.17 Date.prototype.getDay ()**

1. Let  $t$  be this time value.
2. If  $t$  is **NAN**, return **NAN**.
3. Return WeekDay( $t$ ).

#### **15.9.5.18 Date.prototype.getHours ()**

1. Let  $t$  be this time value.
2. If  $t$  is **NAN**, return **NAN**.
3. Return HourFromTime(LocalTime( $t$ )).

#### **15.9.5.19 Date.prototype.getUTCHours ()**

1. Let  $t$  be this time value.
2. If  $t$  is **NAN**, return **NAN**.
3. Return HourFromTime( $t$ ).

#### **15.9.5.20 Date.prototype.getMinutes ()**

1. Let  $t$  be this time value.
2. If  $t$  is **NAN**, return **NAN**.
3. Return MinFromTime(LocalTime( $t$ )).

#### **15.9.5.21 Date.prototype.getUTCMinutes ()**

1. Let  $t$  be this time value.
2. If  $t$  is **NAN**, return **NAN**.
3. Return MinFromTime( $t$ ).

#### **15.9.5.22 Date.prototype.getSeconds ()**

1. Let  $t$  be this time value.
2. If  $t$  is **NAN**, return **NAN**.
3. Return SecFromTime(LocalTime( $t$ )).

#### **15.9.5.23 Date.prototype.getUTCSeconds ()**

1. Let  $t$  be this time value.
2. If  $t$  is **NAN**, return **NAN**.
3. Return SecFromTime( $t$ ).

#### **15.9.5.24 Date.prototype.getMilliseconds ()**

1. Let  $t$  be this time value.
2. If  $t$  is **NAN**, return **NAN**.
3. Return msFromTime(LocalTime( $t$ )).

#### **15.9.5.25 Date.prototype.getUTCMilliseconds ()**

1. Let  $t$  be this time value.
2. If  $t$  is **NAN**, return **NAN**.
3. Return msFromTime( $t$ ).

#### **15.9.5.26 Date.prototype.getTimezoneOffset ()**

Returns the difference between local time and UTC time in minutes.

1. Let  $t$  be this time value.
2. If  $t$  is `NAN`, return `NAN`.
3. Return  $(t - \text{LocalTime}(t)) / \text{msPerMinute}$ .

#### **15.9.5.27 Date.prototype.setTime (time)**

1. Let  $v$  be `TimeClip(ToNumber(time))`.
2. Set the `[[PrimitiveValue]]` internal property of this Date object to  $v$ .
3. Return  $v$ .

#### **15.9.5.28 Date.prototype.setMilliseconds (ms)**

1. Let  $t$  be the result of `LocalTime(this time value)`.
2. Let  $time$  be `MakeTime(HourFromTime(t), MinFromTime(t), SecFromTime(t), ToNumber(ms))`.
3. Let  $u$  be `TimeClip(UTC(MakeDate(Day(t), time)))`.
4. Set the `[[PrimitiveValue]]` internal property of this Date object to  $u$ .
5. Return  $u$ .

#### **15.9.5.29 Date.prototype.setUTCMilliseconds (ms)**

1. Let  $t$  be this time value.
2. Let  $time$  be `MakeTime(HourFromTime(t), MinFromTime(t), SecFromTime(t), ToNumber(ms))`.
3. Let  $v$  be `TimeClip(MakeDate(Day(t), time))`.
4. Set the `[[PrimitiveValue]]` internal property of this Date object to  $v$ .
5. Return  $v$ .

#### **15.9.5.30 Date.prototype.setSeconds (sec [, ms ])**

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

1. Let  $t$  be the result of `LocalTime(this time value)`.
2. Let  $s$  be `ToNumber(sec)`.
3. If  $ms$  is not specified, then let  $milli$  be `msFromTime(t)`; otherwise, let  $milli$  be `ToNumber(ms)`.
4. Let  $date$  be `MakeDate(Day(t), MakeTime(HourFromTime(t), MinFromTime(t), s, milli))`.
5. Let  $u$  be `TimeClip(UTC(date))`.
6. Set the `[[PrimitiveValue]]` internal property of this Date object to  $u$ .
7. Return  $u$ .

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

#### **15.9.5.31 Date.prototype.setUTCSeconds (sec [, ms ])**

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

1. Let  $t$  be this time value.
2. Let  $s$  be `ToNumber(sec)`.
3. If  $ms$  is not specified, then let  $milli$  be `msFromTime(t)`; otherwise, let  $milli$  be `ToNumber(ms)`.
4. Let  $date$  be `MakeDate(Day(t), MakeTime(HourFromTime(t), MinFromTime(t), s, milli))`.
5. Let  $v$  be `TimeClip(date)`.
6. Set the `[[PrimitiveValue]]` internal property of this Date object to  $v$ .
7. Return  $v$ .

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

#### **15.9.5.32 Date.prototype.setMinutes (min [, sec [, ms ]])**

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

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

1. Let  $t$  be the result of LocalTime(this time value).
2. Let  $m$  be ToNumber( $min$ ).
3. If  $sec$  is not specified, then let  $s$  be SecFromTime( $t$ ); otherwise, let  $s$  be ToNumber( $sec$ ).
4. If  $ms$  is not specified, then let  $milli$  be msFromTime( $t$ ); otherwise, let  $milli$  be ToNumber( $ms$ ).
5. Let  $date$  be MakeDate(Day( $t$ ), MakeTime(HourFromTime( $t$ ),  $m$ ,  $s$ ,  $milli$ )).
6. Let  $u$  be TimeClip(UTC( $date$ )).
7. Set the [[PrimitiveValue]] internal property of this Date object to  $u$ .
8. Return  $u$ .

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

#### **15.9.5.33 Date.prototype.setUTCMilliseconds (min [, sec [, ms ]])**

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

If  $ms$  is not specified, this function behaves as if  $ms$  were specified with the value return by `getUTCMilliseconds()`.

1. Let  $t$  be this time value.
2. Let  $m$  be ToNumber( $min$ ).
3. If  $sec$  is not specified, then let  $s$  be SecFromTime( $t$ ); otherwise, let  $s$  be ToNumber( $sec$ ).
4. If  $ms$  is not specified, then let  $milli$  be msFromTime( $t$ ); otherwise, let  $milli$  be ToNumber( $ms$ ).
5. Let  $date$  be MakeDate(Day( $t$ ), MakeTime(HourFromTime( $t$ ),  $m$ ,  $s$ ,  $milli$ )).
6. Let  $v$  be TimeClip( $date$ ).
7. Set the [[PrimitiveValue]] internal property of this Date object to  $v$ .
8. Return  $v$ .

The `length` property of the `setUTCMilliseconds` method is 3.

#### **15.9.5.34 Date.prototype.setHours (hour [, min [, sec [, ms ]]])**

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

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

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

1. Let  $t$  be the result of LocalTime(this time value).
2. Let  $h$  be ToNumber( $hour$ ).
3. If  $min$  is not specified, then let  $m$  be MinFromTime( $t$ ); otherwise, let  $m$  be ToNumber( $min$ ).
4. If  $sec$  is not specified, then let  $s$  be SecFromTime( $t$ ); otherwise, let  $s$  be ToNumber( $sec$ ).
5. If  $ms$  is not specified, then let  $milli$  be msFromTime( $t$ ); otherwise, let  $milli$  be ToNumber( $ms$ ).
6. Let  $date$  be MakeDate(Day( $t$ ), MakeTime( $h$ ,  $m$ ,  $s$ ,  $milli$ )).
7. Let  $u$  be TimeClip(UTC( $date$ )).
8. Set the [[PrimitiveValue]] internal property of this Date object to  $u$ .
9. Return  $u$ .

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

#### **15.9.5.35 Date.prototype.setUTCHours (hour [, min [, sec [, ms ]]])**

If  $min$  is not specified, this behaves as if  $min$  were specified with the value `getUTCMilliseconds()`.

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

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

1. Let  $t$  be this time value.

2. Let  $h$  be  $\text{ToNumber}(hour)$ .
3. If  $min$  is not specified, then let  $m$  be  $\text{MinFromTime}(t)$ ; otherwise, let  $m$  be  $\text{ToNumber}(min)$ .
4. If  $sec$  is not specified, then let  $s$  be  $\text{SecFromTime}(t)$ ; otherwise, let  $s$  be  $\text{ToNumber}(sec)$ .
5. If  $ms$  is not specified, then let  $milli$  be  $\text{msFromTime}(t)$ ; otherwise, let  $milli$  be  $\text{ToNumber}(ms)$ .
6. Let  $\text{newDate}$  be  $\text{MakeDate}(\text{Day}(t), \text{MakeTime}(h, m, s, milli))$ .
7. Let  $v$  be  $\text{TimeClip}(\text{newDate})$ .
8. Set the  $\text{[[PrimitiveValue]]}$  internal property of this Date object to  $v$ .
9. Return  $v$ .

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

#### **15.9.5.36 Date.prototype.setDate (date)**

1. Let  $t$  be the result of  $\text{LocalTime}(\text{this time value})$ .
2. Let  $dt$  be  $\text{ToNumber}(date)$ .
3. Let  $\text{newDate}$  be  $\text{MakeDate}(\text{MakeDay}(\text{YearFromTime}(t), \text{MonthFromTime}(t), dt), \text{TimeWithinDay}(t))$ .
4. Let  $u$  be  $\text{TimeClip}(\text{UTC}(\text{newDate}))$ .
5. Set the  $\text{[[PrimitiveValue]]}$  internal property of this Date object to  $u$ .
6. Return  $u$ .

#### **15.9.5.37 Date.prototype.setUTCDate (date)**

1. Let  $t$  be this time value.
2. Let  $dt$  be  $\text{ToNumber}(date)$ .
3. Let  $\text{newDate}$  be  $\text{MakeDate}(\text{MakeDay}(\text{YearFromTime}(t), \text{MonthFromTime}(t), dt), \text{TimeWithinDay}(t))$ .
4. Let  $v$  be  $\text{TimeClip}(\text{newDate})$ .
5. Set the  $\text{[[PrimitiveValue]]}$  internal property of this Date object to  $v$ .
6. Return  $v$ .

#### **15.9.5.38 Date.prototype.setMonth (month [, date ] )**

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

1. Let  $t$  be the result of  $\text{LocalTime}(\text{this time value})$ .
2. Let  $m$  be  $\text{ToNumber}(month)$ .
3. If `date` is not specified, then let  $dt$  be  $\text{DateFromTime}(t)$ ; otherwise, let  $dt$  be  $\text{ToNumber}(date)$ .
4. Let  $\text{newDate}$  be  $\text{MakeDate}(\text{MakeDay}(\text{YearFromTime}(t), m, dt), \text{TimeWithinDay}(t))$ .
5. Let  $u$  be  $\text{TimeClip}(\text{UTC}(\text{newDate}))$ .
6. Set the  $\text{[[PrimitiveValue]]}$  internal property of this Date object to  $u$ .
7. Return  $u$ .

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

#### **15.9.5.39 Date.prototype.setUTCMonth (month [, date ] )**

If `date` is not specified, this behaves as if `date` were specified with the value `getUTCDate()`.

1. Let  $t$  be this time value.
2. Let  $m$  be  $\text{ToNumber}(month)$ .
3. If `date` is not specified, then let  $dt$  be  $\text{DateFromTime}(t)$ ; otherwise, let  $dt$  be  $\text{ToNumber}(date)$ .
4. Let  $\text{newDate}$  be  $\text{MakeDate}(\text{MakeDay}(\text{YearFromTime}(t), m, dt), \text{TimeWithinDay}(t))$ .
5. Let  $v$  be  $\text{TimeClip}(\text{newDate})$ .
6. Set the  $\text{[[PrimitiveValue]]}$  internal property of this Date object to  $v$ .
7. Return  $v$ .

The `length` property of the `setUTCMonth` method is 2.

#### 15.9.5.40 Date.prototype.setFullYear (year [, month [, date ]])

If *month* is not specified, this behaves as if *month* were specified with the value `getMonth()`.

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

1. Let *t* be the result of `LocalTime(this time value)`; but if this time value is `NaN`, let *t* be `+0`.
2. Let *y* be `ToNumber(year)`.
3. If *month* is not specified, then let *m* be `MonthFromTime(t)`; otherwise, let *m* be `ToNumber(month)`.
4. If *date* is not specified, then let *dt* be `DateFromTime(t)`; otherwise, let *dt* be `ToNumber(date)`.
5. Let *newDate* be `MakeDate(MakeDay(y, m, dt), TimeWithinDay(t))`.
6. Let *u* be `TimeClip(UTC(newDate))`.
7. Set the `[[PrimitiveValue]]` internal property of this Date object to *u*.
8. Return *u*.

The `length` property of the `setFullYear` method is **3**.

#### 15.9.5.41 Date.prototype.setUTCFullYear (year [, month [, date ]])

If *month* is not specified, this behaves as if *month* were specified with the value `getUTCMonth()`.

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

1. Let *t* be this time value; but if this time value is `NaN`, let *t* be `+0`.
2. Let *y* be `ToNumber(year)`.
3. If *month* is not specified, then let *m* be `MonthFromTime(t)`; otherwise, let *m* be `ToNumber(month)`.
4. If *date* is not specified, then let *dt* be `DateFromTime(t)`; otherwise, let *dt* be `ToNumber(date)`.
5. Let *newDate* be `MakeDate(MakeDay(y, m, dt), TimeWithinDay(t))`.
6. Let *v* be `TimeClip(newDate)`.
7. Set the `[[PrimitiveValue]]` internal property of this Date object to *v*.
8. Return *v*.

The `length` property of the `setUTCFullYear` method is **3**.

#### 15.9.5.42 Date.prototype.toUTCString ()

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

**NOTE** The intent is to produce a String representation of a date that is more readable than the format specified in 15.9.1.15. It is not essential that the chosen format be unambiguous or easily machine parseable. If an implementation does not have a preferred human-readable format it is recommended to use the format defined in 15.9.1.15 but with a space rather than a "T" used to separate the date and time elements.

#### 15.9.5.43 Date.prototype.toISOString ()

This function returns a String value representing the instance in time represented by this Date object. The format of the String is the Date Time string format defined in 15.9.1.15. All fields are present in the String. The time zone is always UTC, denoted by the suffix Z. If the time value of this object is not a finite Number a **RangeError** exception is thrown.

#### 15.9.5.44 Date.prototype.toJSON (key)

This function provides a String representation of a Date object for use by `JSON.stringify` (15.12.3).

When the `toJSON` method is called with argument *key*, the following steps are taken:

1. Let *O* be the result of calling `ToObject`, giving it the `this` value as its argument.
2. Let *tv* be `ToPrimitive(O, hint Number)`.

3. If  $tv$  is a Number and is not finite, return **null**.
4. Let  $toISO$  be the result of calling the  $[[Get]]$  internal method of  $O$  with argument "**toISOString**".
- 5. ReturnIfAbrupt( $toISO$ )**.
6. If  $\text{IsCallable}(toISO)$  is **false**, throw a **TypeError** exception.
7. Return the result of calling the  $[[Call]]$  internal method of  $toISO$  with  $O$  as the **this** value and an empty argument list.

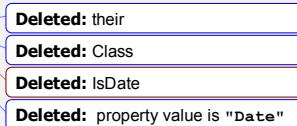
NOTE 1 The argument is ignored.

NOTE 2 The **toJSON** function is intentionally generic; it does not require that its **this** value be a Date object. Therefore, it can be transferred to other kinds of objects for use as a method. However, it does require that any such object have a **toISOString** method. An object is free to use the argument  $key$  to filter its stringification.

### 15.9.6 Properties of Date Instances

Date instances inherit properties from the Date prototype object and have a  $[[NativeBrand]]$  internal whose value is **NativeDate**. Date instances also have a  $[[PrimitiveValue]]$  internal property.

The  $[[PrimitiveValue]]$  internal property is time value represented by this Date object.



### 15.10 RegExp (Regular Expression) Objects

A RegExp object contains a regular expression and the associated flags.

NOTE The form and functionality of regular expressions is modelled after the regular expression facility in the Perl 5 programming language.

#### 15.10.1 Patterns

The **RegExp** constructor applies the following grammar to the input pattern String. An error occurs if the grammar cannot interpret the String as an expansion of *Pattern*.

##### Syntax

*Pattern* ::=  
    *Disjunction*

*Disjunction* ::=  
    *Alternative*  
    *Alternative* | *Disjunction*

*Alternative* ::=  
    [empty]  
    *Alternative* *Term*

*Term* ::=  
    *Assertion*  
    *Atom*  
    *Atom Quantifier*

*Assertion* ::=  
    ^  
    \$  
    \ b  
    \ B  
    ( ? = *Disjunction* )  
    ( ? ! *Disjunction* )

*Quantifier ::*

*QuantifierPrefix*  
  *QuantifierPrefix* ?

*QuantifierPrefix ::*

  \*  
  +  
  ?  
  { *DecimalDigits* }  
  { *DecimalDigits* , }  
  { *DecimalDigits* , *DecimalDigits* }

*Atom ::*

*PatternCharacter*  
  .  
  \ *AtomEscape*  
  *CharacterClass*  
  ( *Disjunction* )  
  ( ? : *Disjunction* )

*PatternCharacter ::*

*SourceCharacter* **but not one of**  
  ^ \$ \ . \* + ? ( ) [ ] { } |

*AtomEscape ::*

*DecimalEscape*  
  *CharacterEscape*  
  *CharacterClassEscape*

*CharacterEscape ::*

*ControlEscape*  
  c *ControlLetter*  
  *HexEscapeSequence*  
  *UnicodeEscapeSequence*  
  *IdentityEscape*

*ControlEscape :: one of*

  f n r t v

*ControlLetter :: one of*

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

*IdentityEscape ::*

*SourceCharacter* **but not** *IdentifierPart*  
  <ZWJ>  
  <ZWNJ>

*DecimalEscape ::*

*DecimalIntegerLiteral* [lookahead ≠ *DecimalDigit*]

*CharacterClassEscape :: one of*

  d D s S w W

*CharacterClass ::*

  [ [lookahead ≠ {^}] *ClassRanges* ]  
  [ ^ *ClassRanges* ]

*ClassRanges* ::  
 [empty]  
*NonemptyClassRanges*

*NonemptyClassRanges* ::  
*ClassAtom*  
*ClassAtom NonemptyClassRangesNoDash*  
*ClassAtom* - *ClassAtom ClassRanges*

*NonemptyClassRangesNoDash* ::  
*ClassAtom*  
*ClassAtomNoDash NonemptyClassRangesNoDash*  
*ClassAtomNoDash* - *ClassAtom ClassRanges*

*ClassAtom* ::  
 -  
*ClassAtomNoDash*

*ClassAtomNoDash* ::  
*SourceCharacter but not one of \ or ] or -*  
*\ ClassEscape*

*ClassEscape* ::  
*DecimalEscape*  
**b**  
*CharacterEscape*  
*CharacterClassEscape*

## 15.10.2 Pattern Semantics

A regular expression pattern is converted into an internal procedure using the process described below. An implementation is encouraged to use more efficient algorithms than the ones listed below, as long as the results are the same. The internal procedure is used as the value of a `RegExp` object's `[[Match]]` internal property.

### 15.10.2.1 Notation

The descriptions below use the following variables:

- *Input* is the String being matched by the regular expression pattern. The notation `input[n]` means the *n*<sup>th</sup> character of *input*, where *n* can range between 0 (inclusive) and *InputLength* (exclusive).
- *InputLength* is the number of characters in the *Input* String.
- *NcapturingParens* is the total number of left capturing parentheses (i.e. the total number of times the *Atom :: ( Disjunction )* production is expanded) in the pattern. A left capturing parenthesis is any ( pattern character that is matched by the ( terminal of the *Atom :: ( Disjunction )* production.
- *IgnoreCase* is the setting of the `RegExp` object's `ignoreCase` property.
- *Multiline* is the setting of the `RegExp` object's `multiline` property.

Furthermore, the descriptions below use the following internal data structures:

- A *CharSet* is a mathematical set of characters.
- A *State* is an ordered pair `(endIndex, captures)` where *endIndex* is an integer and *captures* is an internal array of *NcapturingParens* values. *States* are used to represent partial match states in the regular expression matching algorithms. The *endIndex* is one plus the index of the last input character matched so far by the pattern, while *captures* holds the results of capturing parentheses. The *n*<sup>th</sup> element of *captures* is either a String that represents the value obtained by the *n*<sup>th</sup> set of capturing parentheses or `undefined` if the *n*<sup>th</sup> set of capturing parentheses hasn't been reached yet. Due to backtracking, many *States* may be in use at any time during the matching process.

- A *MatchResult* is either a *State* or the special token **failure** that indicates that the match failed.
- A *Continuation* procedure is an internal closure (i.e. an internal procedure with some arguments already bound to values) that takes one *State* argument and returns a *MatchResult* result. If an internal closure references variables bound in the function that creates the closure, the closure uses the values that these variables had at the time the closure was created. The *Continuation* attempts to match the remaining portion (specified by the closure's already-bound arguments) of the pattern against the input String, starting at the intermediate state given by its *State* argument. If the match succeeds, the *Continuation* returns the final *State* that it reached; if the match fails, the *Continuation* returns **failure**.
- A *Matcher* procedure is an internal closure that takes two arguments -- a *State* and a *Continuation* -- and returns a *MatchResult* result. A *Matcher* attempts to match a middle subpattern (specified by the closure's already-bound arguments) of the pattern against the input String, starting at the intermediate state given by its *State* argument. The *Continuation* argument should be a closure that matches the rest of the pattern. After matching the subpattern of a pattern to obtain a new *State*, the *Matcher* then calls *Continuation* on that new *State* to test if the rest of the pattern can match as well. If it can, the *Matcher* returns the *State* returned by *Continuation*; if not, the *Matcher* may try different choices at its choice points, repeatedly calling *Continuation* until it either succeeds or all possibilities have been exhausted.
- An *AssertionTester* procedure is an internal closure that takes a *State* argument and returns a Boolean result. The assertion tester tests a specific condition (specified by the closure's already-bound arguments) against the current place in the input String and returns **true** if the condition matched or **false** if not.
- An *EscapeValue* is either a character or an integer. An *EscapeValue* is used to denote the interpretation of a *DecimalEscape* escape sequence: a character *ch* means that the escape sequence is interpreted as the character *ch*, while an integer *n* means that the escape sequence is interpreted as a backreference to the *n*<sup>th</sup> set of capturing parentheses.

### 15.10.2.2 Pattern

The production *Pattern :: Disjunction* evaluates as follows:

1. Evaluate *Disjunction* to obtain a *Matcher* *m*.
2. Return an internal closure that takes two arguments, a String *str* and an integer *index*, and performs the following:
  1. Let *Input* be the given String *str*. This variable will be used throughout the algorithms in 15.10.2.
  2. Let *InputLength* be the length of *Input*. This variable will be used throughout the algorithms in 15.10.2.
  3. Let *c* be a *Continuation* that always returns its *State* argument as a successful *MatchResult*.
  4. Let *cap* be an internal array of *NcapturingParens* **undefined** values, indexed 1 through *NcapturingParens*.
  5. Let *x* be the *State* (*index*, *cap*).
  6. Call *m(x, c)* and return its result.

**NOTE** A Pattern evaluates ("compiles") to an internal procedure value. `RegExp.prototype.exec` can then apply this procedure to a String and an offset within the String to determine whether the pattern would match starting at exactly that offset within the String, and, if it does match, what the values of the capturing parentheses would be. The algorithms in 15.10.2 are designed so that compiling a pattern may throw a **SyntaxError** exception; on the other hand, once the pattern is successfully compiled, applying its result internal procedure to find a match in a String cannot throw an exception (except for any host-defined exceptions that can occur anywhere such as out-of-memory).

### 15.10.2.3 Disjunction

The production *Disjunction :: Alternative* evaluates by evaluating *Alternative* to obtain a *Matcher* and returning that *Matcher*.

The production *Disjunction :: Alternative | Disjunction* evaluates as follows:

1. Evaluate *Alternative* to obtain a *Matcher* *m1*.

2. Evaluate *Disjunction* to obtain a Matcher *m*<sub>2</sub>.
3. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
  1. Call *m*<sub>1</sub>(*x*, *c*) and let *r* be its result.
  2. If *r* isn't **failure**, return *r*.
  3. Call *m*<sub>2</sub>(*x*, *c*) and return its result.

**NOTE** The `|` regular expression operator separates two alternatives. The pattern first tries to match the left *Alternative* (followed by the sequel of the regular expression); if it fails, it tries to match the right *Disjunction* (followed by the sequel of the regular expression). If the left *Alternative*, the right *Disjunction*, and the sequel all have choice points, all choices in the sequel are tried before moving on to the next choice in the left *Alternative*. If choices in the left *Alternative* are exhausted, the right *Disjunction* is tried instead of the left *Alternative*. Any capturing parentheses inside a portion of the pattern skipped by `|` produce **undefined** values instead of Strings. Thus, for example,

```
/a|ab/.exec("abc")
returns the result "a" and not "ab". Moreover,
/((a)|(ab))((c)|(bc))/.exec("abc")
returns the array
["abc", "a", "a", undefined, "bc", undefined, "bc"]
and not
["abc", "ab", undefined, "ab", "c", "c", undefined]
```

#### 15.10.2.4 Alternative

The production *Alternative* :: [empty] evaluates by returning a Matcher that takes two arguments, a State *x* and a Continuation *c*, and returns the result of calling *c*(*x*).

The production *Alternative* :: *Alternative Term* evaluates as follows:

1. Evaluate *Alternative* to obtain a Matcher *m*<sub>1</sub>.
2. Evaluate *Term* to obtain a Matcher *m*<sub>2</sub>.
3. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
  1. Create a Continuation *d* that takes a State argument *y* and returns the result of calling *m*<sub>2</sub>(*y*, *c*).
  2. Call *m*<sub>1</sub>(*x*, *d*) and return its result.

**NOTE** Consecutive *Terms* try to simultaneously match consecutive portions of the input String. If the left *Alternative*, the right *Term*, and the sequel of the regular expression all have choice points, all choices in the sequel are tried before moving on to the next choice in the right *Term*, and all choices in the right *Term* are tried before moving on to the next choice in the left *Alternative*.

#### 15.10.2.5 Term

The production *Term* :: *Assertion* evaluates by returning an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:

1. Evaluate *Assertion* to obtain an AssertionTester *t*.
2. Call *t*(*x*) and let *r* be the resulting Boolean value.
3. If *r* is **false**, return **failure**.
4. Call *c*(*x*) and return its result.

The production *Term* :: *Atom* evaluates by evaluating *Atom* to obtain a Matcher and returning that Matcher.

The production *Term* :: *Atom Quantifier* evaluates as follows:

1. Evaluate *Atom* to obtain a Matcher *m*.
2. Evaluate *Quantifier* to obtain the three results: an integer *min*, an integer (or  $\infty$ ) *max*, and Boolean *greedy*.
3. If *max* is finite and less than *min*, then throw a **SyntaxError** exception.
4. Let *parenIndex* be the number of left capturing parentheses in the entire regular expression that occur to the left of this production expansion's *Term*. This is the total number of times the *Atom* :: ( *Disjunction* )

production is expanded prior to this production's *Term* plus the total number of *Atom* :: ( *Disjunction* ) productions enclosing this *Term*.

5. Let *parenCount* be the number of left capturing parentheses in the expansion of this production's *Atom*. This is the total number of *Atom* :: ( *Disjunction* ) productions enclosed by this production's *Atom*.
6. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:

1. Call *RepeatMatcher*(*m*, *min*, *max*, *greedy*, *x*, *c*, *parenIndex*, *parenCount*) and return its result.

The abstract operation *RepeatMatcher* takes eight parameters, a Matcher *m*, an integer *min*, an integer (or  $\infty$ ) *max*, a Boolean *greedy*, a State *x*, a Continuation *c*, an integer *parenIndex*, and an integer *parenCount*, and performs the following:

1. If *max* is zero, then call *c(x)* and return its result.
2. Create an internal Continuation closure *d* that takes one State argument *y* and performs the following:
  1. If *min* is zero and *y*'s *endIndex* is equal to *x*'s *endIndex*, then return **failure**.
  2. If *min* is zero then let *min2* be zero; otherwise let *min2* be *min*-1.
  3. If *max* is  $\infty$ , then let *max2* be  $\infty$ ; otherwise let *max2* be *max*-1.
  4. Call *RepeatMatcher*(*m*, *min2*, *max2*, *greedy*, *y*, *c*, *parenIndex*, *parenCount*) and return its result.
3. Let *cap* be a fresh copy of *x*'s *captures* internal array.
4. For every integer *k* that satisfies *parenIndex* < *k* and *k*  $\leq$  *parenIndex+parenCount*, set *cap[k]* to **undefined**.
5. Let *e* be *x*'s *endIndex*.
6. Let *xr* be the State (*e*, *cap*).
7. If *min* is not zero, then call *m(xr, d)* and return its result.
8. If *greedy* is **false**, then
  - a. Call *c(x)* and let *z* be its result.
  - b. If *z* is not **failure**, return *z*.
  - c. Call *m(xr, d)* and return its result.
9. Call *m(xr, d)* and let *z* be its result.
10. If *z* is not **failure**, return *z*.
11. Call *c(x)* and return its result.

**NOTE 1** An *Atom* followed by a *Quantifier* is repeated the number of times specified by the *Quantifier*. A *Quantifier* can be non-greedy, in which case the *Atom* pattern is repeated as few times as possible while still matching the sequel, or it can be greedy, in which case the *Atom* pattern is repeated as many times as possible while still matching the sequel. The *Atom* pattern is repeated rather than the input String that it matches, so different repetitions of the *Atom* can match different input substrings.

**NOTE 2** If the *Atom* and the sequel of the regular expression all have choice points, the *Atom* is first matched as many (or as few, if non-greedy) times as possible. All choices in the sequel are tried before moving on to the next choice in the last repetition of *Atom*. All choices in the last ( $n^{th}$ ) repetition of *Atom* are tried before moving on to the next choice in the next-to-last ( $n-1^{st}$ ) repetition of *Atom*; at which point it may turn out that more or fewer repetitions of *Atom* are now possible; these are exhausted (again, starting with either as few or as many as possible) before moving on to the next choice in the ( $n-1^{st}$ ) repetition of *Atom* and so on.

Compare

```
/a[a-z]{2,4}/.exec("abcdefghijkl")
```

which returns "abcde" with

```
/a[a-z]{2,4}?.exec("abcdefghijkl")
```

which returns "abc".

Consider also

```
/(aa|aabaac|ba|b|c)*/.exec("aabaac")
```

which, by the choice point ordering above, returns the array

```
["aaba", "ba"]
```

and not any of:

```
["aabaac", "aabaac"]
["aabaac", "c"]
```

The above ordering of choice points can be used to write a regular expression that calculates the greatest common divisor of two numbers (represented in unary notation). The following example calculates the gcd of 10 and 15:

```
"aaaaaaaaaaa,aaaaaaaaaaaaaa".replace(/^(\a+)\1*,\1+$/, "$1")
```

which returns the gcd in unary notation "aaaaa".

NOTE 3 Step 4 of the RepeatMatcher clears *Atom*'s captures each time *Atom* is repeated. We can see its behaviour in the regular expression

```
/(\z)((\a+)?(\b+)?(\c)*)*/.exec("zaacbbbcac")
```

which returns the array

```
["zaacbbbcac", "z", "ac", "a", undefined, "c"]
```

and not

```
["zaacbbbcac", "z", "ac", "a", "bbb", "c"]
```

because each iteration of the outermost **\*** clears all captured Strings contained in the quantified *Atom*, which in this case includes capture Strings numbered 2, 3, 4, and 5.

NOTE 4 Step 1 of the RepeatMatcher's *d* closure states that, once the minimum number of repetitions has been satisfied, any more expansions of *Atom* that match the empty String are not considered for further repetitions. This prevents the regular expression engine from falling into an infinite loop on patterns such as:

```
/(\a*)*/.exec("b")
```

or the slightly more complicated:

```
/(\a*)b\1+/.exec("baaaac")
```

which returns the array

```
["b", ""]
```

#### 15.10.2.6 Assertion

The production *Assertion* :: **^** evaluates by returning an internal AssertionTester closure that takes a State argument *x* and performs the following:

1. Let *e* be *x*'s *endIndex*.
2. If *e* is zero, return **true**.
3. If *Multiline* is **false**, return **false**.
4. If the character *Input*[*e*-1] is one of *LineTerminator*, return **true**.
5. Return **false**.

The production *Assertion* :: **\$** evaluates by returning an internal AssertionTester closure that takes a State argument *x* and performs the following:

1. Let *e* be *x*'s *endIndex*.
2. If *e* is equal to *InputLength*, return **true**.
3. If *multiline* is **false**, return **false**.
4. If the character *Input*[*e*] is one of *LineTerminator*, return **true**.
5. Return **false**.

The production *Assertion* :: **\b** evaluates by returning an internal AssertionTester closure that takes a State argument *x* and performs the following:

1. Let *e* be *x*'s *endIndex*.
2. Call *IsWordChar*(*e*-1) and let *a* be the Boolean result.
3. Call *IsWordChar*(*e*) and let *b* be the Boolean result.
4. If *a* is **true** and *b* is **false**, return **true**.
5. If *a* is **false** and *b* is **true**, return **true**.
6. Return **false**.

The production *Assertion* :: **\B** evaluates by returning an internal AssertionTester closure that takes a State argument *x* and performs the following:

1. Let *e* be *x*'s *endIndex*.

2. Call *IsWordChar*(*e*-1) and let *a* be the Boolean result.
3. Call *IsWordChar*(*e*) and let *b* be the Boolean result.
4. If *a* is **true** and *b* is **false**, return **false**.
5. If *a* is **false** and *b* is **true**, return **false**.
6. Return **true**.

The production *Assertion* :: ( ? = *Disjunction* ) evaluates as follows:

1. Evaluate *Disjunction* to obtain a Matcher *m*.
2. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following steps:
  - 1.Let *d* be a Continuation that always returns its State argument as a successful MatchResult.
  - 2.Call *m*(*x*, *d*) and let *r* be its result.
  - 3.If *r* is **failure**, return **failure**.
  - 4.Let *y* be *r*'s State.
  - 5.Let *cap* be *y*'s *captures* internal array.
  - 6.Let *xe* be *x*'s *endIndex*.
  - 7.Let *z* be the State (*xe*, *cap*).
  - 8.Call *c*(*z*) and return its result.

The production *Assertion* :: ( ? ! *Disjunction* ) evaluates as follows:

1. Evaluate *Disjunction* to obtain a Matcher *m*.
2. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following steps:
  - 1.Let *d* be a Continuation that always returns its State argument as a successful MatchResult.
  - 2.Call *m*(*x*, *d*) and let *r* be its result.
  - 3.If *r* isn't **failure**, return **failure**.
  - 4.Call *c*(*x*) and return its result.

The abstract operation *IsWordChar* takes an integer parameter *e* and performs the following:

1. If *e* == -1 or *e* == *InputLength*, return **false**.
2. Let *c* be the character *Input*[*e*].
3. If *c* is one of the sixty-three characters below, return **true**.
 

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9																
4. Return **false**.

#### 15.10.2.7 Quantifier

The production *Quantifier* :: *QuantifierPrefix* evaluates as follows:

1. Evaluate *QuantifierPrefix* to obtain the two results: an integer *min* and an integer (or  $\infty$ ) *max*.
2. Return the three results *min*, *max*, and **true**.

The production *Quantifier* :: *QuantifierPrefix* ? evaluates as follows:

1. Evaluate *QuantifierPrefix* to obtain the two results: an integer *min* and an integer (or  $\infty$ ) *max*.
2. Return the three results *min*, *max*, and **false**.

The production *QuantifierPrefix* :: \* evaluates by returning the two results 0 and  $\infty$ .

The production *QuantifierPrefix* :: + evaluates by returning the two results 1 and  $\infty$ .

The production *QuantifierPrefix* :: ? evaluates by returning the two results 0 and 1.

The production *QuantifierPrefix* :: { *DecimalDigits* } evaluates as follows:

1. Let *i* be the MV of *DecimalDigits* (see 7.8.3).
2. Return the two results *i* and *i*.

The production *QuantifierPrefix* :: { *DecimalDigits* , } evaluates as follows:

1. Let *i* be the MV of *DecimalDigits*.
2. Return the two results *i* and  $\infty$ .

The production *QuantifierPrefix* :: { *DecimalDigits* , *DecimalDigits* } evaluates as follows:

1. Let *i* be the MV of the first *DecimalDigits*.
2. Let *j* be the MV of the second *DecimalDigits*.
3. Return the two results *i* and *j*.

#### 15.10.2.8 Atom

The production *Atom* :: *PatternCharacter* evaluates as follows:

1. Let *ch* be the character represented by *PatternCharacter*.
2. Let *A* be a one-element CharSet containing the character *ch*.
3. Call *CharacterSetMatcher*(*A*, **false**) and return its Matcher result.

The production *Atom* :: . evaluates as follows:

1. Let *A* be the set of all characters except *LineTerminator*.
2. Call *CharacterSetMatcher*(*A*, **false**) and return its Matcher result.

The production *Atom* :: \ *AtomEscape* evaluates by evaluating *AtomEscape* to obtain a Matcher and returning that Matcher.

The production *Atom* :: *CharacterClass* evaluates as follows:

1. Evaluate *CharacterClass* to obtain a CharSet *A* and a Boolean *invert*.
2. Call *CharacterSetMatcher*(*A*, *invert*) and return its Matcher result.

The production *Atom* :: ( *Disjunction* ) evaluates as follows:

1. Evaluate *Disjunction* to obtain a Matcher *m*.
2. Let *parenIndex* be the number of left capturing parentheses in the entire regular expression that occur to the left of this production expansion's initial left parenthesis. This is the total number of times the *Atom* :: ( *Disjunction* ) production is expanded prior to this production's *Atom* plus the total number of *Atom* :: ( *Disjunction* ) productions enclosing this *Atom*.
3. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following steps:

1. Create an internal Continuation closure *d* that takes one State argument *y* and performs the following steps:
  1. Let *cap* be a fresh copy of *y*'s *captures* internal array.
  2. Let *xe* be *x*'s *endIndex*.
  3. Let *ye* be *y*'s *endIndex*.
  4. Let *s* be a fresh String whose characters are the characters of *Input* at positions *xe* (inclusive) through *ye* (exclusive).
  5. Set *cap*[*parenIndex*+1] to *s*.
  6. Let *z* be the State (*ye*, *cap*).
  7. Call *c(z)* and return its result.
2. Call *m(x, d)* and return its result.

The production *Atom :: ( ? : Disjunction )* evaluates by evaluating *Disjunction* to obtain a Matcher and returning that Matcher.

The abstract operation *CharacterSetMatcher* takes two arguments, a CharSet *A* and a Boolean flag *invert*, and performs the following:

1. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following steps:
  1. Let *e* be *x*'s *endIndex*.
  2. If *e* == *InputLength*, return **failure**.
  3. Let *ch* be the character *Input[e]*.
  4. Let *cc* be the result of *Canonicalize(ch)*.
  5. If *invert* is **false**, then
    - a. If there does not exist a member *a* of set *A* such that *Canonicalize(a) == cc*, return **failure**.
  6. Else *invert* is **true**,
    - a. If there exists a member *a* of set *A* such that *Canonicalize(a) == cc*, return **failure**.
7. Let *cap* be *x*'s *captures* internal array.
8. Let *y* be the State (*e+1, cap*).
9. Call *c(y)* and return its result.

The abstract operation *Canonicalize* takes a character parameter *ch* and performs the following steps:

1. If *IgnoreCase* is **false**, return *ch*.
2. Let *u* be *ch* converted to upper case as if by calling the standard built-in method **String.prototype.toUpperCase** on the one-character String *ch*.
3. If *u* does not consist of a single character, return *ch*.
4. Let *cu* be *u*'s character.
5. If *ch*'s code unit value is greater than or equal to decimal 128 and *cu*'s code unit value is less than decimal 128, then return *ch*.
6. Return *cu*.

**NOTE 1** Parentheses of the form *( Disjunction )* serve both to group the components of the *Disjunction* pattern together and to save the result of the match. The result can be used either in a backreference (\ followed by a nonzero decimal number), referenced in a replace String, or returned as part of an array from the regular expression matching internal procedure. To inhibit the capturing behaviour of parentheses, use the form *(?: Disjunction )* instead.

**NOTE 2** The form *(?= Disjunction )* specifies a zero-width positive lookahead. In order for it to succeed, the pattern inside *Disjunction* must match at the current position, but the current position is not advanced before matching the sequel. If *Disjunction* can match at the current position in several ways, only the first one is tried. Unlike other regular expression operators, there is no backtracking into a *(?=* form (this unusual behaviour is inherited from Perl). This only matters when the *Disjunction* contains capturing parentheses and the sequel of the pattern contains backreferences to those captures.

For example,

```
/(?=(a+))/ .exec("baaabac")
```

matches the empty String immediately after the first **b** and therefore returns the array:

```
[ "", "aaa" ]
```

To illustrate the lack of backtracking into the lookahead, consider:

```
/(?=(a+)a*b\1)/ .exec("baaabac")
```

This expression returns

```
[ "aba", "a" ]
```

and not:

```
[ "aaaba", "a" ]
```

**NOTE 3** The form *(?! Disjunction )* specifies a zero-width negative lookahead. In order for it to succeed, the pattern inside *Disjunction* must fail to match at the current position. The current position is not advanced before matching the sequel. *Disjunction* can contain capturing parentheses, but backreferences to them only make sense from within

*Disjunction* itself. Backreferences to these capturing parentheses from elsewhere in the pattern always return **undefined** because the negative lookahead must fail for the pattern to succeed. For example,

```
/(.*)?a(?!a+)b\2c)\2(.*)/.exec("baaabaaac")
```

looks for an **a** not immediately followed by some positive number *n* of **a**'s, a **b**, another *n* **a**'s (specified by the first **\2**) and a **c**. The second **\2** is outside the negative lookahead, so it matches against **undefined** and therefore always succeeds. The whole expression returns the array:

```
["baaabaaac", "ba", undefined, "abaac"]
```

In case-insensitive matches all characters are implicitly converted to upper case immediately before they are compared. However, if converting a character to upper case would expand that character into more than one character (such as converting "**B**" (**\u000DF**) into "**ss**"), then the character is left as-is instead. The character is also left as-is if it is not an ASCII character but converting it to upper case would make it into an ASCII character. This prevents Unicode characters such as **\u0131** and **\u017F** from matching regular expressions such as **/[a-z]/i**, which are only intended to match ASCII letters. Furthermore, if these conversions were allowed, then **/[^w]/i** would match each of **a**, **b**, ..., **h**, but not **i** or **s**.

### 15.10.2.9 AtomEscape

The production *AtomEscape* :: *DecimalEscape* evaluates as follows:

1. Evaluate *DecimalEscape* to obtain an *EscapeValue* *E*.
2. If *E* is a character, then
  - a. Let *ch* be *E*'s character.
  - b. Let *A* be a one-element CharSet containing the character *ch*.
  - c. Call *CharacterSetMatcher*(*A*, **false**) and return its Matcher result.
3. *E* must be an integer. Let *n* be that integer.
4. If *n*=0 or *n*>*NCapturingParens* then throw a **SyntaxError** exception.
5. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
  1. Let *cap* be *x*'s *captures* internal array.
  2. Let *s* be *cap*[*n*].
  3. If *s* is **undefined**, then call *c(x)* and return its result.
  4. Let *e* be *x*'s *endIndex*.
  5. Let *len* be *s*'s length.
  6. Let *f* be *e*+*len*.
  7. If *f*>*InputLength*, return **failure**.
  8. If there exists an integer *i* between 0 (inclusive) and *len* (exclusive) such that *Canonicalize*(*s*[*i*]) is not the same character as *Canonicalize*(*Input*[*e*+*i*]), then return **failure**.
  9. Let *y* be the State (*f*, *cap*).
  10. Call *c(y)* and return its result.

The production *AtomEscape* :: *CharacterEscape* evaluates as follows:

1. Evaluate *CharacterEscape* to obtain a character *ch*.
2. Let *A* be a one-element CharSet containing the character *ch*.
3. Call *CharacterSetMatcher*(*A*, **false**) and return its Matcher result.

The production *AtomEscape* :: *CharacterClassEscape* evaluates as follows:

1. Evaluate *CharacterClassEscape* to obtain a CharSet *A*.
2. Call *CharacterSetMatcher*(*A*, **false**) and return its Matcher result.

**NOTE** An escape sequence of the form **\** followed by a nonzero decimal number *n* matches the result of the *n*th set of capturing parentheses (see 15.10.2.11). It is an error if the regular expression has fewer than *n* capturing parentheses. If the regular expression has *n* or more capturing parentheses but the *n*th one is **undefined** because it has not captured anything, then the backreference always succeeds.

### 15.10.2.10 CharacterEscape

The production *CharacterEscape* :: *ControlEscape* evaluates by returning the character according to Table 25.

**Table 25 — ControlEscape Character Values**

<i>ControlEscape</i>	<i>Code Unit</i>	<i>Name</i>	<i>Symbol</i>
t	\u0009	horizontal tab	<HT>
n	\u000A	line feed (new line)	<LF>
v	\u000B	vertical tab	<VT>
f	\u000C	form feed	<FF>
r	\u000D	carriage return	<CR>

Deleted: 23

Deleted: 23

Formatted Table

The production *CharacterEscape* :: c *ControlLetter* evaluates as follows:

1. Let *ch* be the character represented by *ControlLetter*.
2. Let *i* be *ch*'s code unit value.
3. Let *j* be the remainder of dividing *i* by 32.
4. Return the character whose code unit value is *j*.

The production *CharacterEscape* :: *HexEscapeSequence* evaluates by evaluating the CV of the *HexEscapeSequence* (see 7.8.4) and returning its character result.

The production *CharacterEscape* :: *UnicodeEscapeSequence* evaluates by evaluating the CV of the *UnicodeEscapeSequence* (see 7.8.4) and returning its character result.

The production *CharacterEscape* :: *IdentityEscape* evaluates by returning the character represented by *IdentityEscape*.

### 15.10.2.11 DecimalEscape

The production *DecimalEscape* :: *DecimalIntegerLiteral* [lookahead ≠ *DecimalDigit*] evaluates as follows:

1. Let *i* be the MV of *DecimalIntegerLiteral*.
2. If *i* is zero, return the EscapeValue consisting of a <NUL> character (Unicode value 0000).
3. Return the EscapeValue consisting of the integer *i*.

The definition of “the MV of *DecimalIntegerLiteral*” is in 7.8.3.

NOTE If \ is followed by a decimal number *n* whose first digit is not 0, then the escape sequence is considered to be a backreference. It is an error if *n* is greater than the total number of left capturing parentheses in the entire regular expression. \0 represents the <NUL> character and cannot be followed by a decimal digit.

### 15.10.2.12 CharacterClassEscape

The production *CharacterClassEscape* :: d evaluates by returning the ten-element set of characters containing the characters 0 through 9 inclusive.

The production *CharacterClassEscape* :: D evaluates by returning the set of all characters not included in the set returned by *CharacterClassEscape* :: d.

The production *CharacterClassEscape* :: s evaluates by returning the set of characters containing the characters that are on the right-hand side of the *WhiteSpace* (7.2) or *LineTerminator* (7.3) productions.

The production *CharacterClassEscape* :: S evaluates by returning the set of all characters not included in the set returned by *CharacterClassEscape* :: s.

The production *CharacterClassEscape* :: **w** evaluates by returning the set of characters containing the sixty-three characters:

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	_															

The production *CharacterClassEscape* :: **w** evaluates by returning the set of all characters not included in the set returned by *CharacterClassEscape* :: **w**.

#### 15.10.2.13 CharacterClass

The production *CharacterClass* :: **[** [lookahead ≠ **{}**] *ClassRanges* **]** evaluates by evaluating *ClassRanges* to obtain a CharSet and returning that CharSet and the Boolean **false**.

The production *CharacterClass* :: **[** ^ *ClassRanges* **]** evaluates by evaluating *ClassRanges* to obtain a CharSet and returning that CharSet and the Boolean **true**.

#### 15.10.2.14 ClassRanges

The production *ClassRanges* :: [empty] evaluates by returning the empty CharSet.

The production *ClassRanges* :: *NonemptyClassRanges* evaluates by evaluating *NonemptyClassRanges* to obtain a CharSet and returning that CharSet.

#### 15.10.2.15 NonemptyClassRanges

The production *NonemptyClassRanges* :: *ClassAtom* evaluates by evaluating *ClassAtom* to obtain a CharSet and returning that CharSet.

The production *NonemptyClassRanges* :: *ClassAtom NonemptyClassRangesNoDash* evaluates as follows:

1. Evaluate *ClassAtom* to obtain a CharSet *A*.
2. Evaluate *NonemptyClassRangesNoDash* to obtain a CharSet *B*.
3. Return the union of CharSets *A* and *B*.

The production *NonemptyClassRanges* :: *ClassAtom - ClassAtom ClassRanges* evaluates as follows:

1. Evaluate the first *ClassAtom* to obtain a CharSet *A*.
2. Evaluate the second *ClassAtom* to obtain a CharSet *B*.
3. Evaluate *ClassRanges* to obtain a CharSet *C*.
4. Call *CharacterRange(A, B)* and let *D* be the resulting CharSet.
5. Return the union of CharSets *D* and *C*.

The abstract operation *CharacterRange* takes two CharSet parameters *A* and *B* and performs the following:

1. If *A* does not contain exactly one character or *B* does not contain exactly one character then throw a **SyntaxError** exception.
2. Let *a* be the one character in CharSet *A*.
3. Let *b* be the one character in CharSet *B*.
4. Let *i* be the code unit value of character *a*.
5. Let *j* be the code unit value of character *b*.
6. If *i > j* then throw a **SyntaxError** exception.
7. Return the set containing all characters numbered *i* through *j*, inclusive.

#### 15.10.2.16 NonemptyClassRangesNoDash

The production *NonemptyClassRangesNoDash* :: *ClassAtom* evaluates by evaluating *ClassAtom* to obtain a CharSet and returning that CharSet.

The production *NonemptyClassRangesNoDash* :: *ClassAtomNoDash* *NonemptyClassRangesNoDash* evaluates as follows:

1. Evaluate *ClassAtomNoDash* to obtain a CharSet *A*.
2. Evaluate *NonemptyClassRangesNoDash* to obtain a CharSet *B*.
3. Return the union of CharSets *A* and *B*.

The production *NonemptyClassRangesNoDash* :: *ClassAtomNoDash* – *ClassAtom* *ClassRanges* evaluates as follows:

1. Evaluate *ClassAtomNoDash* to obtain a CharSet *A*.
2. Evaluate *ClassAtom* to obtain a CharSet *B*.
3. Evaluate *ClassRanges* to obtain a CharSet *C*.
4. Call *CharacterRange*(*A*, *B*) and let *D* be the resulting CharSet.
5. Return the union of CharSets *D* and *C*.

**NOTE 1** *ClassRanges* can expand into single *ClassAtoms* and/or ranges of two *ClassAtoms* separated by dashes. In the latter case the *ClassRanges* includes all characters between the first *ClassAtom* and the second *ClassAtom*, inclusive; an error occurs if either *ClassAtom* does not represent a single character (for example, if one is \w) or if the first *ClassAtom*'s code unit value is greater than the second *ClassAtom*'s code unit value.

**NOTE 2** Even if the pattern ignores case, the case of the two ends of a range is significant in determining which characters belong to the range. Thus, for example, the pattern /[\E-\F]/i matches only the letters E, F, e, and f, while the pattern /[\E-\f]/i matches all upper and lower-case ASCII letters as well as the symbols [, \, ], ^, \_, and `.

**NOTE 3** A – character can be treated literally or it can denote a range. It is treated literally if it is the first or last character of *ClassRanges*, the beginning or end limit of a range specification, or immediately follows a range specification.

#### 15.10.2.17 ClassAtom

The production *ClassAtom* :: – evaluates by returning the CharSet containing the one character –.

The production *ClassAtom* :: *ClassAtomNoDash* evaluates by evaluating *ClassAtomNoDash* to obtain a CharSet and returning that CharSet.

#### 15.10.2.18 ClassAtomNoDash

The production *ClassAtomNoDash* :: *SourceCharacter* **but not one of \ or ] or –** evaluates by returning a one-element CharSet containing the character represented by *SourceCharacter*.

The production *ClassAtomNoDash* :: \ *ClassEscape* evaluates by evaluating *ClassEscape* to obtain a CharSet and returning that CharSet.

#### 15.10.2.19 ClassEscape

The production *ClassEscape* :: *DecimalEscape* evaluates as follows:

1. Evaluate *DecimalEscape* to obtain an EscapeValue *E*.
2. If *E* is not a character then throw a **SyntaxError** exception.
3. Let *ch* be *E*'s character.
4. Return the one-element CharSet containing the character *ch*.

The production *ClassEscape* :: \b evaluates by returning the CharSet containing the one character <BS> (Unicode value 0008).

The production *ClassEscape* :: *CharacterEscape* evaluates by evaluating *CharacterEscape* to obtain a character and returning a one-element CharSet containing that character.

The production *ClassEscape* :: *CharacterClassEscape* evaluates by evaluating *CharacterClassEscape* to obtain a CharSet and returning that CharSet.

**NOTE** A *ClassAtom* can use any of the escape sequences that are allowed in the rest of the regular expression except for `\b`, `\B`, and backreferences. Inside a *CharacterClass*, `\b` means the backspace character, while `\B` and backreferences raise errors. Using a backreference inside a *ClassAtom* causes an error.

### 15.10.3 The RegExp Constructor Called as a Function

#### 15.10.3.1 RegExp(pattern, flags)

The following steps are taken:

1. If `Type(pattern)` is object and `pattern` has a `[[NativeBrand]]` internal property whose value is `NativeRegExp` and `flags` is `undefined`, then return `pattern`.
2. Return the result of the abstraction operation `RegExpCreate` with arguments `pattern` and `flags`.

### 15.10.4 The RegExp Constructor

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

#### 15.10.4.1 new RegExp(pattern, flags)

The following steps are taken:

1. Return the result of the abstraction operation `RegExpCreate` with arguments `pattern` and `flags`.

The abstract operation `RegExpCreate` with arguments `pattern` and `flags` does the following:

If `pattern` is an object `R` that has a `[[NativeBrand]]` internal property whose value is `NativeRegExp` and `flags` is `undefined`, then let `P` be the `pattern` used to construct `R` and let `F` be the `flags` used to construct `R`. If `pattern` is an object `R` that has a `[[NativeBrand]]` internal property whose value is `NativeRegExp` and `flags` is not `undefined`, then throw a `TypeError` exception. Otherwise, let `P` be the empty String if `pattern` is `undefined` and `Tostring(pattern)` otherwise, and let `F` be the empty String if `flags` is `undefined` and `Tostring(flags)` otherwise.

If the characters of `P` do not have the syntactic form `Pattern`, then throw a `SyntaxError` exception. Otherwise let the newly constructed object have a `[[Match]]` internal property obtained by evaluating ("compiling") the characters of `P` as a `Pattern` as described in 15.10.2.

If `F` contains any character other than "g", "i", or "m", or if it contains the same character more than once, then throw a `SyntaxError` exception.

If a `SyntaxError` exception is not thrown, then:

Let `S` be a String in the form of a `Pattern` equivalent to `P`, in which certain characters are escaped as described below. `S` may or may not be identical to `P` or `pattern`; however, the internal procedure that would result from evaluating `S` as a `Pattern` must behave identically to the internal procedure given by the constructed object's `[[Match]]` internal property.

The characters / occurring in the pattern shall be escaped in `S` as necessary to ensure that the String value formed by concatenating the Strings "/", `S`, "/", and `F` can be parsed (in an appropriate lexical context) as a `RegularExpressionLiteral` that behaves identically to the constructed regular expression. For example, if `P` is "/", then `S` could be "\/" or "\u002F", among other possibilities, but not "/", because /// followed by `F` would be parsed as a `SingleLineComment` rather than a `RegularExpressionLiteral`. If `P` is the empty String, this specification can be met by letting `S` be "(?:)".

**Deleted:** If `pattern` is an object `R` whose that has an `[[ClassIsRegExpNativeBrand]]` internal property whose value is `NativeRegExp` is "RegExp" and `flags` is `undefined`, then return unchanged. Otherwise call the standard built-in `RegExp` constructor (15.10.4.1) as if by the expression `new RegExp(pattern, flags)` and return the object constructed by that constructor.

**Formatted:** Outline numbered + Level: 1 + Numbering Style: 1, 2, 3, ... + Start at: 1 + Alignment: Left + Aligned at: 0" + Tab after: 0.25" + Indent at: 0.25"

**Deleted:** n

**Deleted:** IsRegExp

**Deleted:** whose `[[Class]]` internal property is "RegExp"

**Deleted:** n

**Deleted:** IsRegExp

**Deleted:** whose `[[Class]]` internal property is "RegExp"

The following properties of the newly constructed object are data properties with the attributes that are specified in 15.10.7. The `[[Value]]` of each property is set as follows:

The `source` property of the newly constructed object is set to `S`.

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

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

The property of the newly constructed object is set to a Boolean value that is `true` if `F` contains the character "m" and `false` otherwise.

The `lastIndex` property of the newly constructed object is set to `0`.

The `[[Prototype]]` internal property of the newly constructed object is set to the standard built-in `RegExp` prototype object as specified in 15.10.6.

The newly constructed object has a `[[NativeBrand]]` internal property whose value is `NativeRegExp`.

**NOTE** If pattern is a *StringLiteral*, the usual escape sequence substitutions are performed before the String is processed by `RegExp`. If pattern must contain an escape sequence to be recognised by `RegExp`, any backslash \ characters must be escaped within the *StringLiteral* to prevent them being removed when the contents of the *StringLiteral* are formed.

## 15.10.5 Properties of the `RegExp` Constructor

The value of the `[[Prototype]]` internal property of the `RegExp` constructor is the standard built-in `Function` prototype object (15.3.4).

Besides the internal properties and the `length` property (whose value is `2`), the `RegExp` constructor has the following properties:

### 15.10.5.1 `RegExp.prototype`

The initial value of `RegExp.prototype` is the `RegExp` prototype object (15.10.6).

This property shall have the attributes { `[[Writable]]: false`, `[[Enumerable]]: false`, `[[Configurable]]: false` }.

## 15.10.6 Properties of the `RegExp` Prototype Object

The value of the `[[Prototype]]` internal property of the `RegExp` prototype object is the standard built-in `Object` prototype object (15.2.4). The `RegExp` prototype object is itself a regular expression object; it has a `[[NativeBrand]]` internal property whose value is `NativeRegExp`. The initial values of the `RegExp` prototype object's data properties (15.10.7) are set as if the object was created by the expression `new RegExp()` where `RegExp` is that standard built-in constructor with that name.

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

In the following descriptions of functions that are properties of the `RegExp` prototype object, the phrase "this `RegExp` object" refers to the object that is the `this` value for the invocation of the function; a `TypeError` exception is thrown if the `this` value is not an object that has a `[[NativeBrand]]` internal property whose value is `NativeRegExp`.

**Deleted:** `[[Class]]` internal property of the  
**Deleted:** `n`  
**Deleted:** `IsRegExp`  
**Deleted:** is set to "`RegExp`".

**Deleted:** `s`  
**Deleted:** `n`  
**Deleted:** `IsRegExp`  
**Deleted:** `[[Class]]` is "`RegExp`"

**Deleted:** or an object for which the value of the  
**Deleted:** `n`  
**Deleted:** `Class`  
**Deleted:** `IsRegExp`  
**Deleted:** is not "`RegExp`"

### 15.10.6.1 RegExp.prototype.constructor

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

### 15.10.6.2 RegExp.prototype.exec(string)

Performs a regular expression match of `string` against the regular expression and returns an Array object containing the results of the match, or `null` if `string` did not match.

The String `ToString(string)` is searched for an occurrence of the regular expression pattern as follows:

1. Let `R` be this `RegExp` object.
2. `ReturnIfAbrupt(R).`
3. Let `S` be the value of `ToString(string)`.
4. `ReturnIfAbrupt(S).`
5. `Return the result of the RegExpExec abstraction operation with arguments R and S.`

The abstract operation `RegExpExec` with arguments `R` (an object) and `S` (a string) performs the following steps:

1. Let `length` be the length of `S`.
2. Let `lastIndex` be the result of calling the `[[Get]]` internal method of `R` with argument `"lastIndex"`.
3. Let `i` be the value of `ToInteger(lastIndex)`.
4. `ReturnIfAbrupt(i).`
5. Let `global` be the result of calling the `[[Get]]` internal method of `R` with argument `"global"`.
6. `ReturnIfAbrupt(global).`
7. If `global` is `false`, then let `i = 0`.
8. Let `matchSucceeded` be `false`.
9. Repeat, while `matchSucceeded` is `false`
  - a. If `i < 0` or `i > length`, then
    - i. Let `putStatus` be the result of calling the `[[Put]]` internal method of `R` with arguments `"lastIndex"`, `0`, and `true`.
    - ii. ReturnIfAbrupt(putStatus).
    - iii. Return `null`.
  - b. Call the `[[Match]]` internal method of `R` with arguments `S` and `i`.
  - c. If `[[Match]]` returned `failure`, then
    - i. Let `i = i + 1`.
  - d. else
    - i. Let `r` be the State result of the call to `[[Match]]`.
    - ii. Set `matchSucceeded` to `true`.
10. Let `e` be `r`'s `endIndex` value.
11. If `global` is `true`,
  - a. Let `putStatus` be the result of calling the `[[Put]]` internal method of `R` with arguments `"lastIndex"`, `e`, and `true`.
  - b. ReturnIfAbrupt(putStatus).
12. Let `n` be the length of `r`'s `captures` array. (This is the same value as 15.10.2.1's `NCapturingParens`.)
13. Let `A` be the result of the abstract operation `ArrayCreate` with argument `0`.
14. Let `matchIndex` be `i`.
15. Assert: The following `[[DefineOwnProperty]]` calls will not result in an abrupt completion.
16. Call the `[[DefineOwnProperty]]` internal method of `A` with arguments `"index"`, Property Descriptor `{[[Value]]: matchIndex, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, and `true`.
17. Call the `[[DefineOwnProperty]]` internal method of `A` with arguments `"input"`, Property Descriptor `{[[Value]]: S, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, and `true`.
18. Call the `[[DefineOwnProperty]]` internal method of `A` with arguments `"length"`, Property Descriptor `{[[Value]]: n + 1}`, and `true`.
19. Let `matchedSubstr` be the matched substring (i.e. the portion of `S` between offset `i` inclusive and offset `e` exclusive).
20. Call the `[[DefineOwnProperty]]` internal method of `A` with arguments `"0"`, Property Descriptor `{[[Value]]: matchedSubstr, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, and `true`.

**Deleted:** Call

**Deleted:** Call

**Deleted:** a new array created as if by the expression `new Array()` where `Array` is the standard built-in constructor with that name.

21. For each integer  $i$  such that  $i > 0$  and  $i \leq n$ 
  - a. Let  $captureI$  be  $i^{\text{th}}$  element of  $r$ 's *captures* array.
  - b. Call the `[[DefineOwnProperty]]` internal method of  $A$  with arguments `ToString(i)`, *Property Descriptor* `{[[Value]]: captureI, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, and `true`.
22. Return  $A$ .

#### 15.10.6.3 `RegExp.prototype.test(string)`

The following steps are taken:

1. Let  $R$  be this `RegExp` object.
2. ReturnIfAbrupt( $R$ ).
3. Let  $S$  be the value of `ToString(string)`.
4. ReturnIfAbrupt( $S$ ).
5. Let  $match$  be the result of the `RegExpExec` abstraction operation with arguments  $R$  and  $S$ .
6. ReturnIfAbrupt( $match$ ).
7. If  $match$  is not null, then return `true`; else return `false`.

#### 15.10.6.4 `RegExp.prototype.toString()`

Return the String value formed by concatenating the Strings `"/"`, the String value of the `source` property of this `RegExp` object, and `"/"`; plus `"g"` if the `global` property is `true`, `"i"` if the `ignoreCase` property is `true`, and `"m"` if the `multiline` property is `true`.

**NOTE** The returned String has the form of a *RegularExpressionLiteral* that evaluates to another `RegExp` object with the same behaviour as this object.

### 15.10.7 Properties of `RegExp` Instances

`RegExp` instances inherit properties from the `RegExp` prototype object and have a `[[NativeBrand]]` internal property whose value is `NativeRegExp`. `RegExp` instances also have a `[[Match]]` internal property and a `length` property.

The value of the `[[Match]]` internal property is an implementation dependent representation of the *Pattern* of the `RegExp` object.

`RegExp` instances also have the following properties.

#### 15.10.7.1 `source`

The value of the `source` property is a String in the form of a *Pattern* representing the current regular expression. This property shall have the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }`.

#### 15.10.7.2 `global`

The value of the `global` property is a Boolean value indicating whether the flags contained the character `"g"`. This property shall have the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }`.

#### 15.10.7.3 `ignoreCase`

The value of the `ignoreCase` property is a Boolean value indicating whether the flags contained the character `"i"`. This property shall have the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }`.

**Deleted:** evaluating the `RegExp.prototype.exec` (15.10.6.2) algorithm upon this `RegExp` object using `string` as the argument

**Comment [AWB7104]:** Needs an algorithm

**Deleted:** their  
**Deleted:** Class  
**Deleted:** IsRegExp  
**Deleted:** value is "RegExp"

#### 15.10.7.4 multiline

The value of the `multiline` property is a Boolean value indicating whether the flags contained the character "m". This property shall have the attributes { [[Writable]]: `false`, [[Enumerable]]: `false`, [[Configurable]]: `false` }.

#### 15.10.7.5 lastIndex

The value of the `lastIndex` property specifies the String position at which to start the next match. It is coerced to an integer when used (see 15.10.6.2). This property shall have the attributes { [[Writable]]: `true`, [[Enumerable]]: `false`, [[Configurable]]: `false` }.

**NOTE** Unlike the other standard built-in properties of RegExp instances, `lastIndex` is writable.

### 15.11 Error Objects

Instances of Error objects are thrown as exceptions when runtime errors occur. The Error objects may also serve as base objects for user-defined exception classes.

#### 15.11.1 The Error Constructor Called as a Function

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

##### 15.11.1.1 Error (message)

The [[Prototype]] internal property of the newly constructed object is set to the original Error prototype object, the one that is the initial value of `Error.prototype` (15.11.3.1).

The newly constructed object has a [[NativeBrand]] internal property whose value is NativeError.

The [[Extensible]] internal property of the newly constructed object is set to `true`.

If the argument `message` is not `undefined`, the `message` own property of the newly constructed object is set to `ToString(message)`.

- Deleted:** [[Class]] internal property of the
- Deleted:** is set to "Error"
- Deleted:** n
- Deleted:** IsError

#### 15.11.2 The Error Constructor

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

##### 15.11.2.1 new Error (message)

The [[Prototype]] internal property of the newly constructed object is set to the original Error prototype object, the one that is the initial value of `Error.prototype` (15.11.3.1).

The newly constructed object has a [[NativeBrand]] internal property whose value is NativeError.

The [[Extensible]] internal property of the newly constructed object is set to `true`.

If the argument `message` is not `undefined`, the `message` own property of the newly constructed object is set to `ToString(message)`.

- Deleted:** [[Class]] internal property of the
- Deleted:** Error
- Deleted:** n
- Deleted:** IsError
- Deleted:** is set to "Error"

#### 15.11.3 Properties of the Error Constructor

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

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

### 15.11.3.1 Error.prototype

The initial value of `Error.prototype` is the Error prototype object (15.11.4).

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

## 15.11.4 Properties of the Error Prototype Object

The Error prototype object is itself an Error object and has a `[[NativeBrand]]` internal property whose value is `NativeError`.

The value of the `[[Prototype]]` internal property of the Error prototype object is the standard built-in Object prototype object (15.2.4).

**Deleted:** `n`

**Deleted:** `IsError`

**Deleted:** (its `[[Class]]` is "Error")

### 15.11.4.1 Error.prototype.constructor

The initial value of `Error.prototype.constructor` is the built-in `Error` constructor.

### 15.11.4.2 Error.prototype.name

The initial value of `Error.prototype.name` is "Error".

### 15.11.4.3 Error.prototype.message

The initial value of `Error.prototype.message` is the empty String.

### 15.11.4.4 Error.prototype.toString ( )

The following steps are taken:

1. Let `O` be the `this` value.
2. If `Type(O)` is not `Object`, throw a `TypeError` exception.
3. Let `name` be the result of calling the `[[Get]]` internal method of `O` with argument "`name`".
4. `ReturnIfAbrupt(name)`.
5. If `name` is `undefined`, then let `name` be "Error"; else let `name` be `ToString(name)`.
6. Let `msg` be the result of calling the `[[Get]]` internal method of `O` with argument "`message`".
7. `ReturnIfAbrupt(msg)`.
8. If `msg` is `undefined`, then let `msg` be the empty String; else let `msg` be `ToString(msg)`.
9. If `name` is the empty String, return `msg`.
10. If `msg` is the empty String, return `name`.
11. Return the result of concatenating `name`, ":" , a single space character, and `msg`.

**Deleted:** <#> If `msg` is `undefined`, then let `msg` be the empty String; else let `msg` be `ToString(msg)`. ¶

## 15.11.5 Properties of Error Instances

Error instances inherit properties from the Error prototype object and have a `[[NativeBrand]]` internal property whose value is `NativeError`. Error instances have no special properties.

**Deleted:** `n`

**Deleted:** `IsError`

**Deleted:** their `[[Class]]` internal property value is "Error"

## 15.11.6 Native Error Types Used in This Standard

One of the `NativeError` objects below is thrown when a runtime error is detected. All of these objects share the same structure, as described in 15.11.7.

### 15.11.6.1 EvalError

This exception is not currently used within this specification. This object remains for compatibility with previous editions of this specification.

### 15.11.6.2 RangeError

Indicates a numeric value has exceeded the allowable range. See 15.4.2.2, 15.4.5.1, 15.7.4.2, 15.7.4.5, 15.7.4.6, 15.7.4.7, and 15.9.5.43.

### 15.11.6.3 ReferenceError

Indicate that an invalid reference value has been detected. See 8.9.1, 8.9.2, 10.2.1, 10.2.1.1.4, 10.2.1.2.4, and 11.13.1.

**Deleted:** 7

**Deleted:** 7

### 15.11.6.4 SyntaxError

Indicates that a parsing error has occurred. See 11.1.5, 11.3.1, 11.3.2, 11.4.1, 11.4.4, 11.4.5, 11.13.1, 11.13.2, 12.2.1, 12.10.1, 12.14.1, 13.1, 15.1.2.1, 15.3.2.1, 15.10.2.2, 15.10.2.5, 15.10.2.9, 15.10.2.15, 15.10.2.19, 15.10.4.1, and 15.12.2.

### 15.11.6.5 TypeError

Indicates the actual type of an operand is different than the expected type. See 8.6.2, 8.9.2, 8.10.5, 8.12.5, 8.12.7, 8.12.8, 8.12.9, 9.9, 9.10, 10.2.1, 10.2.1.1.3, 10.6, 11.2.2, 11.2.3, 11.4.1, 11.8.6, 11.8.7, 11.3.1, 13.2, 13.2.3, 15, 15.2.3.2, 15.2.3.3, 15.2.3.4, 15.2.3.5, 15.2.3.6, 15.2.3.7, 15.2.3.8, 15.2.3.9, 15.2.3.10, 15.2.3.11, 15.2.3.12, 15.2.3.13, 15.2.3.14, 15.2.4.3, 15.3.4.2, 15.3.4.3, 15.3.4.4, 15.3.4.5, 15.3.4.5.2, 15.3.4.5.3, 15.3.5, 15.3.5.3, 15.3.5.4, 15.4.4.3, 15.4.4.11, 15.4.4.16, 15.4.4.17, 15.4.4.18, 15.4.4.19, 15.4.4.20, 15.4.4.21, 15.4.4.22, 15.4.5.1, 15.5.4.2, 15.5.4.3, 15.6.4.2, 15.6.4.3, 15.7.4, 15.7.4.2, 15.7.4.4, 15.9.5, 15.9.5.44, 15.10.4.1, 15.10.6, 15.11.4.4 and 15.12.3.

**Deleted:** 7

### 15.11.6.6 URIError

Indicates that one of the global URI handling functions was used in a way that is incompatible with its definition. See 15.1.3.

### 15.11.7 NativeError Object Structure

When an ECMAScript implementation detects a runtime error, it throws an instance of one of the *NativeError* objects defined in 15.11.6. Each of these objects has the structure described below, differing only in the name used as the constructor name instead of *NativeError*, in the **name** property of the prototype object, and in the implementation-defined **message** property of the prototype object.

For each error object, references to *NativeError* in the definition should be replaced with the appropriate error object name from 15.11.6.

#### 15.11.7.1 NativeError Constructors Called as Functions

When a *NativeError* constructor is called as a function rather than as a constructor, it creates and initialises a new object. A call of the object as a function is equivalent to calling it as a constructor with the same arguments.

#### 15.11.7.2 NativeError (message)

The **[[Prototype]]** internal property of the newly constructed object is set to the prototype object for this error constructor. The **[[newTarget]]** internal property whose value is *NativeError*. The **[[Extensible]]** internal property of the newly constructed object is set to **true**.

**Deleted:** [[Class]] internal property of the

**Deleted:** n

**Deleted:** IsError

**Deleted:** is set to "Error"

### 15.11.7.3 The NativeError Constructors

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

#### 15.11.7.4 new NativeError (message)

The `[[Prototype]]` internal property of the newly constructed object is set to the prototype object for this *NativeError* constructor. The newly constructed object has a `[[NativeBrand]]` internal property whose value is `NativeError`. The `[[Extensible]]` internal property of the newly constructed object is set to `true`.

If the argument `message` is not `undefined`, the `message` own property of the newly constructed object is set to `ToString(message)`.

#### 15.11.7.5 Properties of the NativeError Constructors

The value of the `[[Prototype]]` internal property of a *NativeError* constructor is the Function prototype object (15.3.4).

Besides the internal properties and the `length` property (whose value is `1`), each *NativeError* constructor has the following property:

#### 15.11.7.6 NativeError.prototype

The initial value of `NativeError.prototype` is a *NativeError* prototype object (15.11.7.7). Each *NativeError* constructor has a separate prototype object.

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

#### 15.11.7.7 Properties of the NativeError Prototype Objects

Each *NativeError* prototype object is an Error object and has a `[[NativeBrand]]` internal property whose value is `NativeError`.

The value of the `[[Prototype]]` internal property of each *NativeError* prototype object is the standard built-in Error prototype object (15.11.4).

#### 15.11.7.8 NativeError.prototype.constructor

The initial value of the `constructor` property of the prototype for a given *NativeError* constructor is the *NativeError* constructor function itself (15.11.7).

#### 15.11.7.9 NativeError.prototype.name

The initial value of the `name` property of the prototype for a given *NativeError* constructor is the name of the constructor (the name used instead of *NativeError*).

#### 15.11.7.10 NativeError.prototype.message

The initial value of the `message` property of the prototype for a given *NativeError* constructor is the empty String.

**NOTE** The prototypes for the *NativeError* constructors do not themselves provide a `toString` function, but instances of errors will inherit it from the Error prototype object.

**Deleted:** `[[Class]]` internal property of the  
**Deleted:** `n`  
**Deleted:** `IsError`  
**Deleted:** is set to "Error"

### 15.11.7.11 Properties of NativeError Instances

*NativeError instances inherit properties from their NativeError prototype object and have a [[NativeBrand]] internal property whose value is NativeError.* NativeError instances have no special properties.

**Deleted:** n  
**Deleted:** IsError

**Deleted:** their [[Class]] internal property value is "Error"

### 15.12 The JSON Object

The **JSON** object is a single object that contains two functions, **parse** and **stringify**, that are used to parse and construct JSON texts. The JSON Data Interchange Format is described in RFC 4627 <<http://www.ietf.org/rfc/rfc4627.txt>>. The JSON interchange format used in this specification is exactly that described by RFC 4627 with two exceptions:

- The top level **JSONText** production of the ECMAScript JSON grammar may consist of any **JSONValue** rather than being restricted to being a **JSONObject** or a **JSONArray** as specified by RFC 4627.
- Conforming implementations of **JSON.parse** and **JSON.stringify** must support the exact interchange format described in this specification without any deletions or extensions to the format. This differs from RFC 4627 which permits a JSON parser to accept non-JSON forms and extensions.

The value of the **[[Prototype]]** internal property of the **JSON** object is the standard built-in Object prototype object (15.2.4). The **JSON** object has a **[[NativeBrand]] internal property whose value is NativeJSON**. The value of the **[[Extensible]]** internal property of the **JSON** object is set to **true**.

**Deleted:** value of the [[Class]] internal property of the  
**Deleted:** n  
**Deleted:** IsJSON  
**Deleted:** is "JSON"

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

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

#### 15.12.1 The JSON Grammar

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

##### 15.12.1.1 The JSON Lexical Grammar

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

##### Syntax

```

JSONWhiteSpace ::=
    <TAB>
    <CR>
    <LF>
    <SP>

JSONString ::=
    " JSONStringCharactersopt "

JSONStringCharacters ::=
    JSONStringCharacter JSONStringCharactersopt

JSONStringCharacter ::=

```

*SourceCharacter but not one of " or \ or U+0000 through U+001F  
 \JSONEscapeSequence*

*JSONEscapeSequence ::  
 JSONEscapeCharacter  
 UnicodeEscapeSequence*

*JSONEscapeCharacter :: one of  
 " / \ b f n r t*

*JSONNumber ::  
 -opt DecimalIntegerLiteral JSONFraction<sub>opt</sub> ExponentPart<sub>opt</sub>*

*JSONFraction ::  
 . DecimalDigits*

*JSONNullLiteral ::  
 NullLiteral*

*JSONBooleanLiteral ::  
 BooleanLiteral*

#### 15.12.1.2 The JSON Syntactic Grammar

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

##### Syntax

*JSONText :*  
*JSONValue*

*JSONValue :*  
*JSONNullLiteral*  
*JSONBooleanLiteral*  
*JSONObject*  
*JSONArray*  
*JSONString*  
*JSONNumber*

*JSONObject :*  
 { }  
 { *JSONMemberList* }

*JSONMember :*  
*JSONString* : *JSONValue*

*JSONMemberList :*  
*JSONMember*  
*JSONMemberList* , *JSONMember*

*JSONArray :*  
 [ ]  
 [ *JSONElementList* ]

*JSONElementList :*  
*JSONValue*  
*JSONElementList* , *JSONValue*

### 15.12.2 `parse ( text [, reviver ] )`

The `parse` function parses a JSON text (a JSON-formatted String) and produces an ECMAScript value. The JSON format is a restricted form of ECMAScript literal. JSON objects are realized as ECMAScript objects. JSON arrays are realized as ECMAScript arrays. JSON strings, numbers, booleans, and null are realized as ECMAScript Strings, Numbers, Booleans, and `null`. JSON uses a more limited set of white space characters than `WhiteSpace` and allows Unicode code points U+2028 and U+2029 to directly appear in `JSONString` literals without using an escape sequence. The process of parsing is similar to 11.1.4 and 11.1.5 as constrained by the JSON grammar.

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

1. Let `JText` be `ToString(text)`.
2. `ReturnIfAbrupt(text)`.
3. Parse `JText` using the grammars in 15.12.1. Throw a `SyntaxError` exception if `JText` did not conform to the JSON grammar for the goal symbol `JSONText`.
4. Let `unfiltered` be the result of parsing and evaluating `JText` as if it was the source text of an ECMAScript `Program` but using `JSONString` in place of `StringLiteral`. Note that since `JText` conforms to the JSON grammar this result will be either a primitive value or an object that is defined by either an `ArrayLiteral` or an `ObjectLiteral`.
5. If `IsCallable(reviver)` is `true`, then
  - a. Let `root` be the result of the abstraction operation `ObjectCreate (15.2)`.
  - b. Call the `[[DefineOwnProperty]]` internal method of `root` with the empty String, the PropertyDescriptor `{[[Value]]: unfiltered, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, and `false` as arguments.
  - c. Return the result of calling the abstract operation `Walk`, passing `root` and the empty String. The abstract operation `Walk` is described below.
6. Else
  - a. Return `unfiltered`.

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

1. Let `val` be the result of calling the `[[Get]]` internal method of `holder` with argument `name`.
2. `ReturnIfAbrupt(val)`.
3. If `val` is an object, then
  - a. If `val` has a `[[NativeBrand]]` internal property with value `NativeArray`, then
    - i. Set `I` to 0.
    - ii. Let `len` be the result of calling the `[[Get]]` internal method of `val` with argument `"length"`.
    - iii. Assert: len is not an abrupt completion and its value is a positive integer.
    - iv. Repeat while `I < len`,
      1. Let `newElement` be the result of calling the abstract operation `Walk`, passing `val` and `ToString(I)`.
      2. If `newElement` is `undefined`, then
        - a. Let `status` be the result of calling the `[[Delete]]` internal method of `val` with `ToString(I)` and `false` as arguments.
      3. Else
        - a. Let `status` be the result of calling the `[[DefineOwnProperty]]` internal method of `val` with arguments `ToString(I)`, the Property Descriptor `{[[Value]]: newElement, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`, and `false`.
      4. ReturnIfAbrupt(status).
      5. Add 1 to `I`.
    - b. Else

**Deleted:** a new object created as if by the expression `new Object()`, where `Object` is standard built-in constructor with that name

**Deleted:** the `[[Class]]` internal property of  
**Deleted:** is "Array"  
**Deleted:** isArray

**Deleted:** Call

**Deleted:** Call

- i. Let *keys* be an internal List of String values consisting of the names of all the own properties of *val* whose *[[Enumerable]]* attribute is **true**. The ordering of the Strings *is the same as that used by the **Object.keys** standard built-in function.*
  - ii. For each String *P* in *keys* do,
    1. Let *newElement* be the result of calling the abstract operation *Walk*, passing *val* and *P*.
    2. If *newElement* is **undefined**, then
      - a. *Let status be the result of calling the [[Delete]] internal method of val with P and false as arguments.*
    3. Else
      - a. *Let status be the result of calling the [[DefineOwnProperty]] internal method of val with arguments P, the Property Descriptor {[[Value]]: newElement, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}, and false.*
  4. **4. ReturnIfAbrupt(status).**
4. Return the result of calling the *[[Call]]* internal method of *reviver* passing *holder* as the **this** value and with an argument list consisting of *name* and *val*.

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

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

### 15.12.3 **stringify ( value [ , replacer [ , space ] ] )**

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

These are the steps in stringifying an object:

1. Let *stack* be an empty List.
2. Let *indent* be the empty String.
3. Let *PropertyList* and *ReplacerFunction* be **undefined**.
4. If *Type(replacer)* is Object, then
  - a. If *IsCallable(replacer)* is **true**, then
    - i. Let *ReplacerFunction* be *replacer*.
  - b. Else if *replacer* has a *[[NativeBrand]] internal property* with value *NativeArray*, then
    - i. Let *PropertyList* be an empty internal List
    - ii. For each value *v* of a property of *replacer* that has an array index property name. The properties are enumerated in the ascending array index order of their names.
      1. Let *item* be **undefined**.
      2. If *Type(v)* is String then let *item* be *v*.
      3. Else if *Type(v)* is Number then let *item* be *ToString(v)*.
      4. Else if *Type(v)* is Object then,
        - a. If *v* has either an *[[NativeBrand]] internal property* whose value is either *StringWrapper* or *NumberWrapper* then let *item* be *ToString(v)*.
      5. If *item* is not **undefined** and *item* is not currently an element of *PropertyList* then,
        - a. Append *item* to the end of *PropertyList*.
  5. If *Type(space)* is Object then,
    - a. If *space* has a *[[NativeBrand]] internal property* whose value is *NumberWrapper*, then,
      - i. Let *space* be *ToNumber(space)*.
    - b. Else if *space* has a *[[NativeBrand]] internal property* with value *StringWrapper*, then,
      - i. Let *space* be *ToString(space)*.
  6. If *Type(space)* is Number

**Deleted:** should

**Deleted:** be

**Deleted:** Call

**Deleted:** Call

**Deleted:** The first parameter is required.

**Deleted:** the *[[Class]] internal property of*

**Deleted:** *isArray*

**Deleted:** *"Array"*

**Deleted:** the *[[Class]] internal property of*

**Deleted:** *"String" or "Number"*

**Deleted:** *IsString*

**Deleted:** *or an *[[IsNumber]]**

**Deleted:** the *[[Class]] internal property of*

**Deleted:** *IsNumber*

**Deleted:** *"Number"*

**Deleted:** the *[[Class]] internal property of*

**Deleted:** *IsString*

**Deleted:** *"String"*

- a. Let *space* be  $\min(10, \text{ToInteger}(\text{space}))$ .
- b. Set *gap* to a String containing *space* space characters. This will be the empty String if *space* is less than 1.
7. Else if *Type(space)* is String
  - a. If the number of characters in *space* is 10 or less, set *gap* to *space* otherwise set *gap* to a String consisting of the first 10 characters of *space*.
8. Else
  - a. Set *gap* to the empty String.
9. Let *wrapper* be be the result of the abstraction operation *ObjectCreate* (15.2).
10. Call the *[[DefineOwnProperty]]* internal method of *wrapper* with arguments the empty String, the Property Descriptor *{[[Value]]: value, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}*, and *false*.
11. Return the result of calling the abstract operation *Str* with the empty String and *wrapper*.

**Deleted:** a new object created as if by the expression `new Object()`, where `Object` is standard built-in constructor with that name

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

1. Let *value* be the result of calling the *[[Get]]* internal method of *holder* with argument *key*.
2. *ReturnIfAbrupt(value)*.
3. If *Type(value)* is Object, then
  - a. Let *toJSON* be the result of calling the *[[Get]]* internal method of *value* with argument `"toJSON"`.
  - b. If *IsCallable(toJSON)* is *true*
    - i. Let *value* be the result of calling the *[[Call]]* internal method of *toJSON* passing *value* as the *this* value and with an argument list consisting of *key*.
    - ii. *ReturnIfAbrupt(value)*.
4. If *ReplacerFunction* is not *undefined*, then
  - a. Let *value* be the result of calling the *[[Call]]* internal method of *ReplacerFunction* passing *holder* as the *this* value and with an argument list consisting of *key* and *value*.
  - b. *ReturnIfAbrupt(value)*.
5. If *Type(value)* is Object then,
  - a. If *value* has an *[[NativeBrand]] internal property with value NumberWrapper*, then,
    - i. Let *value* be *ToNumber(value)*.
  - b. Else if *value* has an *[[NativeBrand]] internal property with value StringWrapper*, then,
    - i. Let *value* be *ToString(value)*.
  - c. Else if *value* has an *[[NativeBrand]] internal property with value BooleanWrapper*, then,
    - i. Let *value* be the value of the *[[PrimitiveValue]]* internal property of *value*.
6. If *value* is *null* then return `"null"`.
7. If *value* is *true* then return `"true"`.
8. If *value* is *false* then return `"false"`.
9. If *Type(value)* is String, then return the result of calling the abstract operation *Quote* with argument *value*.
10. If *Type(value)* is Number
  - a. If *value* is finite then return *ToString(value)*.
  - b. Else, return `"null"`.
11. If *Type(value)* is Object, and *IsCallable(value)* is *false*
  - a. If *value* has an *[[NativeBrand]] internal property with value NativeArray*, then
    - i. Return the result of calling the abstract operation *J4* with argument *value*.
  - b. Else, return the result of calling the abstract operation *JO* with argument *value*.
12. Return *undefined*.

**Deleted:** the *[[Class]]* internal property of  
**Deleted:** *IsNumber*  
**Deleted:** *is "Number"*  
**Deleted:** the *[[Class]]* internal property of  
**Deleted:** *IsString*  
**Deleted:** *is "String"*  
**Deleted:** the *[[Class]]* internal property of  
**Deleted:** *IsBoolean*  
**Deleted:** *is "Boolean"*

**Deleted:** the *[[Class]]* internal property of  
**Deleted:** *IsArray*  
**Deleted:** *is "Array"*

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

1. Let *product* be the double quote character.
2. For each character *C* in *value*
  - a. If *C* is the double quote character or the backslash character
    - i. Let *product* be the concatenation of *product* and the backslash character.
    - ii. Let *product* be the concatenation of *product* and *C*.
  - b. Else if *C* is backspace, formfeed, newline, carriage return, or tab
    - i. Let *product* be the concatenation of *product* and the backslash character.

ii. Let *abbrev* be the character corresponding to the value of *C* as follows:

backspace	" <b>b</b> "
formfeed	" <b>f</b> "
newline	" <b>n</b> "
carriage return	" <b>r</b> "
tab	" <b>t</b> "

iii. Let *product* be the concatenation of *product* and *abbrev*.

- c. Else if *C* is a control character having a code unit value less than the space character
  - i. Let *product* be the concatenation of *product* and the backslash character.
  - ii. Let *product* be the concatenation of *product* and "u".
  - iii. Let *hex* be the result of converting the numeric code unit value of *C* to a String of four hexadecimal digits. *Alphabetic hexadecimal digits are presented as lowercase characters.*
  - iv. Let *product* be the concatenation of *product* and *hex*.
- d. Else
  - i. Let *product* be the concatenation of *product* and *C*.

3. Let *product* be the concatenation of *product* and the double quote character.

4. Return *product*.

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

**Deleted:** ,

**Deleted:** *ReplacerFunction*, and *space*

1. If *stack* contains *value* then throw a **TypeError** exception because the structure is cyclical.
2. Append *value* to *stack*.
3. Let *stepback* be *indent*.
4. Let *indent* be the concatenation of *indent* and *gap*.
5. If *PropertyList* is not **undefined**, then
  - a. Let *K* be *PropertyList*.
6. Else
  - a. Let *K* be an internal List of Strings consisting of the names of all the own properties of *value* whose **[Enumerable]** attribute is **true**. The ordering of the Strings **is the same as that used by the Object.keys standard built-in function**.
7. Let *partial* be an empty List.
8. For each element *P* of *K*.
  - a. Let *strP* be the result of calling the abstract operation *Str* with arguments *P* and *value*.
  - b. *ReturnIfAbrupt(strP)*.
  - c. If *strP* is not **undefined**
    - i. Let *member* be the result of calling the abstract operation *Quote* with argument *P*.
    - ii. Let *member* be the concatenation of *member* and the colon character.
    - iii. If *gap* is not the empty String
      1. Let *member* be the concatenation of *member* and the space character.
    - iv. Let *member* be the concatenation of *member* and *strP*.
    - v. Append *member* to *partial*.
9. If *partial* is empty, then
  - a. Let *final* be "{}".
10. Else
  - a. If *gap* is the empty String
    - i. Let *properties* be a String formed by concatenating all the element Strings of *partial* with each adjacent pair of Strings separated with the comma character. A comma is not inserted either before the first String or after the last String.
    - ii. Let *final* be the result of concatenating "{}", *properties*, and "{}".
  - b. Else *gap* is not the empty String
    - i. Let *separator* be the result of concatenating the comma character, the line feed character, and *indent*.
    - ii. Let *properties* be a String formed by concatenating all the element Strings of *partial* with each adjacent pair of Strings separated with *separator*. The *separator* String is not inserted either before the first String or after the last String.
    - iii. Let *final* be the result of concatenating "{}", the line feed character, *indent*, *properties*, the line feed character, *stepback*, and "{}".
11. Remove the last element of *stack*.
12. Let *indent* be *stepback*.

**Deleted:** should be

13. Return *final*.

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

**Deleted:** , and space

1. If *stack* contains *value* then throw a **TypeError** exception because the structure is cyclical.
2. Append *value* to *stack*.
3. Let *stepback* be *indent*.
4. Let *indent* be the concatenation of *indent* and *gap*.
5. Let *partial* be an empty List.
6. **Assert:** *value* is a native array and hence its "length" property is a non-negative integer.
7. Let *len* be the result of calling the [[Get]] internal method of *value* with argument "length".
8. Let *index* be 0.
9. Repeat while *index < len*
  - a. Let *strP* be the result of calling the abstract operation *Str* with arguments *ToString(index)* and *value*.
  - b. **ReturnIfAbrupt(strP)**.
  - c. If *strP* is **undefined**
    - i. Append "null" to *partial*.
  - d. Else
    - i. Append *strP* to *partial*.
  - e. Increment *index* by 1.
10. If *partial* is empty ,then
  - a. Let *final* be "[ ]".
11. Else
  - a. If *gap* is the empty String
    - i. Let *properties* be a String formed by concatenating all the element Strings of *partial* with each adjacent pair of Strings separated with the comma character. A comma is not inserted either before the first String or after the last String.
    - ii. Let *final* be the result of concatenating "[ ", *properties*, and " ] ".
  - b. Else
    - i. Let *separator* be the result of concatenating the comma character, the line feed character, and *indent*.
    - ii. Let *properties* be a String formed by concatenating all the element Strings of *partial* with each adjacent pair of Strings separated with *separator*. The *separator* String is not inserted either before the first String or after the last String.
    - iii. Let *final* be the result of concatenating "[ ", the line feed character, *indent*, *properties*, the line feed character, *stepback*, and " ] ".
12. Remove the last element of *stack*.
13. Let *indent* be *stepback*.
14. Return *final*.

**NOTE 1** JSON structures are allowed to be nested to any depth, but they must be acyclic. If *value* is or contains a cyclic structure, then the *stringify* function must throw a **TypeError** exception. This is an example of a value that cannot be stringified:

```
a = [];
a[0] = a;
my_text = JSON.stringify(a); // This must throw an TypeError.
```

**NOTE 2** Symbolic primitive values are rendered as follows:

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

**NOTE 3** String values are wrapped in double quotes. The characters " and \ are escaped with \ prefixes. Control characters are replaced with escape sequences \uHHHH, or with the shorter forms, \b (backspace), \f (formfeed), \n (newline), \r (carriage return), \t (tab).

NOTE 4 Finite numbers are stringified as if by calling `ToString(number)`. `NaN` and `Infinity` regardless of sign are represented as the String `null`.

NOTE 5 Values that do not have a JSON representation (such as `undefined` and functions) do not produce a String. Instead they produce the `undefined` value. In arrays these values are represented as the String `null`. In objects an unrepresentable value causes the property to be excluded from stringification.

NOTE 6 An object is rendered as an opening left brace followed by zero or more properties, separated with commas, closed with a right brace. A property is a quoted String representing the key or property name, a colon, and then the stringified property value. An array is rendered as an opening left bracket followed by zero or more values, separated with commas, closed with a right bracket.

## 16 Errors

An implementation must report most errors at the time the relevant ECMAScript language construct is evaluated. An *early error* is an error that can be detected and reported prior to the evaluation of any construct in the *Program* containing the error. An implementation must report early errors in a *Program* prior to the first evaluation of that *Program*. Early errors in `eval` code are reported at the time `eval` is called but prior to evaluation of any construct within the `eval` code. All errors that are not early errors are runtime errors.

An implementation must treat any instance of the following kinds of errors as an early error:

- Any syntax error.
- Attempts to define an *ObjectLiteral* that has multiple `get` property assignments with the same name or multiple `set` property assignments with the same name.
- Attempts to define an *ObjectLiteral* that has both a data property assignment and a `get` or `set` property assignment with the same name.
- Errors in regular expression literals that are not implementation-defined syntax extensions.
- Attempts in strict mode code to define an *ObjectLiteral* that has multiple data property assignments with the same name.
- The occurrence of a *WithStatement* in strict mode code.
- The occurrence of an *Identifier* value appearing more than once within a *FormalParameterList* of an individual strict mode *FunctionDeclaration* or *FunctionExpression*.
- Improper uses of `return`, `break`, and `continue`.
- Attempts to call `PutValue` on any value for which an early determination can be made that the value is not a Reference (for example, executing the assignment statement `3=4`).

An implementation shall not treat other kinds of errors as early errors even if the compiler can prove that a construct cannot execute without error under any circumstances. An implementation may issue an early warning in such a case, but it should not report the error until the relevant construct is actually executed.

An implementation shall report all errors as specified, except for the following:

- An implementation may extend program syntax and regular expression pattern or flag syntax. To permit this, all operations (such as calling `eval`, using a regular expression literal, or using the `Function` or `RegExp` constructor) that are allowed to throw `SyntaxError` are permitted to exhibit implementation-defined behaviour instead of throwing `SyntaxError` when they encounter an implementation-defined extension to the program syntax or regular expression pattern or flag syntax.
- An implementation may provide additional types, values, objects, properties, and functions beyond those described in this specification. This may cause constructs (such as looking up a variable in the global scope) to have implementation-defined behaviour instead of throwing an error (such as `ReferenceError`).
- An implementation may define behaviour other than throwing `RangeError` for `toFixed`, `toExponential`, and `toPrecision` when the *fractionDigits* or *precision* argument is outside the specified range.

DRAFT

## Annex A (informative)

### Grammar Summary

Field Code Changed  
Field Code Changed  
Field Code Changed

#### A.1 Lexical Grammar

*SourceCharacter* ::  
any Unicode code unit

*InputElementDiv* ::  
*WhiteSpace*  
*LineTerminator*  
*Comment*  
*Token*  
*DivPunctuator*

*InputElementRegExp* ::  
*WhiteSpace*  
*LineTerminator*  
*Comment*  
*Token*  
*RegularExpressionLiteral*

*WhiteSpace* ::  
<TAB>  
<VT>  
<FF>  
<SP>  
<NBSP>  
<BOM>  
<USP>

*LineTerminator* ::  
<LF>  
<CR>  
<LS>  
<PS>

*LineTerminatorSequence* ::  
<LF>  
<CR> [lookahead ≠ <LF>]  
<LS>  
<PS>  
<CR> <LF>

*Comment* ::  
*MultiLineComment*  
*SingleLineComment*

See clause 6

See clause 7

See clause 7

See 7.2

See 7.3

See 7.3

See 7.4

<i>MultiLineComment ::</i>	See 7.4
<i>/* MultiLineCommentChars<sub>opt</sub> */</i>	
<i>MultiLineCommentChars ::</i>	See 7.4
<i>MultiLineNotAsteriskChar MultiLineCommentChars<sub>opt</sub></i>	
<i>* PostAsteriskCommentChars<sub>opt</sub></i>	
<i>PostAsteriskCommentChars ::</i>	See 7.4
<i>MultiLineNotForwardSlashOrAsteriskChar MultiLineCommentChars<sub>opt</sub></i>	
<i>* PostAsteriskCommentChars<sub>opt</sub></i>	
<i>MultiLineNotAsteriskChar ::</i>	See 7.4
<i>SourceCharacter but not *</i>	
<i>MultiLineNotForwardSlashOrAsteriskChar ::</i>	See 7.4
<i>SourceCharacter but not one of / or *</i>	
<i>SingleLineComment ::</i>	See 7.4
<i>// SingleLineCommentChars<sub>opt</sub></i>	
<i>SingleLineCommentChars ::</i>	See 7.4
<i>SingleLineCommentChar SingleLineCommentChars<sub>opt</sub></i>	
<i>SingleLineCommentChar ::</i>	See 7.4
<i>SourceCharacter but not LineTerminator</i>	
<i>Token ::</i>	See 7.5
<i>IdentifierName</i>	
<i>Punctuator</i>	
<i>NumericLiteral</i>	
<i>StringLiteral</i>	
<i>Identifier ::</i>	See 7.6
<i>IdentifierName but not ReservedWord</i>	
<i>IdentifierName ::</i>	See 7.6
<i>IdentifierStart</i>	
<i>IdentifierName IdentifierPart</i>	
<i>IdentifierStart ::</i>	See 7.6
<i>UnicodeLetter</i>	
<i>\$</i>	
<i>\ UnicodeEscapeSequence</i>	

*IdentifierPart* ::

*IdentifierStart*  
*UnicodeCombiningMark*  
*UnicodeDigit*  
*UnicodeConnectorPunctuation*  
*<ZWNJ>*  
*<ZWJ>*

See 7.6

*UnicodeLetter* ::

any character in the Unicode categories “Uppercase letter (Lu)”, “Lowercase letter (Ll)”, “Titlecase letter (Lt)”, “Modifier letter (Lm)”, “Other letter (Lo)”, or “Letter number (Ni)”.

See 7.6

*UnicodeCombiningMark* ::

any character in the Unicode categories “Non-spacing mark (Mn)” or “Combining spacing mark (Mc)”

See 7.6

*UnicodeDigit* ::

any character in the Unicode category “Decimal number (Nd)”

See 7.6

*UnicodeConnectorPunctuation* ::

any character in the Unicode category “Connector punctuation (Pc)”

See 7.6

*ReservedWord* ::

*Keyword*  
*FutureReservedWord*  
*NullLiteral*  
*BooleanLiteral*

See 7.6.1

*Keyword* :: one of

break	do	instanceof	typeof
case	else	new	var
catch	finally	return	void
continue	for	switch	while
debugger	function	this	with
default	if	throw	
delete	in	try	

See 7.6.1.1

*FutureReservedWord* :: one of

class	enum	extends	super
const	export	import	

See 7.6.1.2

The following tokens are also considered to be *FutureReservedWords* when parsing strict mode code (see 10.1.1).

implements	let	private	public
interface	package	protected	static
yield			

<i>Punctuator :: one of</i>						<i>See 7.7</i>
{	}	(	)	[	]	
,	;	,	<	>	<=	
>=	==	!=	====	!==		
+	-	*	%	++	--	
<<	>>	>>>	&		^	
!	~	&&		?	:	
=	+=	-=	*=	%=	<<=	
>>=	>>>=	&=	=	^=		
<i>DivPunctuator :: one of</i>						<i>See 7.7</i>
/	/=					<i>See 7.8</i>
<i>Literal ::</i>						
<i>NullLiteral</i>						
<i>BooleanLiteral</i>						
<i>NumericLiteral</i>						
<i>StringLiteral</i>						
<i>RegularExpressionLiteral</i>						
<i>NullLiteral ::</i>	<b>null</b>					<i>See 7.8.1</i>
<i>BooleanLiteral ::</i>	<b>true</b>					<i>See 7.8.2</i>
<i>false</i>						
<i>NumericLiteral ::</i>						<i>See 7.8.3</i>
<i>DecimalLiteral</i>						
<i>HexIntegerLiteral</i>						
<i>DecimalLiteral ::</i>						<i>See 7.8.3</i>
<i>DecimalIntegerLiteral</i>	.	<i>DecimalDigits<sub>opt</sub></i>	<i>ExponentPart<sub>opt</sub></i>			
.	<i>DecimalDigits</i>	<i>ExponentPart<sub>opt</sub></i>				
<i>DecimalIntegerLiteral</i>		<i>ExponentPart<sub>opt</sub></i>				
<i>DecimalIntegerLiteral ::</i>						<i>See 7.8.3</i>
<b>0</b>						
<i>NonZeroDigit DecimalDigits<sub>opt</sub></i>						
<i>DecimalDigits ::</i>						<i>See 7.8.3</i>
<i>DecimalDigit</i>						
<i>DecimalDigits DecimalDigit</i>						
<i>DecimalDigit :: one of</i>						<i>See 7.8.3</i>
0	1	2	3	4	5	6
7	8	9				

<i>NonZeroDigit</i> :: one of 1 2 3 4 5 6 7 8 9	See 7.8.3
<i>ExponentPart</i> :: <i>ExponentIndicator</i> <i>SignedInteger</i>	See 7.8.3
<i>ExponentIndicator</i> :: one of e E	See 7.8.3
<i>SignedInteger</i> :: DecimalDigits + DecimalDigits - DecimalDigits	See 7.8.3
<i>HexIntegerLiteral</i> :: 0x HexDigit 0X HexDigit <i>HexIntegerLiteral</i> HexDigit	See 7.8.3
<i>HexDigit</i> :: one of 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F	See 7.8.3
<i>StringLiteral</i> :: " <i>DoubleStringCharacters</i> <sub>opt</sub> " ' <i>SingleStringCharacters</i> <sub>opt</sub> '	See 7.8.4
<i>DoubleStringCharacters</i> :: <i>DoubleStringCharacter</i> <i>DoubleStringCharacters</i> <sub>opt</sub>	See 7.8.4
<i>SingleStringCharacters</i> :: <i>SingleStringCharacter</i> <i>SingleStringCharacters</i> <sub>opt</sub>	See 7.8.4
<i>DoubleStringCharacter</i> :: SourceCharacter <b>but not one of " or \ or LineTerminator</b> \ <i>EscapeSequence</i> <i>LineContinuation</i>	See 7.8.4
<i>SingleStringCharacter</i> :: SourceCharacter <b>but not one of ' or \ or LineTerminator</b> \ <i>EscapeSequence</i> <i>LineContinuation</i>	See 7.8.4
<i>LineContinuation</i> :: \ <i>LineTerminatorSequence</i>	See 7.8.4
<i>EscapeSequence</i> :: <i>CharacterEscapeSequence</i> 0 [lookahead $\notin$ DecimalDigit] <i>HexEscapeSequence</i> <i>UnicodeEscapeSequence</i>	See 7.8.4
<i>CharacterEscapeSequence</i> :: <i>SingleEscapeCharacter</i> <i>NonEscapeCharacter</i>	See 7.8.4
<i>SingleEscapeCharacter</i> :: one of ' " \ b f n r t v	See 7.8.4

<i>NonEscapeCharacter ::</i>	<i>SourceCharacter but not one of EscapeCharacter or LineTerminator</i>	See 7.8.4
<i>EscapeCharacter ::</i>		See 7.8.4
<i>SingleEscapeCharacter</i>		
<i>DecimalDigit</i>		
<i>x</i>		
<i>u</i>		
<i>HexEscapeSequence ::</i>		See 7.8.4
<i>x HexDigit HexDigit</i>		
<i>UnicodeEscapeSequence ::</i>		See 7.8.4
<i>u HexDigit HexDigit HexDigit</i>		
<i>RegularExpressionLiteral ::</i>		See 7.8.5
<i>/ RegularExpressionBody / RegularExpressionFlags</i>		
<i>RegularExpressionBody ::</i>		See 7.8.5
<i>RegularExpressionFirstChar RegularExpressionChars</i>		
<i>RegularExpressionChars ::</i>		See 7.8.5
<i>[empty]</i>		
<i>RegularExpressionChars RegularExpressionChar</i>		
<i>RegularExpressionFirstChar ::</i>		See 7.8.5
<i>RegularExpressionNonTerminator but not one of * or \ or / or [</i>		
<i>RegularExpressionBackslashSequence</i>		
<i>RegularExpressionClass</i>		
<i>RegularExpressionChar ::</i>		See 7.8.5
<i>RegularExpressionNonTerminator but not \ or / or [</i>		
<i>RegularExpressionBackslashSequence</i>		
<i>RegularExpressionClass</i>		
<i>RegularExpressionBackslashSequence ::</i>		See 7.8.5
<i>\ RegularExpressionNonTerminator</i>		
<i>RegularExpressionNonTerminator ::</i>		See 7.8.5
<i>SourceCharacter but not LineTerminator</i>		
<i>RegularExpressionClass ::</i>		See 7.8.5
<i>[ RegularExpressionClassChars ]</i>		
<i>RegularExpressionClassChars ::</i>		See 7.8.5
<i>[empty]</i>		
<i>RegularExpressionClassChars RegularExpressionClassChar</i>		
<i>RegularExpressionClassChar ::</i>		See 7.8.5
<i>RegularExpressionNonTerminator but not ] or \</i>		
<i>RegularExpressionBackslashSequence</i>		

*RegularExpressionFlags* :::  
 [empty]  
*RegularExpressionFlags IdentifierPart*

See 7.8.5

## A.2 Number Conversions

*StringNumericLiteral* :::  
*StrWhiteSpace<sub>opt</sub>*  
*StrWhiteSpace<sub>opt</sub> StrNumericLiteral StrWhiteSpace<sub>opt</sub>*

See 9.3.1

*StrWhiteSpace* :::  
*StrWhiteSpaceChar StrWhiteSpace<sub>opt</sub>*

See 9.3.1

*StrWhiteSpaceChar* :::  
*WhiteSpace*  
*LineTerminator*

See 9.3.1

*StrNumericLiteral* :::  
*StrDecimalLiteral*  
*HexIntegerLiteral*

See 9.3.1

*StrDecimalLiteral* :::  
*StrUnsignedDecimalLiteral*  
*+ StrUnsignedDecimalLiteral*  
*- StrUnsignedDecimalLiteral*

See 9.3.1

*StrUnsignedDecimalLiteral* :::  
**Infinity**  
*DecimalDigits . DecimalDigits<sub>opt</sub> ExponentPart<sub>opt</sub>*  
*. DecimalDigits ExponentPart<sub>opt</sub>*  
*DecimalDigits ExponentPart<sub>opt</sub>*

See 9.3.1

*DecimalDigits* :::  
*DecimalDigit*  
*DecimalDigits DecimalDigit*

See 9.3.1

*DecimalDigit* :: **one of**  
 0 1 2 3 4 5 6 7 8 9

See 9.3.1

*ExponentPart* :::  
*ExponentIndicator SignedInteger*

See 9.3.1

*ExponentIndicator* :: **one of**  
 e E

See 9.3.1

*SignedInteger* :::  
*DecimalDigits*  
*+ DecimalDigits*  
*- DecimalDigits*

See 9.3.1

*HexIntegerLiteral* ::::  
 0x *HexDigit*  
 0X *HexDigit*  
*HexIntegerLiteral* *HexDigit*

See 9.3.1

*HexDigit* :::: one of  
 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

See 9.3.1

### A.3 Expressions

*PrimaryExpression* :  
 this  
*Identifier*  
*Literal*  
*ArrayLiteral*  
*ObjectLiteral*  
 ( Expression )

See 11.1

*ArrayLiteral* :  
 [ *Elision*<sub>opt</sub> ]  
 [ *ElementList* ]  
 [ *ElementList* , *Elision*<sub>opt</sub> ]

See 11.1.4

*ElementList* :  
*Elision*<sub>opt</sub> AssignmentExpression  
*ElementList* , *Elision*<sub>opt</sub> AssignmentExpression

See 11.1.4

*Elision* :

'  
*Elision* ,

See 11.1.4

*ObjectLiteral* :  
 { }  
 { *PropertyNameAndValueList* }  
 { *PropertyNameAndValueList* , }

See 11.1.5

*PropertyNameAndValueList* :  
*PropertyNameAssignment*  
*PropertyNameAndValueList* , *PropertyNameAssignment*

See 11.1.5

*PropertyNameAssignment* :  
*PropertyName* : AssignmentExpression  
**get** *PropertyName* ( ) { *FunctionBody* }  
**set** *PropertyName* ( *PropertySetParameterList* ) { *FunctionBody* }

See 11.1.5

*PropertyName* :  
*IdentifierName*  
*StringLiteral*  
*NumericLiteral*

See 11.1.5

*PropertySetParameterList* :  
*Identifier*

See 11.1.5

**Deleted:** *PropertyNameAndValueList*  
**Deleted:** *PropertyNameAndValueList*  
**Deleted:** *PropertyNameAssignment*  
**Deleted:** *PropertyNameAssignment*  
**Deleted:** *PropertyNameAssignment*  
**Deleted:** *PropertyNameAssignment*

*MemberExpression :*

- PrimaryExpression*
- FunctionExpression*
- MemberExpression [ Expression ]*
- MemberExpression . IdentifierName*
- new MemberExpression Arguments**

See 11.2

*NewExpression :*

- MemberExpression*
- new NewExpression**

See 11.2

*CallExpression :*

- MemberExpression Arguments*
- CallExpression Arguments*
- CallExpression [ Expression ]*
- CallExpression . IdentifierName*

See 11.2

*Arguments :*

- ()*
- ( ArgumentList )*

See 11.2

*ArgumentList :*

- AssignmentExpression*
- ArgumentList , AssignmentExpression*

See 11.2

*LeftHandSideExpression :*

- NewExpression*
- CallExpression*

See 11.2

*PostfixExpression :*

- LeftHandSideExpression*
- LeftHandSideExpression [no LineTerminator here] ++*
- LeftHandSideExpression [no LineTerminator here] --*

See 11.3

*UnaryExpression :*

- PostfixExpression*
- delete UnaryExpression**
- void UnaryExpression**
- typeof UnaryExpression**
- ++ UnaryExpression**
- UnaryExpression**
- + UnaryExpression**
- UnaryExpression**
- ~ UnaryExpression**
- ! UnaryExpression**

See 11.4

*MultiplicativeExpression :*

- UnaryExpression*
- MultiplicativeExpression \* UnaryExpression*
- MultiplicativeExpression / UnaryExpression*
- MultiplicativeExpression % UnaryExpression*

See 11.5

*AdditiveExpression :*

*MultiplicativeExpression*  
*AdditiveExpression + MultiplicativeExpression*  
*AdditiveExpression - MultiplicativeExpression*

See 11.6

*ShiftExpression :*

*AdditiveExpression*  
*ShiftExpression << AdditiveExpression*  
*ShiftExpression >> AdditiveExpression*  
*ShiftExpression >>> AdditiveExpression*

See 11.7

*RelationalExpression :*

*ShiftExpression*  
*RelationalExpression < ShiftExpression*  
*RelationalExpression > ShiftExpression*  
*RelationalExpression <= ShiftExpression*  
*RelationalExpression >= ShiftExpression*  
*RelationalExpression instanceof ShiftExpression*  
*RelationalExpression in ShiftExpression*

See 11.8

*RelationalExpressionNoIn :*

*ShiftExpression*  
*RelationalExpressionNoIn < ShiftExpression*  
*RelationalExpressionNoIn > ShiftExpression*  
*RelationalExpressionNoIn <= ShiftExpression*  
*RelationalExpressionNoIn >= ShiftExpression*  
*RelationalExpressionNoIn instanceof ShiftExpression*

See 11.8

*EqualityExpression :*

*RelationalExpression*  
*EqualityExpression == RelationalExpression*  
*EqualityExpression != RelationalExpression*  
*EqualityExpression === RelationalExpression*  
*EqualityExpression !== RelationalExpression*

See 11.9

*EqualityExpressionNoIn :*

*RelationalExpressionNoIn*  
*EqualityExpressionNoIn == RelationalExpressionNoIn*  
*EqualityExpressionNoIn != RelationalExpressionNoIn*  
*EqualityExpressionNoIn === RelationalExpressionNoIn*  
*EqualityExpressionNoIn !== RelationalExpressionNoIn*

See 11.9

*BitwiseANDExpression :*

*EqualityExpression*  
*BitwiseANDExpression & EqualityExpression*

See 11.10

*BitwiseANDExpressionNoIn :*

*EqualityExpressionNoIn*  
*BitwiseANDExpressionNoIn & EqualityExpressionNoIn*

See 11.10

*BitwiseXORExpression :*  
*BitwiseANDExpression*  
*BitwiseXORExpression ^ BitwiseANDExpression*

*See 11.10*

*BitwiseXORExpressionNoIn :*  
*BitwiseANDExpressionNoIn*  
*BitwiseXORExpressionNoIn ^ BitwiseANDExpressionNoIn*

*See 11.10*

*BitwiseORExpression :*  
*BitwiseXORExpression*  
*BitwiseORExpression | BitwiseXORExpression*

*See 11.10*

*BitwiseORExpressionNoIn :*  
*BitwiseXORExpressionNoIn*  
*BitwiseORExpressionNoIn | BitwiseXORExpressionNoIn*

*See 11.10*

*LogicalANDExpression :*  
*BitwiseORExpression*  
*LogicalANDExpression && BitwiseORExpression*

*See 11.11*

*LogicalANDExpressionNoIn :*  
*BitwiseORExpressionNoIn*  
*LogicalANDExpressionNoIn && BitwiseORExpressionNoIn*

*See 11.11*

*LogicalORExpression :*  
*LogicalANDExpression*  
*LogicalORExpression || LogicalANDExpression*

*See 11.11*

*LogicalORExpressionNoIn :*  
*LogicalANDExpressionNoIn*  
*LogicalORExpressionNoIn || LogicalANDExpressionNoIn*

*See 11.11*

*ConditionalExpression :*  
*LogicalORExpression*  
*LogicalORExpression ? AssignmentExpression : AssignmentExpression*

*See 11.12*

*ConditionalExpressionNoIn :*  
*LogicalORExpressionNoIn*  
*LogicalORExpressionNoIn ? AssignmentExpression : AssignmentExpressionNoIn*

*See 11.12*

*AssignmentExpression :*  
*ConditionalExpression*  
*LeftHandSideExpression = AssignmentExpression*  
*LeftHandSideExpression AssignmentOperator AssignmentExpression*

*See 11.13*

*AssignmentExpressionNoIn :*  
*ConditionalExpressionNoIn*  
*LeftHandSideExpression = AssignmentExpressionNoIn*  
*LeftHandSideExpression AssignmentOperator AssignmentExpressionNoIn*

*See 11.13*

*AssignmentOperator* : **one of**  
 $\ast\ast=$     $/=$     $\%=$     $+=$     $-=$     $<<=$     $>>=$     $>>>=$     $\&=$     $\wedge=$     $\mid=$

See 11.13

*Expression* :  
*AssignmentExpression*  
*Expression* , *AssignmentExpression*

See 11.14

*ExpressionNoIn* :  
*AssignmentExpressionNoIn*  
*ExpressionNoIn* , *AssignmentExpressionNoIn*

See 11.14

## A.4 Statements

*Statement* :  
*Block*  
*VariableStatement*  
*EmptyStatement*  
*ExpressionStatement*  
*IfStatement*  
*IterationStatement*  
*ContinueStatement*  
*BreakStatement*  
*ReturnStatement*  
*WithStatement*  
*LabelledStatement*  
*SwitchStatement*  
*ThrowStatement*  
*TryStatement*  
*DebuggerStatement*

See clause 12

*Block* :  
 $\{ \text{StatementList}_{\text{opt}} \}$

See 12.1

*StatementList* :  
*Statement*  
*StatementList Statement*

See 12.1

*VariableStatement* :  
 $\text{var } \text{VariableDeclarationList} ;$

See 12.2

*VariableDeclarationList* :  
*VariableDeclaration*  
*VariableDeclarationList , VariableDeclaration*

See 12.2

*VariableDeclarationListNoIn* :  
*VariableDeclarationNoIn*  
*VariableDeclarationListNoIn , VariableDeclarationNoIn*

See 12.2

*VariableDeclaration* :  
*Identifier Initialiser*<sub>opt</sub>

See 12.2

*VariableDeclarationNoIn* :  
*Identifier InitialiserNoIn*<sub>opt</sub>

See 12.2

*Initialiser :*  
  = *AssignmentExpression*

See 12.2

*InitialiserNoIn :*  
  = *AssignmentExpressionNoIn*

See 12.2

*EmptyStatement :*  
  ;

See 12.3

*ExpressionStatement :*  
  [lookahead  $\notin \{\{, \text{function}\}$ ] *Expression* ;

See 12.4

*IfStatement :*  
  **if** ( *Expression* ) *Statement* **else** *Statement*  
  **if** ( *Expression* ) *Statement*

See 12.5

*IterationStatement :*  
  **do** *Statement* **while** ( *Expression* ) ;  
  **while** ( *Expression* ) *Statement*  
  **for** ( *ExpressionNoIn*<sub>opt</sub>; *Expression*<sub>opt</sub>; *Expression*<sub>opt</sub> ) *Statement*  
  **for** ( **var** *VariableDeclarationListNoIn*; *Expression*<sub>opt</sub>; *Expression*<sub>opt</sub> ) *Statement*  
  **for** ( *LeftHandSideExpression* **in** *Expression* ) *Statement*  
  **for** ( **var** *VariableDeclarationNoIn* **in** *Expression* ) *Statement*

See 12.6

*ContinueStatement :*  
  **continue** ;  
  **continue** [no LineTerminator here] *Identifier* ;

See 12.7

*BreakStatement :*  
  **break** ;  
  **break** [no LineTerminator here] *Identifier* ;

See 12.8

*ReturnStatement :*  
  **return** ;  
  **return** [no LineTerminator here] *Expression* ;

See 12.9

*WithStatement :*  
  **with** ( *Expression* ) *Statement*

See 12.10

*SwitchStatement :*  
  **switch** ( *Expression* ) *CaseBlock*

See 12.11

*CaseBlock :*  
  { *CaseClauses*<sub>opt</sub> }  
  { *CaseClauses*<sub>opt</sub> *DefaultClause* *CaseClauses*<sub>opt</sub> }

See 12.11

*CaseClauses :*  
  *CaseClause*  
  *CaseClauses CaseClause*

See 12.11

*CaseClause :*  
**case** Expression : StatementList<sub>opt</sub>

See 12.11

*DefaultClause :*  
**default** : StatementList<sub>opt</sub>

See 12.11

*LabelledStatement :*  
Identifier : Statement

See 12.12

*ThrowStatement :*  
**throw** [no LineTerminator here] Expression ;

See 12.13

*TryStatement :*  
**try** Block Catch  
**try** Block Finally  
**try** Block Catch Finally

See 12.14

*Catch :*  
**catch** ( Identifier ) Block

See 12.14

*Finally :*  
**finally** Block

See 12.14

*DebuggerStatement :*  
**debugger** ;

See 12.15

## A.5 Functions and Programs

*FunctionDeclaration :*  
**function** Identifier ( FormalParameterList<sub>opt</sub> ) { FunctionBody }

See clause 13

*FunctionExpression :*  
**function** Identifier<sub>opt</sub> ( FormalParameterList<sub>opt</sub> ) { FunctionBody }

See clause 13

*FormalParameterList :*  
Identifier  
FormalParameterList , Identifier

See clause 13

*FunctionBody :*  
SourceElements<sub>opt</sub>

See clause 13

*Program :*  
SourceElements<sub>opt</sub>

See clause 14

*SourceElements :*  
SourceElement  
SourceElements SourceElement

See clause 14

*SourceElement :*  
*Statement*  
*FunctionDeclaration*

*See clause 14*

## A.6 Universal Resource Identifier Character Classes

*uri* ::=
   
 $\text{uriCharacters}_{\text{opt}}$  See 15.1.3

*uriCharacters* ::=
   
 $\text{uriCharacter } \text{uriCharacters}_{\text{opt}}$  See 15.1.3

*uriCharacter* ::=
   
 $\text{uriReserved}$   
 $\text{uriUnescaped}$   
 $\text{uriEscaped}$  See 15.1.3

*uriReserved* :: **one of**  
 $\text{; } / \text{? } \text{: } @ \text{& } = \text{+ } \$ \text{, , }$  See 15.1.3

*uriUnescaped* ::=
   
 $\text{uriAlpha}$   
 $\text{DecimalDigit}$   
 $\text{uriMark}$  See 15.1.3

*uriEscaped* ::=
   
 $\% \text{HexDigit} \text{ HexDigit}$  See 15.1.3

*uriAlpha* :: **one of**  
 $\text{a } \text{b } \text{c } \text{d } \text{e } \text{f } \text{g } \text{h } \text{i } \text{j } \text{k } \text{l } \text{m } \text{n } \text{o } \text{p } \text{q } \text{r } \text{s } \text{t } \text{u } \text{v } \text{w } \text{x } \text{y } \text{z }$   
 $\text{A } \text{B } \text{C } \text{D } \text{E } \text{F } \text{G } \text{H } \text{I } \text{J } \text{K } \text{L } \text{M } \text{N } \text{O } \text{P } \text{Q } \text{R } \text{S } \text{T } \text{U } \text{V } \text{W } \text{X } \text{Y } \text{Z }$  See 15.1.3

*uriMark* :: **one of**  
 $\text{- } \text{_ } \text{. } \text{! } \text{~ } \text{* } \text{'} \text{( )}$  See 15.1.3

## A.7 Regular Expressions

*Pattern* ::=
   
 $\text{Disjunction}$  See 15.10.1

*Disjunction* ::=
   
 $\text{Alternative}$   
 $\text{Alternative } | \text{ Disjunction}$  See 15.10.1

*Alternative* ::=
   
 $[\text{empty}]$   
 $\text{Alternative Term}$  See 15.10.1

*Term* ::=
   
 $\text{Assertion}$   
 $\text{Atom}$   
 $\text{Atom Quantifier}$  See 15.10.1

*Assertion ::*

```

^
$ 
\ b
\ B
( ? = Disjunction )
( ? ! Disjunction )

```

See 15.10.1

*Quantifier ::*

```

QuantifierPrefix
QuantifierPrefix ?

```

See 15.10.1

*QuantifierPrefix ::*

```

*
+
?
{ DecimalDigits }
{ DecimalDigits , }
{ DecimalDigits , DecimalDigits }

```

See 15.10.1

*Atom ::*

```

PatternCharacter
.
\ AtomEscape
CharacterClass
( Disjunction )
( ?: Disjunction )

```

See 15.10.1

*PatternCharacter ::*

SourceCharacter **but not one of-**

See 15.10.1

*AtomEscape ::*

```

DecimalEscape
CharacterEscape
CharacterClassEscape

```

See 15.10.1

*CharacterEscape ::*

```

ControlEscape
ControlLetter
HexEscapeSequence
UnicodeEscapeSequence
IdentityEscape

```

See 15.10.1

*ControlEscape :: one of*

f n r t v

See 15.10.1

*ControlLetter :: one of*

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

See 15.10.1

*IdentityEscape ::*  
*SourceCharacter but not IdentifierPart*  
*<ZWJ>*  
*<ZWNJ>*

*See 15.10.1*

*DecimalEscape ::*  
*DecimalIntegerLiteral [lookahead  $\notin$  DecimalDigit]*

*See 15.10.1*

*CharacterClassEscape :: one of*  
**d D s S w W**

*See 15.10.1*

*CharacterClass ::*  
 $[ \text{ lookahead } \notin \{ \} \text{ ClassRanges } ]$   
 $[ ^ \text{ ClassRanges } ]$

*See 15.10.1*

*ClassRanges ::*  
 $\text{[empty]}$   
*NonemptyClassRanges*

*See 15.10.1*

*NonemptyClassRanges ::*  
*ClassAtom*  
*ClassAtom NonemptyClassRangesNoDash*  
*ClassAtom - ClassAtom ClassRanges*

*See 15.10.1*

*NonemptyClassRangesNoDash ::*  
*ClassAtom*  
*ClassAtomNoDash NonemptyClassRangesNoDash*  
*ClassAtomNoDash - ClassAtom ClassRanges*

*See 15.10.1*

*ClassAtom ::*  
 $-$   
*ClassAtomNoDash*

*See 15.10.1*

*ClassAtomNoDash ::*  
*SourceCharacter but not one of \ or ] or -*  
 $\backslash \text{ ClassEscape}$

*See 15.10.1*

*ClassEscape ::*  
*DecimalEscape*  
**b**  
*CharacterEscape*  
*CharacterClassEscape*

*See 15.10.1*

## A.8 JSON

### A.8.1 JSON Lexical Grammar

*JSONWhiteSpace ::*  
*<TAB>*  
*<CR>*

*See 15.12.1.1*

<LF>	
<SP>	
<i>JSONString</i> ::	<i>See 15.12.1.1</i>
" <i>JSONStringCharacters<sub>opt</sub></i> "	
<i>JSONStringCharacters</i> ::	<i>See 15.12.1.1</i>
<i>JSONStringCharacter</i> <i>JSONStringCharacters<sub>opt</sub></i>	
<i>JSONStringCharacter</i> ::	<i>See 15.12.1.1</i>
<i>SourceCharacter</i> <b>but not one of " or \ or U+0000 through U+001F</b>	
\ <i>JSONEscapeSequence</i>	
<i>JSONEscapeSequence</i> ::	<i>See 15.12.1.1</i>
<i>JSONEscapeCharacter</i>	
<i>UnicodeEscapeSequence</i>	
<i>JSONEscapeCharacter</i> :: <b>one of</b>	<i>See 15.12.1.1</i>
" / \ b f n r t	
<i>JSONNumber</i> ::	<i>See 15.12.1.1</i>
- <sub>opt</sub> <i>DecimalIntegerLiteral</i> <i>JSONFraction<sub>opt</sub></i> <i>ExponentPart<sub>opt</sub></i>	
<i>JSONFraction</i> ::	<i>See 15.12.1.1</i>
. <i>DecimalDigits</i>	
<i>JSONNullLiteral</i> ::	<i>See 15.12.1.1</i>
<i>NullLiteral</i>	
<i>JSONBooleanLiteral</i> ::	<i>See 15.12.1.1</i>
<i>BooleanLiteral</i>	
<b>A.8.2 JSON Syntactic Grammar</b>	
<i>JSONText</i> :	<i>See 15.12.1.2</i>
<i>JSONValue</i>	
<i>JSONValue</i> :	<i>See 15.12.1.2</i>
<i>JSONNullLiteral</i>	
<i>JSONBooleanLiteral</i>	
<i>JSONObject</i>	
<i>JSONArray</i>	
<i>JSONString</i>	
<i>JSONNumber</i>	
<i>JSONObject</i> :	<i>See 15.12.1.2</i>
{ }	
{ <i>JSONMemberList</i> }	
<i>JSONMember</i> :	<i>See 15.12.1.2</i>
<i>JSONString</i> : <i>JSONValue</i>	
<i>JSONMemberList</i> :	<i>See 15.12.1.2</i>
<i>JSONMember</i>	
<i>JSONMemberList</i> , <i>JSONMember</i>	
<i>JSONArray</i> :	<i>See 15.12.1.2</i>
[ ]	
[ <i>JSONElementList</i> ]	

*JSONElementList :*  
*JSONValue*  
*JSONElementList , JSONValue*

*See 15.12.1.2*

DRAFT

DRAFT

## Annex B (normative)

### Additional ECMAScript Features for Web Browsers

The ECMAScript language syntax and semantics defined in this annex are required when the ECMAScript host is a web browser. The content of this annex is normative but optional if the ECMAScript host is not a web browser.

#### B.1 Additional Syntax

##### B.1.1 Numeric Literals

The syntax and semantics of 7.8.3 is extended as follows except that this extension is not allowed for strict mode code:

###### Syntax

```
NumericLiteral ::  
  DecimalLiteral  
  BinaryIntegerLiteral  
  OctalIntegerLiteral  
  HexIntegerLiteral  
  LegacyOctalIntegerLiteral
```

```
LegacyOctalIntegerLiteral ::  
  0 OctalDigit  
  LegacyOctalIntegerLiteral OctalDigit
```

###### Semantics

- The MV of *LegacyOctalIntegerLiteral :: 0 OctalDigit* is the MV of *OctalDigit*.
- The MV of *LegacyOctalIntegerLiteral :: LegacyOctalIntegerLiteral OctalDigit* is (the MV of *LegacyOctalIntegerLiteral* times 8) plus the MV of *OctalDigit*.

##### B.1.2 String Literals

The syntax and semantics of 7.8.4 is extended as follows except that this extension is not allowed for strict mode code:

###### Syntax

```
EscapeSequence ::  
  CharacterEscapeSequence  
  OctalEscapeSequence  
  HexEscapeSequence  
  UnicodeEscapeSequence
```

```
OctalEscapeSequence ::  
  OctalDigit [lookahead ≠ DecimalDigit]  
  ZeroToThree OctalDigit [lookahead ≠ DecimalDigit]  
  FourToSeven OctalDigit  
  ZeroToThree OctalDigit OctalDigit
```

```
ZeroToThree :: one of  
  0 1 2 3
```

**Comment [AW105]:** Need to add new material that is not in previous versions of annex B

**Deleted:** informative

**Field Code Changed**

**Field Code Changed**

**Field Code Changed**

**Deleted:** Compatibility

**Deleted:** is

**Deleted:** s

**Deleted:** It

**Deleted:** Past editions of ECMAScript have included additional syntax and semantics for specifying octal literals and octal escape sequences. These have been removed from this edition of ECMAScript. This non-normative annex presents uniform syntax and semantics for octal literals and octal escape sequences for compatibility with some older ECMAScript programs.¶

**Deleted:** can be

**Deleted:** OctalDigit :: one of  
 0 1 2 3 4 5 6 7¶

**Deleted:** <#>The MV of NumericLiteral :: OctalIntegerLiteral is the MV of OctalIntegerLiteral.¶  
<#>The MV of OctalDigit :: 0 is 0.¶  
<#>The MV of OctalDigit :: 1 is 1.¶  
<#>The MV of OctalDigit :: 2 is 2.¶  
<#>The MV of OctalDigit :: 3 is 3.¶  
<#>The MV of OctalDigit :: 4 is 4.¶  
<#>The MV of OctalDigit :: 5 is 5.¶  
<#>The MV of OctalDigit :: 6 is 6.¶  
<#>The MV of OctalDigit :: 7 is 7.¶

**Formatted:** Font: Times New Roman, Italic

**Deleted:** can be

*FourToSeven* :: one of  
4 5 6 7

### Semantics

- The CV of *EscapeSequence* :: *OctalEscapeSequence* is the CV of the *OctalEscapeSequence*.
- The CV of *OctalEscapeSequence* :: *OctalDigit* [*lookahead*  $\notin$  *DecimalDigit*] is the character whose code unit value is the MV of the *OctalDigit*.
- The CV of *OctalEscapeSequence* :: *ZeroToThree OctalDigit* [*lookahead*  $\notin$  *DecimalDigit*] is the character whose code unit value is (8 times the MV of the *ZeroToThree*) plus the MV of the *OctalDigit*.
- The CV of *OctalEscapeSequence* :: *FourToSeven OctalDigit* is the character whose code unit value is (8 times the MV of the *FourToSeven*) plus the MV of the *OctalDigit*.
- The CV of *OctalEscapeSequence* :: *ZeroToThree OctalDigit OctalDigit* is the character whose code unit value is (64 (that is,  $8^2$ ) times the MV of the *ZeroToThree*) plus (8 times the MV of the first *OctalDigit*) plus the MV of the second *OctalDigit*.
- The MV of *ZeroToThree* :: 0 is 0.
- The MV of *ZeroToThree* :: 1 is 1.
- The MV of *ZeroToThree* :: 2 is 2.
- The MV of *ZeroToThree* :: 3 is 3.
- The MV of *FourToSeven* :: 4 is 4.
- The MV of *FourToSeven* :: 5 is 5.
- The MV of *FourToSeven* :: 6 is 6.
- The MV of *FourToSeven* :: 7 is 7.

## B.2 Additional Properties

When the ECMAScript host is a web browser the following these additional properties of the standard native objects are defined.

**Deleted:** Some implementations of

**Deleted:** have included

**Deleted:** for some

**Deleted:** This non-normative annex suggests uniform semantics for such properties without making the properties or their semantics part of this standard.

### B.2.1 escape (string)

The `escape` function is a property of the global object. It computes a new version of a String value in which certain characters have been replaced by a hexadecimal escape sequence.

For those characters being replaced whose code unit value is `0xFF` or less, a two-digit escape sequence of the form `%xx` is used. For those characters being replaced whose code unit value is greater than `0xFF`, a four-digit escape sequence of the form `%uxxxx` is used.

When the `escape` function is called with one argument *string*, the following steps are taken:

1. Call `ToString(string)`.
2. Compute the number of characters in *Result(1)*.
3. Let *R* be the empty string.
4. Let *k* be 0.
5. If *k* equals *Result(2)*, return *R*.
6. Get the character (represented as a 16-bit unsigned integer) at position *k* within *Result(1)*.
7. If *Result(6)* is one of the 69 nonblank characters  
`"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789@*_+-./"`  
then go to step 13.
8. If *Result(6)* is less than 256, go to step 11.
9. Let *S* be a String containing six characters "`%wxyz`" where *wxyz* are four hexadecimal digits encoding the value of *Result(6)*.
10. Go to step 14.
11. Let *S* be a String containing three characters "`%xy`" where *xy* are two hexadecimal digits encoding the value of *Result(6)*.
12. Go to step 14.
13. Let *S* be a String containing the single character *Result(6)*.

14. Let  $R$  be a new String value computed by concatenating the previous value of  $R$  and  $S$ .
15. Increase  $k$  by 1.
16. Go to step 5.

**NOTE** The encoding is partly based on the encoding described in RFC 1738, but the entire encoding specified in this standard is described above without regard to the contents of RFC 1738. This encoding does not reflect changes to RFC 1738 made by RFC 3986.

### B.2.2 unescape (string)

The `unescape` function is a property of the global object. It computes a new version of a String value in which each escape sequence of the sort that might be introduced by the `escape` function is replaced with the character that it represents.

When the `unescape` function is called with one argument *string*, the following steps are taken:

1. Call `ToString(string)`.
2. Compute the number of characters in  $\text{Result}(1)$ .
3. Let  $R$  be the empty String.
4. Let  $k$  be 0.
5. If  $k$  equals  $\text{Result}(2)$ , return  $R$ .
6. Let  $c$  be the character at position  $k$  within  $\text{Result}(1)$ .
7. If  $c$  is not `%`, go to step 18.
8. If  $k$  is greater than  $\text{Result}(2)-6$ , go to step 14.
9. If the character at position  $k+1$  within  $\text{Result}(1)$  is not `u`, go to step 14.
10. If the four characters at positions  $k+2$ ,  $k+3$ ,  $k+4$ , and  $k+5$  within  $\text{Result}(1)$  are not all hexadecimal digits, go to step 14.
11. Let  $c$  be the character whose code unit value is the integer represented by the four hexadecimal digits at positions  $k+2$ ,  $k+3$ ,  $k+4$ , and  $k+5$  within  $\text{Result}(1)$ .
12. Increase  $k$  by 5.
13. Go to step 18.
14. If  $k$  is greater than  $\text{Result}(2)-3$ , go to step 18.
15. If the two characters at positions  $k+1$  and  $k+2$  within  $\text{Result}(1)$  are not both hexadecimal digits, go to step 18.
16. Let  $c$  be the character whose code unit value is the integer represented by two zeroes plus the two hexadecimal digits at positions  $k+1$  and  $k+2$  within  $\text{Result}(1)$ .
17. Increase  $k$  by 2.
18. Let  $R$  be a new String value computed by concatenating the previous value of  $R$  and  $c$ .
19. Increase  $k$  by 1.
20. Go to step 5.

### B.2.3 String.prototype.substr (start, length)

The `substr` method takes two arguments, *start* and *length*, and returns a substring of the result of converting the this object to a String, starting from character position *start* and running for *length* characters (or through the end of the String if *length* is `undefined`). If *start* is negative, it is treated as  $(\text{sourceLength}+start)$  where *sourceLength* is the length of the String. The result is a String value, not a String object. The following steps are taken:

1. Call `ToString`, giving it the `this` value as its argument.
2. Call `ToInteger(start)`.
3. If *length* is `undefined`, use `+∞`; otherwise call `ToInteger(length)`.
4. Compute the number of characters in  $\text{Result}(1)$ .
5. If  $\text{Result}(2)$  is positive or zero, use  $\text{Result}(2)$ ; else use  $\max(\text{Result}(4)+\text{Result}(2), 0)$ .
6. Compute  $\min(\max(\text{Result}(3), 0), \text{Result}(4)-\text{Result}(5))$ .
7. If  $\text{Result}(6) \leq 0$ , return the empty String `""`.
8. Return a String containing  $\text{Result}(6)$  consecutive characters from  $\text{Result}(1)$  beginning with the character at position  $\text{Result}(5)$ .

The `length` property of the `substr` method is **2**.

**NOTE** The `substr` function is intentionally generic; it does not require that its `this` value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

#### B.2.4 Date.prototype.getYear ()

**NOTE** The `getFullYear` method is preferred for nearly all purposes, because it avoids the “year 2000 problem.”

When the `getYear` method is called with no arguments, the following steps are taken:

1. Let  $t$  be this time value.
2. If  $t$  is `NAN`, return `NAN`.
3. Return `YearFromTime(LocalTime( $t$ )) - 1900`.

#### B.2.5 Date.prototype.setYear (year)

**NOTE** The `setFullYear` method is preferred for nearly all purposes, because it avoids the “year 2000 problem.”

When the `setYear` method is called with one argument `year`, the following steps are taken:

1. Let  $t$  be the result of `LocalTime(this time value)`; but if this time value is `NAN`, let  $t$  be `+0`.
2. Call `ToNumber(year)`.
3. If `Result(2)` is `NAN`, set the `[[PrimitiveValue]]` internal property of the `this` value to `NAN` and return `NAN`.
4. If `Result(2)` is not `NAN` and  $0 \leq \text{ToInteger}(\text{Result}(2)) \leq 99$  then `Result(4)` is `ToInteger(Result(2)) + 1900`. Otherwise, `Result(4)` is `Result(2)`.
5. Compute `MakeDay(Result(4), MonthFromTime( $t$ ), DateFromTime( $t$ ))`.
6. Compute `UTC(MakeDate(Result(5), TimeWithinDay( $t$ )))`.
7. Set the `[[PrimitiveValue]]` internal property of the `this` value to `TimeClip(Result(6))`.
8. Return the value of the `[[PrimitiveValue]]` internal property of the `this` value.

#### B.2.6 Date.prototype.toGMTString ()

**NOTE** The property `toUTCString` is preferred. The `toGMTString` property is provided principally for compatibility with old code. It is recommended that the `toUTCString` property be used in new ECMAScript code.

The Function object that is the initial value of `Date.prototype.toGMTString` is the same Function object that is the initial value of `Date.prototype.toUTCString`.

### B.3 Other Additional Features

#### B.3.1 The `_proto_` pseudo property.

##### B.3.1.1 Object.prototype.\_proto\_

The initial value of the `_proto_` property of the Object prototype object is a data property whose initial value is `null`. This property initially has the attributes `{ [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true }`.

**Comment [AWB106]:** The section and algorithm reference in this draft are based up the ES5.1 spec. When the corresponding sections of this document are stable, this section will need to be updated.

Manipulations of this property as tracked by the Boolean valued primordial internal variable `UnderscoreProtoEnabled`. The default initial value of `UnderscoreProtoEnabled` is `true` only if this property is initially present on the primordial Object prototype object.

**NOTE** Any modification of this property or its attributes causes `UnderscoreProtoEnabled` to be set to `false`.

##### B.3.1.2 Changes To Internal Methods

The definition of the `[[Get]]` internal method given in 8.12.3 is replaced with the following:

**Comment [AWB107]:** This is anticipating specification material related to Module loader and establishing a primordial environment. The basic assumption is that a module loader must be able to disable this feature. This seems to suggest that the ability to do so must exist in module loader APIs even if this feature is not present in an implementation.

1. If  $P$  is the string value "`proto`" and UnderscoreProtoEnabled is **true**, then
  - a. Let  $desc$  be the result of calling the `[[GetProperty]]` internal method of  $O$  with property name  $P$ .
  - b. If  $desc$  is not **undefined** and was created by step 1.a to describe the property defined in B.3.1.1 then,
    - i. Return the value of the `[[Prototype]]` internal property of  $O$ .
2. Continue by executing the steps of 8.12.3 starting with step 1.

The definition of the `[[Put]]` internal method given in 8.12.5 is replaced with the following:

1. If  $P$  is the string value "`proto`" and UnderscoreProtoEnabled is **true** and  $O$  is not the standard built-in `Object` prototype object, then
  - a. Let  $desc$  be the result of calling the `[[GetProperty]]` internal method of  $O$  with property name  $P$ .
  - b. If  $desc$  is not **undefined** and was created by step 1.a to describe the property defined in B.3.1.1 then,
    - i. If the type of  $V$  is neither `Object` or `Null`, return.
    - ii. Set the value of the `[[Prototype]]` internal property of  $O$  to  $V$ .
    - iii. Return.
2. Continue by executing the steps of 8.12.5 starting with step 1.

The definition of the `[[Delete]]` internal method given in 8.12.7 is replaced with the following:

1. If UnderscoreProtoEnabled is **true** and  $P$  is the string value "`proto`" and  $O$  is the standard built-in `Object` prototype object, then
  - a. Set UnderscoreProtoEnabled to **false**.
2. Continue by executing the steps of 8.12.7 starting with step 1.

The definition of the `[[DefineOwnProperty]]` internal method given in 8.12.9 is replaced with the following:

1. If UnderscoreProtoEnabled is **true** and  $P$  is the string value "`proto`" and  $O$  is the standard built-in `Object` prototype object, then
  - a. If any attribute contained in  $Desc$  is not present or has a different value from the corresponding attribute in `{ [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true }` then,
    - i. Set UnderscoreProtoEnabled to **false**.
2. Continue by executing the steps of 8.12.9 starting with step 1.

### B.3.1.3 `proto` Object Initialisers

Definitions of two algorithms in 11.1.5 are replaced with the following:

The production `PropertyDefinitionList : PropertyDefinition` is evaluated as follows:

1. Let  $obj$  be the result of creating a new object as if by the expression `new Object()` where `Object` is the standard built-in constructor with that name.
2. Let  $propId$  be the result of evaluating `PropertyDefinition`.
3. If  $propId.name$  is the string value "`proto`" and UnderscoreProtoEnabled is **true** and `IsDataDescriptor(propId.descriptor)` is **true**, then
  - a. Let  $v$  be  $propId.descriptor.value$ .
  - b. If  $desc$  be  $propId.descriptor$
  - c. If the type of  $v$  is either `Object` or `Null`,
    - i. Set the value of the `[[Prototype]]` internal property of  $O$  to  $V$ .
    - ii. Return.
4. Call the `[[DefineOwnProperty]]` internal method of  $obj$  with arguments  $propId.name$ ,  $propId.descriptor$ , and **false**.
5. Return  $obj$ .

**Comment [AWB108]:** Note that `Object.defineProperty(obj, 'proto', desc)` nor any other call of `[[DefineOwnProperty]]` does not modify `[[Prototype]]`.

**Deleted:** `PropertyNameAndValueList`

**Deleted:** `PropertyAssignment`

**Deleted:** `PropertyAssignment`

The production

*PropertyDefinitionList* : *PropertyDefinitionList* , *PropertyDefinition*  
is evaluated as follows:

1. Let *obj* be the result of evaluating *PropertyDefinitionList*.
2. Let *propId* be the result of evaluating *PropertyDefinition*.
3. Let *previous* be the result of calling the [[GetOwnProperty]] internal method of *obj* with argument *propId.name*.
4. If *previous* is not undefined then throw a SyntaxError exception if any of the following conditions are true
  - a. This production is contained in strict code and IsDataDescriptor(*previous*) is true and IsDataDescriptor(*propId.descriptor*) is true.
  - b. IsDataDescriptor(*previous*) is true and IsAccessorDescriptor(*propId.descriptor*) is true.
  - c. IsAccessorDescriptor(*previous*) is true and IsDataDescriptor(*propId.descriptor*) is true.
  - d. IsAccessorDescriptor(*previous*) is true and IsAccessorDescriptor(*propId.descriptor*) is true and either both *previous* and *propId.descriptor* have [[Get]] fields or both *previous* and *propId.descriptor* have [[Set]] fields.
5. If *propId.name* is the string value "proto" and UnderscoreProtoEnabled is true and IsDataDescriptor(*propId.descriptor*) is true, then
  - a. Let *v* be *propId.descriptor.value*.
  - b. If *desc* be *propId.descriptor*
  - c. If the type of *v* is either Object or Null,
    - i. Set the value of the [[Prototype]] internal property of *O* to *V*.
    - ii. Return.
5. Call the [[DefineOwnProperty]] internal method of *obj* with arguments *propId.name*, *propId.descriptor*, and false.
6. Return *obj*.

**Deleted:** *PropertyNameAndValueList*  
**Deleted:** *PropertyNameAndValueList*  
**Deleted:** *PropertyAssignment*  
**Deleted:** *PropertyNameAndValueList*  
**Deleted:** *PropertyAssignment*

## Annex C (informative)

### The Strict Mode of ECMAScript

Field Code Changed
Field Code Changed
Field Code Changed

#### The strict mode restriction and exceptions

- The identifiers "implements", "interface", "let", "package", "private", "protected", "public", "static", and "yield" are classified as *FutureReservedWord* tokens within strict mode code. (7.6.12).
- A conforming implementation, when processing strict mode code, may not extend the syntax of *NumericLiteral* (7.8.3) to include *OctalIntegerLiteral* as described in B.1.1.
- A conforming implementation, when processing strict mode code (see 10.1.1), may not extend the syntax of *EscapeSequence* to include *OctalEscapeSequence* as described in B.1.2.
- Assignment to an undeclared identifier or otherwise unresolvable reference does not create a property in the global object. When a simple assignment occurs within strict mode code, its *LeftHandSide* must not evaluate to an unresolvable Reference. If it does a **ReferenceError** exception is thrown (8.9.2). The *LeftHandSide* also may not be a reference to a data property with the attribute value `{[[Writable]]:false}`, to an accessor property with the attribute value `{[[Set]]:undefined}`, nor to a non-existent property of an object whose `[[Extensible]]` internal property has the value **false**. In these cases a **TypeError** exception is thrown (11.13.1).
- The identifier `eval` or `arguments` may not appear as the *LeftHandSideExpression* of an Assignment operator (11.13) or of a *PostfixExpression* (11.3) or as the *UnaryExpression* operated upon by a Prefix Increment (11.4.4) or a Prefix Decrement (11.4.5) operator.
- Arguments objects for strict mode functions define non-configurable accessor properties named "`caller`" and "`callee`" which throw a **TypeError** exception on access (10.6).
- Arguments objects for strict mode functions do not dynamically share their array indexed property values with the corresponding formal parameter bindings of their functions. (10.6).
- For strict mode functions, if an arguments object is created the binding of the local identifier `arguments` to the arguments object is immutable and hence may not be the target of an assignment expression. (10.5).
- It is a **SyntaxError** if strict mode code contains an *ObjectLiteral* with more than one definition of any data property (11.1.5).
- It is a **SyntaxError** if the Identifier "`eval`" or the Identifier "`arguments`" occurs as the Identifier in a *PropertySetParameterList* of a *PropertyDefinition* that is contained in strict code or if its *FunctionBody* is strict code (11.1.5).
- Strict mode eval code cannot instantiate variables or functions in the variable environment of the caller to eval. Instead, a new variable environment is created and that environment is used for declaration binding instantiation for the eval code (10.4.2).
- If `this` is evaluated within strict mode code, then the `this` value is not coerced to an object. A `this` value of `null` or `undefined` is not converted to the global object and primitive values are not converted to wrapper objects. The `this` value passed via a function call (including calls made using `Function.prototype.apply` and `Function.prototype.call`) do not coerce the passed this value to an object (10.4.3, 11.1.1, 15.3.4.3, 15.3.4.4).
- When a `delete` operator occurs within strict mode code, a **SyntaxError** is thrown if its *UnaryExpression* is a direct reference to a variable, function argument, or function name(11.4.1).

Deleted: 7

Deleted: *PropertyAssignment*

- When a `delete` operator occurs within strict mode code, a **TypeError** is thrown if the property to be deleted has the attribute `{ [[Configurable]]:false }` (11.4.1).
- It is a **SyntaxError** if a `VariableDeclaration` or `VariableDeclarationNoIn` occurs within strict code and its `Identifier` is `eval` or `arguments` (12.2.1).
- Strict mode code may not include a `WithStatement`. The occurrence of a `WithStatement` in such a context is an **SyntaxError** (12.10).
- It is a **SyntaxError** if a `TryStatement` with a `Catch` occurs within strict code and the `Identifier` of the `Catch` production is `eval` or `arguments` (12.14.1)
- It is a **SyntaxError** if the identifier `eval` or `arguments` appears within a `FormalParameterList` of a strict mode `FunctionDeclaration` or `FunctionExpression` (13.1)
- A strict mode function may not have two or more formal parameters that have the same name. An attempt to create such a function using a `FunctionDeclaration`, `FunctionExpression`, or `Function` constructor is a **SyntaxError** (13.1, 15.3.2).
- An implementation may not extend, beyond that defined in this specification, the meanings within strict mode functions of properties named `caller` or `arguments` of function instances. ECMAScript code may not create or modify properties with these names on function objects that correspond to strict mode functions (10.6, 13.2, 15.3.4.5.3).
- It is a **SyntaxError** to use within strict mode code the identifiers `eval` or `arguments` as the `Identifier` of a `FunctionDeclaration` or `FunctionExpression` or as a formal parameter name (13.1). Attempting to dynamically define such a strict mode function using the `Function` constructor (15.3.2) will throw a **SyntaxError** exception.

DRY

## Annex D (informative)

### Corrections and Clarifications Possible Compatibility Impact

**Deleted:** in the 5<sup>th</sup> Edition with

**Deleted:** 3<sup>rd</sup> Edition

#### In the 6<sup>th</sup> Edition

15.9.1.15: If a time zone offset is not present, the local time zone is used. Edition 5.1 incorrectly stated that a missing time zone should be interpreted as "z".

**Formatted:** Font: (Default) Courier New

15.9.5.2: Previous editions did not specify the value returned by Date.prototype.toString when this time value is NaN. The 6<sup>th</sup> Edition specifies the result to be the String value is "Invalid Date".

#### In the 5<sup>th</sup> Edition

Throughout: In the Edition 3 specification the meaning of phrases such as "as if by the expression `new Array()`" are subject to misinterpretation. In the Edition 5 specification text for all internal references and invocations of standard built-in objects and methods has been clarified by making it explicit that the intent is that the actual built-in object is to be used rather than the current dynamic value of the correspondingly named property.

11.8.1: ECMAScript generally uses a left to right evaluation order, however the Edition 3 specification language for the > and <= operators resulted in a partial right to left order. The specification has been corrected for these operators such that it now specifies a full left to right evaluation order. However, this change of order is potentially observable if side-effects occur during the evaluation process.

**Deleted:** 2

**Deleted:** , 11.8.3, 11.8.5

11.1.4: Edition 5 clarifies the fact that a trailing comma at the end of an `ArrayInitialiser` does not add to the length of the array. This is not a semantic change from Edition 3 but some implementations may have previously misinterpreted this.

11.2.3: Edition 5 reverses the order of steps 2 and 3 of the algorithm. The original order as specified in Editions 1 through 3 was incorrectly specified such that side-effects of evaluating `Arguments` could affect the result of evaluating `MemberExpression`.

12.4: In Edition 3, an object is created, as if by `new Object()` to serve as the scope for resolving the name of the exception parameter passed to a `catch` clause of a `try` statement. If the actual exception object is a function and it is called from within the `catch` clause, the scope object will be passed as the `this` value of the call. The body of the function can then define new properties on its `this` value and those property names become visible identifiers bindings within the scope of the `catch` clause after the function returns. In Edition 5, when an exception parameter is called as a function, `undefined` is passed as the `this` value.

13: In Edition 3, the algorithm for the production `FunctionExpression` with an `Identifier` adds an object created as if by `new Object()` to the scope chain to serve as a scope for looking up the name of the function. The identifier resolution rules (10.1.4 in Edition 3) when applied to such an object will, if necessary, follow the object's prototype chain when attempting to resolve an identifier. This means all the properties of `Object.prototype` are visible as identifiers within that scope. In practice most implementations of Edition 3 have not implemented this semantics. Edition 5 changes the specified semantics by using a Declarative Environment Record to bind the name of the function.

14: In Edition 3, the algorithm for the production `SourceElements : SourceElements SourceElement` did not correctly propagate statement result values in the same manner as `Block`. This could result in the `eval` function producing an incorrect result when evaluating a `Program` text. In practice most implementations of Edition 3 have implemented the correct propagation rather than what was specified in Edition 5.



INTERNATIONAL

15.10.6: `RegExp.prototype` is now a `RegExp` object rather than an instance of `Object`. The value of its `[[Class]]` internal property which is observable using `Object.prototype.toString` is now “`RegExp`” rather than “`Object`”.

DRAFT

DRAFT

## Annex E (informative)

### **Additions and Changes that Introduce Incompatibilities with Prior Editions**

<b>Field Code Changed</b>
<b>Field Code Changed</b>
<b>Field Code Changed</b>
<b>Deleted: in the 5<sup>th</sup> Edition</b>
<b>Deleted: the 3<sup>rd</sup></b>

#### In the 6<sup>th</sup> Edition

#### In the 5<sup>th</sup> Edition

7.1: Unicode format control characters are no longer stripped from ECMAScript source text before processing. In Edition 5, if such a character appears in a *StringLiteral* or *RegularExpressionLiteral* the character will be incorporated into the literal where in Edition 3 the character would not be incorporated into the literal.

7.2: Unicode character <BOM> is now treated as whitespace and its presence in the middle of what appears to be an identifier could result in a syntax error which would not have occurred in Edition 3

7.3: Line terminator characters that are preceded by an escape sequence are now allowed within a string literal token. In Edition 3 a syntax error would have been produced.

7.8.5: Regular expression literals now return a unique object each time the literal is evaluated. This change is detectable by any programs that test the object identity of such literal values or that are sensitive to the shared side effects.

7.8.5: Edition 5 requires early reporting of any possible RegExp constructor errors that would be produced when converting a *RegularExpressionLiteral* to a RegExp object. Prior to Edition 5 implementations were permitted to defer the reporting of such errors until the actual execution time creation of the object.

7.8.5: In Edition 5 unescaped "/" characters may appear as a *CharacterClass* in a regular expression literal. In Edition 3 such a character would have been interpreted as the final character of the literal.

10.4.2: In Edition 5, indirect calls to the **eval** function use the global environment as both the variable environment and lexical environment for the eval code. In Edition 3, the variable and lexical environments of the caller of an indirect **eval** was used as the environments for the eval code.

15.4.4: In Edition 5 all methods of **Array.prototype** are intentionally generic. In Edition 3 **toString** and **toLocaleString** were not generic and would throw a **TypeError** exception if applied to objects that were not instances of Array.

10.6: In Edition 5 the array indexed properties of argument objects that correspond to actual formal parameters are enumerable. In Edition 3, such properties were not enumerable.

10.6: In Edition 5 the value of the **[[Class]]** internal property of an arguments object is "**Arguments**". In Edition 3, it was "**object**". This is observable if **toString** is called as a method of an arguments object.

12.6.4: for-in statements no longer throw a **TypeError** if the **in** expression evaluates to **null** or **undefined**. Instead, the statement behaves as if the value of the expression was an object with no enumerable properties.

15: In Edition 5, the following new properties are defined on built-in objects that exist in Edition 3:  
**Object.getPrototypeOf**, **Object.getOwnPropertyDescriptor**, **Object.getOwnPropertyNames**,  
**Object.create**, **Object.defineProperty**, **Object.defineProperties**, **Object.seal**,  
**Object.freeze**, **Object.preventExtensions**, **Object.isSealed**, **Object.isFrozen**,

```
Object.isExtensible, Object.keys, Function.prototype.bind, Array.prototype.indexOf,  
Array.prototype.lastIndexOf, Array.prototype.every, Array.prototype.some,  
Array.prototype.forEach, Array.prototype.map, Array.prototype.filter,  
Array.prototype.reduce, Array.prototype.reduceRight, String.prototype.trim, Date.now,  
Date.prototype.toISOString, Date.prototype.toJSON.
```

15: Implementations are now required to ignore extra arguments to standard built-in methods unless otherwise explicitly specified. In Edition 3 the handling of extra arguments was unspecified and implementations were explicitly allowed to throw a **TypeError** exception.

15.1.1: The value properties **NaN**, **Infinity**, and **undefined** of the Global Object have been changed to be read-only properties.

15.1.2.1: Implementations are no longer permitted to restrict the use of eval in ways that are not a direct call. In addition, any invocation of eval that is not a direct call uses the global environment as its variable environment rather than the caller's variable environment.

15.1.2.2: The specification of the function **parseInt** no longer allows implementations to treat Strings beginning with a 0 character as octal values.

15.3.4.3: In Edition 3, a **TypeError** is thrown if the second argument passed to **Function.prototype.apply** is neither an array object nor an arguments object. In Edition 5, the second argument may be any kind of generic array-like object that has a valid **length** property.

15.3.4.3, 15.3.4.4: In Edition 3 passing **undefined** or **null** as the first argument to either **Function.prototype.apply** or **Function.prototype.call** causes the global object to be passed to the indirectly invoked target function as the **this** value. If the first argument is a primitive value the result of calling **ToObject** on the primitive value is passed as the **this** value. In Edition 5, these transformations are not performed and the actual first argument value is passed as the **this** value. This difference will normally be unobservable to existing ECMAScript Edition 3 code because a corresponding transformation takes place upon activation of the target function. However, depending upon the implementation, this difference may be observable by host object functions called using **apply** or **call**. In addition, invoking a standard built-in function in this manner with **null** or **undefined** passed as the **this** value will in many cases cause behaviour in Edition 5 implementations that differ from Edition 3 behaviour. In particular, in Edition 5 built-in functions that are specified to actually use the passed **this** value as an object typically throw a **TypeError** exception if passed **null** or **undefined** as the **this** value.

15.3.5.2: In Edition 5, the **prototype** property of Function instances is not enumerable. In Edition 3, this property was enumerable.

15.5.5.2: In Edition 5, the individual characters of a String object's **[PrimitiveValue]** may be accessed as array indexed properties of the String object. These properties are non-writable and non-configurable and shadow any inherited properties with the same names. In Edition 3, these properties did not exist and ECMAScript code could dynamically add and remove writable properties with such names and could access inherited properties with such names.

15.9.4.2: **Date.parse** is now required to first attempt to parse its argument as an ISO format string. Programs that use this format but depended upon implementation specific behaviour (including failure) may behave differently.

15.10.2.12: In Edition 5, \s now additionally matches <BOM>.

15.10.4.1: In Edition 3, the exact form of the String value of the **source** property of an object created by the **RegExp** constructor is implementation defined. In Edition 5, the String must conform to certain specified requirements and hence may be different from that produced by an Edition 3 implementation.

15.10.6.4: In Edition 3, the result of `RegExp.prototype.toString` need not be derived from the value of the `RegExp` object's `source` property. In Edition 5 the result must be derived from the `source` property in a specified manner and hence may be different from the result produced by an Edition 3 implementation.

15.11.2.1, 15.11.4.3: In Edition 5, if an initial value for the `message` property of an `Error` object is not specified via the `Error` constructor the initial value of the property is the empty String. In Edition 3, such an initial value is implementation defined.

15.11.4.4: In Edition 3, the result of `Error.prototype.toString` is implementation defined. In Edition 5, the result is fully specified and hence may differ from some Edition 3 implementations.

15.12: In Edition 5, the name `JSON` is defined in the global environment. In Edition 3, testing for the presence of that name will show it to be undefined unless it is defined by the program or implementation.

DRAFT

DRAFT

## Annex F (informative)

### Technically Significant Corrections and Clarifications in the 5.1 Edition

Field Code Changed  
Field Code Changed  
Field Code Changed

7.8.4: CV definitions added for *DoubleStringCharacter* :: *LineContinuation* and *SingleStringCharacter* :: *LineContinuation*.

10.2.1.1.3: The argument *S* is not ignored. It controls whether an exception is thrown when attempting to set an immutable binding.

10.2.1.2.2: In algorithm step 5, **true** is passed as the last argument to `[[DefineOwnProperty]]`.

10.5: Former algorithm step 5.e is now 5.f and a new step 5.e was added to restore compatibility with 3<sup>rd</sup> Edition when redefining global functions.

11.5.3: In the final bullet item, use of IEEE 754 round-to-nearest mode is specified.

12.6.3: Missing `ToBoolean` restored in step 3.a.ii of both algorithms.

12.6.4: Additional final sentences in each of the last two paragraphs clarify certain property enumeration requirements.

12.7, 12.8, 12.9: BNF modified to clarify that a `continue` or `break` statement without an *Identifier* or a `return` statement without an *Expression* may have a *LineTerminator* before the semi-colon.

12.14: Step 3 of algorithm 1 and step 2.a of algorithm 3 are corrected such that the value field of *B* is passed as a parameter rather than *B* itself.

15.1.2.2: In step 2 of algorithm, clarify that *S* may be the empty string.

15.1.2.3: In step 2 of algorithm clarify that *trimmedString* may be the empty string.

15.1.3: Added notes clarifying that ECMAScript's URI syntax is based upon RFC 2396 and not the newer RFC 3986. In the algorithm for `Decode`, a step was removed that immediately preceded the current step 4.d.vii.10.a because it tested for a condition that cannot occur.

15.2.3.7: Corrected use of variable *P* in steps 5 and 6 of algorithm.

15.2.4.2: Edition 5 handling of `undefined` and `null` as `this` value caused existing code to fail. Specification modified to maintain compatibility with such code. New steps 1 and 2 added to the algorithm.

15.3.4.3: Steps 5 and 7 of Edition 5 algorithm have been deleted because they imposed requirements upon the *argArray* argument that are inconsistent with other uses of generic array-like objects.

15.4.4.12: In step 9.a, incorrect reference to `relativeStart` was replaced with a reference to `actualStart`.

15.4.4.15: Clarified that the default value for *fromIndex* is the length minus 1 of the array.

15.4.4.18: In step 9 of the algorithm, `undefined` is now the specified return value.

15.4.4.22: In step 9.c.ii the first argument to the `[[Call]]` internal method has been changed to `undefined` for consistency with the definition of `Array.prototype.reduce`.

15.4.5.1: In Algorithm steps 3.I.ii and 3.I.iii the variable name was inverted resulting in an incorrectly inverted test.

15.5.4.9: Normative requirement concerning canonically equivalent strings deleted from paragraph following algorithm because it is listed as a recommendation in NOTE 2.

15.5.4.14: In `split` algorithm step 11.a and 13.a, the positional order of the arguments to `SplitMatch` was corrected to match the actual parameter signature of `SplitMatch`. In step 13.a.iii.7.d, `lengthA` replaces `A.length`.

15.5.5.2: In first paragraph, removed the implication that the individual character property access had “array index” semantics. Modified algorithm steps 3 and 5 such that they do not enforce “array index” requirement.

15.9.1.15: Specified legal value ranges for fields that lacked them. Eliminated “time-only” formats. Specified default values for all optional fields.

15.10.2.2: The step numbers of the algorithm for the internal closure produced by step 2 were incorrectly numbered in a manner that implied that they were steps of the outer algorithm.

15.10.2.6: In the abstract operation `IsWordChar` the first character in the list in step 3 is “`a`” rather than “`A`”.

15.10.2.8: In the algorithm for the closure returned by the abstract operation `CharacterSetMatcher`, the variable defined by step 3 and passed as an argument in step 4 was renamed to `ch` in order to avoid a name conflict with a formal parameter of the closure.

15.10.6.2: Step 9.e was deleted because it performed an extra increment of `i`.

15.11.1.1: Removed requirement that the `message` own property is set to the empty String when the `message` argument is `undefined`.

15.11.1.2: Removed requirement that the `message` own property is set to the empty String when the `message` argument is `undefined`.

15.11.4.4: Steps 6-10 modified/added to correctly deal with missing or empty `message` property value.

15.11.1.2: Removed requirement that the `message` own property is set to the empty String when the `message` argument is `undefined`.

15.12.3: In step 10.b.iii of the `J4` internal operation, the last element of the concatenation is “[”.

B.2.1: Added to NOTE that the encoding is based upon RFC 1738 rather than the newer RFC 3986.

Annex C: An item was added corresponding to 7.6.12 regarding `FutureReservedWords` in strict mode.

## Bibliography

- [1] IEEE Std 754-2008: IEEE Standard for Floating-Point Arithmetic. Institute of Electrical and Electronic Engineers, New York (2008)
- [2] The Unicode Consortium. The Unicode Standard, Version 3.0, defined by: The Unicode Standard, Version 3.0 (Reading, MA, Addison-Wesley, 2000. ISBN 0-201-61633-5)
- [3] Unicode Inc. (2010), Unicode Technical Report #15: Unicode Normalization Forms
- [4] ISO 8601:2004(E) *Data elements and interchange formats – Information interchange -- Representation of dates and times*
- [5] RFC 1738 "Uniform Resource Locators (URL)", available at <<http://tools.ietf.org/html/rfc1738>>
- [6] RFC 2396 "Uniform Resource Identifiers (URI): Generic Syntax", available at <<http://tools.ietf.org/html/rfc2396>>
- [7] RFC 3629 "UTF-8, a transformation format of ISO 10646", available at <<http://tools.ietf.org/html/rfc3629>>
- [8] RFC 4627 "The application/json Media Type for JavaScript Object Notation (JSON)", available at <<http://tools.ietf.org/html/rfc4627>>

DRAFT