ecma INTERNATIONAL

Ecma/TC39/2012/081

Draft Standard ECMA-262

6th Edition / Draft November 22, 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: November 22, 2012 Draft

Standard

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

Contents

Page

Introduction vii

1 Scope 1

2 Conformance 1

3 Normative references 1

4 Overview 1
4.1 Web Scripting 2
4.2 Language Overview 2
4.2.1 Objects 3
4.2.2 The Strict Variant of ECMAScript 4
4.3 Terms and definitions 4

5 Notational Conventions 7
5.1 Syntactic and Lexical Grammars 7
5.1.1 Context-Free Grammars 7
5.1.2 The Lexical and RegExp Grammars 8
5.1.3 The Numeric String Grammar 8
5.1.4 The Syntactic Grammar 8
5.1.5 The JSON Grammar 9
5.1.6 Grammar Notation 9
5.2 Algorithm Conventions 12
5.3 Static Semantic Rules 13

6 Source Text 14

7 Lexical Conventions 15
7.1 Unicode Format-Control Characters 16
7.2 White Space 16
7.3 Line Terminators 17
7.4 Comments 17
7.5 Tokens 18
7.6 Identifier Names and Identifiers 19
7.6.1 Reserved Words 20
7.7 Punctuators 21
7.8 Literals 21
7.8.1 Null Literals 21
7.8.2 Boolean Literals 21
7.8.3 Numeric Literals 22
7.8.4 String Literals 24
7.8.5 Regular Expression Literals 27
7.8.6 Template Literal Lexical Components 28
7.9 Automatic Semicolon Insertion 30
7.9.1 Rules of Automatic Semicolon Insertion 30
7.9.2 Examples of Automatic Semicolon Insertion 31

8 Types 32
8.1 ECMAScript Language Types 32
8.1.1 The Undefined Type 32
8.1.2 The Null Type 32
8.1.3 The Boolean Type 32
8.1.4 The String Type 33
8.1.5 The Number Type 33
8.1.6 The Object Type 34

8.2 ECMAScript Specification Types 42
8.2.1 Data Blocks 42
8.2.2 The List and Record Specification Type 42
8.2.3 The Completion Record Specification Type 43
8.2.4 The Reference Specification Type 44
8.2.5 The Property Descriptor Specification Types 45
8.2.6 The Lexical Environment and Environment Record Specification Types 48
8.3 Ordinary Object Internal Methods and Internal Data Properties 48
8.3.1 [[GetInheritance]] ( ) 48
8.3.2 [[SetInheritance]] (V) 48
8.3.3 [[IsExtensible]] ( ) 48
8.3.4 [[PreventExtensions]] ( ) 49
8.3.5 [[HasOwnProperty]] (P) 49
8.3.6 [[GetOwnProperty]] (P) 49
8.3.7 [[GetP]] (P, Receiver) 49
8.3.8 [[SetP] (P, V, Receiver) 50
8.3.9 [[Delete]] (P) 51
8.3.10 [[DefineOwnProperty]] (P, Desc) 51
8.3.11 [[Enumerate]] () 53
8.3.12 [[Keys]] ( ) 54
8.3.13 [[OwnPropertyKeys]] ( ) 54
8.3.14 [[Freeze]] ( ) 54
8.3.15 [[Seal]] ( ) 54
8.3.16 [[IsFrozen]] ( ) 55
8.3.17 [[IsSealed]] ( ) 55
8.3.18 ObjectCreate Abstract Operation 55
8.3.19 Ordinary Function Objects 55
8.4 Built-in Exotic Object Internal Methods and Data Fields 57
8.4.1 Bound Function Exotic Objects 58
8.4.2 Array Exotic Objects 59
8.4.3 String Exotic Objects 61
8.4.4 Exotic Symbol Objects 63
8.4.5 Exotic Arguments Objects 66
8.4.5 Typed Array Exotic Objects 66
8.4.6 Built-in Function Objects 66
8.5 Proxy Object Internal Methods and Internal Data Properties 66
8.5.1 [[GetInheritance]] ( ) 67
8.5.3 [[IsExtensible]] ( ) 67
8.5.4 [[PreventExtensions]] ( ) 68
8.5.5 [[HasOwnProperty]] (P) 68
8.5.6 [[GetOwnProperty]] (P) 69
8.5.7 [[GetP]] (P, Receiver) 70
8.5.8 [[SetP] (P, V, Receiver) 70
8.5.9 [[Delete]] (P) 71
8.5.10 [[DefineOwnProperty]] (P, Desc) 71
8.5.11 [[Enumerate]] () 72
8.5.12 [[Keys]] ( ) 72
8.5.13 [[OwnPropertyKeys]] ( ) 73
8.5.14 [[Freeze]] ( ) 73
8.5.15 [[Seal]] ( ) 73
8.5.16 [[IsFrozen]] ( ) 73
8.5.17 [[IsSealed]] ( ) 73

9 Abstract Operations 73
9.1 Type Conversion and Testing 73
9.1.1 ToPrimitive 74
9.1.2 ToBoolean 75
9.1.3 ToNumber 75
9.1.4 ToInteger 78
9.1.5 ToInt32: (Signed 32 Bit Integer) 78

9.1.6 ToUint32: (Unsigned 32 Bit Integer) 79
9.1.7 ToUint16: (Unsigned 16 Bit Integer) 79
9.1.8 ToString 79
9.1.9 ToObject 81
9.1.10 ToPropertyKey 81
9.2 Testing and Comparison Operations 81
9.2.1 CheckObjectCoercible 81
9.2.2 IsCallable 82
9.2.3 The SameValue Algorithm 82
9.2.4 IsConstructor 82
9.3 Operations on Objects 82
9.3.1 Get (O, P) 82
9.3.2 Put (O, P, V, Throw) 83
9.3.3 CreateOwnDataProperty (O, P, V) 83
9.3.4 DefinePropertyOrThrow (O, P, desc) 83
9.3.5 DeletePropertyOrThrow (O, P) 83
9.3.6 HasProperty (O, P) 84
9.3.7 GetMethod (O, P) 84
9.3.8 Invoke(O,P [,args]) 84
9.3.9 MakeObjectSecure (O, immutable) 84
9.3.10 TestIfSecureObject (O, immutable) 85
9.3.11 CreateArrayFromList (elements) 86
9.3.12 OrdinaryHasInstance (C, O) 86

10 Executable Code and Execution Contexts 86
10.1 Types of Executable Code 86
10.1.1 Strict Mode Code 87
10.1.2 Non-ECMAScript Functions 87
10.2 Lexical Environments 87
10.2.1 Environment Records 88
10.2.2 Lexical Environment Operations 99
10.3 Code Realms 100
10.4 Execution Contexts 101
10.4.1 Identifier Resolution 102
10.4.2 GetThisEnvironment 103
10.4.3 This Resolution 103
10.4.4 GetGlobalObject 103
10.5 Declaration Binding Instantiation 103
10.5.1 Global Declaration Instantiation 103
10.5.2 Module Declaration Instantiation 104
10.5.3 Function Declaration Instantiation 105
10.5.4 Block Declaration Instantiation 106
10.5.5 Eval Declaration Instantiation 107
10.6 Arguments Object 107

11 Expressions 110
11.1 Primary Expressions 110
11.1.1 The `this` Keyword 111
11.1.2 Identifier Reference 111
11.1.3 Literals 111
11.1.4 Array Initialiser 112
11.1.5 Object Initialiser 115
11.1.6 Function Defining Expressions 118
11.1.7 Generator Comprehensions 118
11.1.8 Regular Expression Literals 118
11.1.9 Template Literals 119
11.1.10 The Grouping Operator 122
11.2 Left-Hand-Side Expressions 123
11.2.1 Property Accessors 124
11.2.2 The `new` Operator 125

11.2.3 Function Calls ... 126
11.2.4 The `super` Keyword ... 127
11.2.5 Argument Lists ... 128
11.2.6 Tagged Templates ... 129
11.3 Postfix Expressions ... 129
11.3.1 Postfix Increment Operator ... 130
11.3.2 Postfix Decrement Operator ... 130
11.4 Unary Operators ... 130
11.4.1 The `delete` Operator ... 131
11.4.2 The `void` Operator ... 132
11.4.3 The `typeof` Operator ... 132
11.4.4 Prefix Increment Operator ... 132
11.4.5 Prefix Decrement Operator ... 133
11.4.6 Unary + Operator ... 133
11.4.7 Unary – Operator ... 133
11.4.8 Bitwise NOT Operator (~) ... 133
11.4.9 Logical NOT Operator (!) ... 134
11.5 Multiplicative Operators ... 134
11.5.1 Applying the * Operator ... 134
11.5.2 Applying the / Operator ... 135
11.5.3 Applying the % Operator ... 135
11.6 Additive Operators ... 136
11.6.1 The Addition operator (+) ... 136
11.6.2 The Subtraction Operator (–) ... 137
11.6.3 Applying the Additive Operators to Numbers ... 137
11.7 Bitwise Shift Operators ... 137
11.7.1 The Left Shift Operator (<<) ... 138
11.7.2 The Signed Right Shift Operator (>>) ... 138
11.7.3 The Unsigned Right Shift Operator (>>>) ... 138
11.8 Relational Operators ... 139
11.8.1 Runtime Semantics ... 140
11.9 Equality Operators ... 142
11.9.1 Runtime Semantics ... 143
11.10 Binary Bitwise Operators ... 145
11.11 Binary Logical Operators ... 146
11.12 Conditional Operator (? :) ... 147
11.13 Assignment Operators ... 148
Static Semantics ... 148
Runtime Semantics ... 149
11.13.1 Destructuring Assignment ... 149
11.14 Comma Operator (,) ... 153

12 Statements and Declarations ... 154
Static Semantics ... 154
Runtime Semantics ... 154
12.1 Block ... 155
12.2 Declarations and the Variable Statement ... 158
12.2.1 Let and Const Declarations ... 158
12.2.2 Variable Statement ... 161
12.2.4 Destructuring Binding Patterns ... 163
12.3 Empty Statement ... 168
12.4 Expression Statement ... 168
12.5 The `if` Statement ... 168
12.6 Iteration Statements ... 169
12.6.1 The `do-while` Statement ... 170
12.6.2 The `while` Statement ... 170
12.6.3 The `for` Statement ... 171
12.6.4 The `for-in` and `for-of` Statements ... 172

12.7 The `continue` Statement ... 176
12.8 The `break` Statement ... 176
12.9 The `return` Statement ... 177
12.10 The `with` Statement ... 177
12.11 The `switch` Statement ... 178
12.12 Labelled Statements ... 182
12.13 The `throw` Statement ... 183
12.14 The `try` Statement ... 183
12.15 The `debugger` statement ... 185

13 Functions and Generators ... 186
13.1 Function Definitions ... 186
13.2 Arrow Function Definitions ... 191
13.3 Method Definitions ... 194
13.4 Generator Definitions ... 198
13.5 Class Definitions ... 199
13.6 Creating Function Objects and Constructors ... 202
13.7 Tail Position Calls ... 204

14 Scripts and Modules ... 204
14.1 Script ... 204
14.1.1 Directive Prologues and the Use Strict Directive ... 207
14.2 Modules ... 207

15 Standard Built-in ECMAScript Objects ... 207
15.1 The Global Object ... 208
15.1.1 Value Properties of the Global Object ... 209
15.1.2 Function Properties of the Global Object ... 209
15.1.3 URI Handling Function Properties ... 212
15.1.4 Constructor Properties of the Global Object ... 216
15.1.5 Other Properties of the Global Object ... 218
15.2 Object Objects ... 218
15.2.1 The Object Constructor Called as a Function ... 218
15.2.2 The Object Constructor ... 218
15.2.3 Properties of the Object Constructor ... 218
15.2.4 Properties of the Object Prototype Object ... 221
15.2.5 Properties of Object Instances ... 223
15.3 Function Objects ... 223
15.3.1 The Function Constructor Called as a Function ... 223
15.3.2 The Function Constructor ... 224
15.3.3 Properties of the Function Constructor ... 224
15.3.4 Properties of the Function Prototype Object ... 225
15.3.5 Properties of Function Instances ... 227
15.4 Array Objects ... 227
15.4.1 The Array Constructor Called as a Function ... 228
15.4.2 The Array Constructor ... 228
15.4.3 Properties of the Array Constructor ... 229
15.4.4 Properties of the Array Prototype Object ... 230
15.4.5 Properties of Array Instances ... 249
15.4.6 Array Iterator Object Structure ... 249
15.5 String Objects ... 251
15.5.1 The String Constructor Called as a Function ... 251
15.5.2 The String Constructor ... 251
15.5.3 Properties of the String Constructor ... 251
15.5.4 Properties of the String Prototype Object ... 253
15.5.5 Properties of String Instances ... 265
15.6 Boolean Objects ... 265
15.6.1 The Boolean Constructor Called as a Function ... 265
15.6.2 The Boolean Constructor ... 265
15.6.3 Properties of the Boolean Constructor ... 266

15.6.4 Properties of the Boolean Prototype Object ... 266
15.6.5 Properties of Boolean Instances ... 267
15.7 Number Objects ... 267
15.7.1 The Number Constructor Called as a Function ... 267
15.7.2 The Number Constructor ... 267
15.7.3 Properties of the Number Constructor ... 267
15.7.4 Properties of the Number Prototype Object ... 269
15.7.5 Properties of Number Instances ... 273
15.8 The Math Object ... 273
15.8.1 Value Properties of the Math Object ... 274
15.8.2 Function Properties of the Math Object ... 275
15.9 Date Objects ... 282
15.9.1 Overview of Date Objects and Definitions of Abstract Operations ... 282
15.9.2 The Date Constructor Called as a Function ... 287
15.9.3 The Date Constructor ... 287
15.9.4 Properties of the Date Constructor ... 288
15.9.5 Properties of the Date Prototype Object ... 289
15.9.6 Properties of Date Instances ... 297
15.10 RegExp (Regular Expression) Objects ... 297
15.10.1 Patterns ... 298
15.10.2 Pattern Semantics ... 300
15.10.3 The RegExp Constructor Called as a Function ... 311
15.10.4 The RegExp Constructor ... 311
15.10.5 Properties of the RegExp Constructor ... 313
15.10.6 Properties of the RegExp Prototype Object ... 313
15.10.7 Properties of RegExp Instances ... 315
15.11 Error Objects ... 315
15.11.1 The Error Constructor Called as a Function ... 315
15.11.2 The Error Constructor ... 316
15.11.3 Properties of the Error Constructor ... 316
15.11.4 Properties of the Error Prototype Object ... 316
15.11.5 Properties of Error Instances ... 317
15.11.6 Native Error Types Used in This Standard ... 317
15.11.7 *NativeError* Object Structure ... 318
15.12 The JSON Object ... 319
15.12.1 The JSON Grammar ... 320
15.12.2 JSON.parse (text [, reviver]) ... 321
15.12.3 JSON.stringify (value [, replacer [, space]]) ... 322
15.13 Binary Data Objects ... 327
15.13.1 The BinaryData Module ... 327
15.13.2 The BinaryData.Type Object ... 327
15.13.3 The BinaryData.ArrayType Object ... 327
15.13.4 The BinaryData.StructType Object ... 327
15.13.5 ArrayBufferObjects ... 327
15.13.6 *TypeArray* Object Structures ... 328
15.13.7 DataView Objects ... 333
15.14 Map Objects ... 337
15.14.1 Abstract Operations For Map Objects ... 337
15.14.2 The Map Constructor Called as a Function ... 337
15.14.3 The Map Constructor ... 338
15.14.4 Properties of the Map Constructor ... 338
15.14.5 Properties of the Map Prototype Object ... 338
15.14.6 Properties of Map Instances ... 341
15.14.7 Map Iterator Object Structure ... 341
15.15 WeakMap Objects ... 343
15.15.1 Abstract Operations For WeakMap Objects ... 343
15.15.2 The WeakMap Constructor Called as a Function ... 344
15.15.3 The WeakMap Constructor ... 344
15.15.4 Properties of the WeakMap Constructor ... 344

15.15.5 Properties of the WeakMap Prototype Object 345
15.15.6 Properties of WeakMap Instances 346
15.16 Set Objects 346
15.16.1 Abstract Operations For Set Objects 346
15.16.2 The Set Constructor Called as a Function 347
15.16.3 The Set Constructor 347
15.16.4 Properties of the Set Constructor 348
15.16.5 Properties of the Set Prototype Object 348
15.16.6 Properties of Set Instances 350
15.16.7 Set Iterator Object Structure 350
15.17 The Reflect Module 351
15.17.1 Exported Function Properties Reflecting the Essentional Internal Methods 351
15.17.2 Exported Function Properties Derived from the Essentional Internal Methods 354
15.18 Proxy Objects 355

16 Errors 355

Annex A (informative) Grammar Summary 357
A.1 Lexical Grammar 357
A.2 Number Conversions 363
A.3 Expressions 364
A.4 Statements 368
A.5 Functions and Scripts 370
A.6 Universal Resource Identifier Character Classes 371
A.7 Regular Expressions 371
A.8 JSON 373
A.8.1 JSON Lexical Grammar 373
A.8.2 JSON Syntactic Grammar 374

Annex B (normative) Additional ECMAScript Features for Web Browsers 377
B.1 Additional Syntax 377
B.1.1 Numeric Literals 377
B.1.2 String Literals 377
B.2 Additional Properties 378
B.2.1 Additional Properties of the Global Object 378
B.2.2 Additional Properties of the String.prototype Object 379
B.2.3 Additional Properties of the Date.prototype Object 382
B.3 Other Additional Features 383
B.3.1 The __proto__ pseudo property. 383

Annex C (informative) The Strict Mode of ECMAScript 385

Annex D (informative) Corrections and Clarifications with Possible Compatibility Impact 387
In Edition 6 387
In 5.1 Edition 5.1 387
In 5th Edition 5 388

Annex E (informative) Additions and Changes that Introduce Incompatibilities with Prior Editions 391
In the 6th Edition 391
In the 5th Edition 391

Annex F (informative) Static Semantic Rule Cross Reference 395

Scrap Heap 397
8.3.10 [[Enumerate]] (includePrototype, onlyEnumerable) 398
10.5.3 Function Declaration Instantiation 399

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[1] as the fourth edition of ECMAScript, it informs continuing evolution of the language. The fifth edition of ECMAScript (published as ECMA-262 5th 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>.

[1] 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."

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 5.1.0 or later and ISO/IEC 10646. If the adopted ISO/IEC 10646-1 subset is not otherwise specified, it is presumed to be the Unicode set, collection 10646.

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 10646:2003: *Information Technology – Universal Multiple-Octet Coded Character Set (UCS) plus Amendment 1:2005, Amendment 2:2006, Amendment 3:2008, and Amendment 4:2008*, plus additional amendments and corrigenda, or successor

The Unicode Standard, Version 5.0, as amended by Unicode 5.1.0, or successor

Unicode Standard Annex #15, Unicode Normalization Forms, version Unicode 5.1.0, or successor

Unicode Standard Annex #31, Unicode Identifiers and Pattern Syntax, version Unicode 5.1.0, or successor.

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 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 used as a scripting language, but has become widely used as a general purpose programming language.

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 is now used both as a general propose programming language and to provide core scripting capabilities for a variety of host environments. Therefore the core 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.

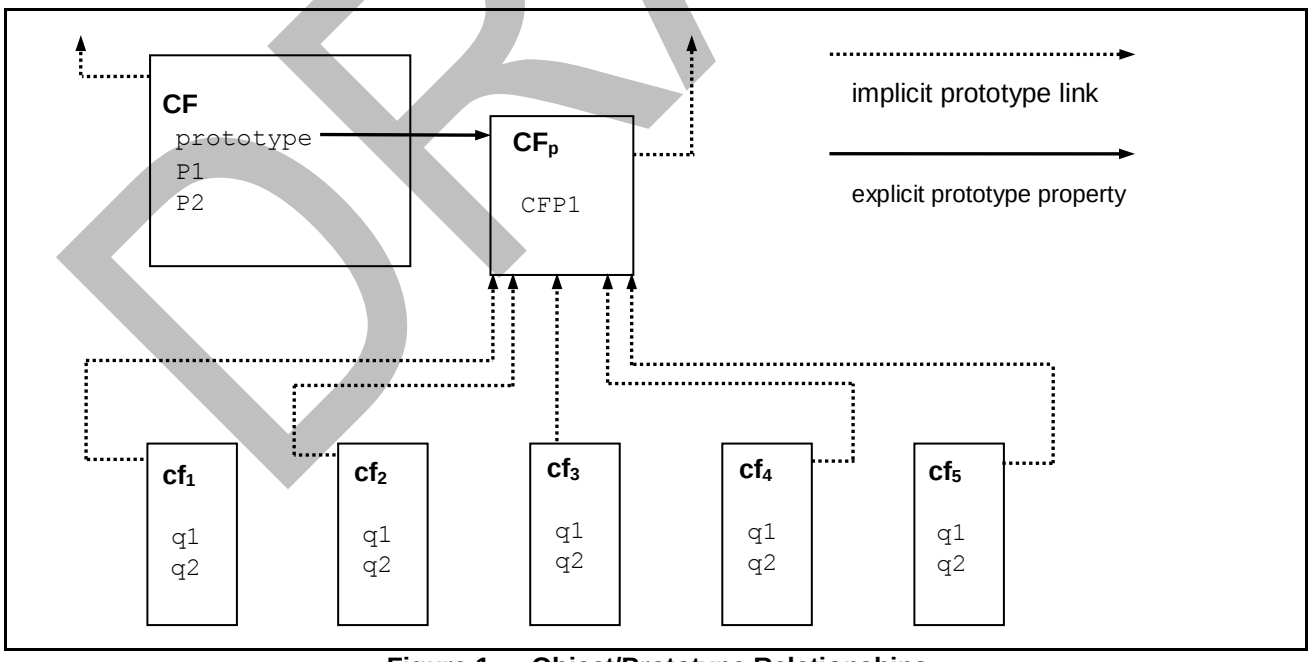

Figure 1 — Object/Prototype Relationships

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, while 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_1**, **cf_2**, **cf_3**, **cf_4**, and **cf_5**. Each of these objects contains properties named `q1` and `q2`. The dashed lines represent the implicit prototype relationship; so, for example, **cf_3**'s prototype is **CF_p**. The constructor, **CF**, has two properties itself, named `P1` and `P2`, which are not visible to **CF_p**, **cf_1**, **cf_2**, **cf_3**, **cf_4**, or **cf_5**. The property named `CFP1` in **CF_p** is shared by **cf_1**, **cf_2**, **cf_3**, **cf_4**, and **cf_5** (but not by **CF**), as are any properties found in **CF_p**'s implicit prototype chain that are not named `q1`, `q2`, or `CFP1`. Notice that there is no implicit prototype link between **CF** and **CF_p**.

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_1**, **cf_2**, **cf_3**, **cf_4**, and **cf_5** by assigning a new value to the property in **CF_p**.

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
ordinary object
object that has the default behaviour for the internal methods that must be supported by all ECMAScript objects.

4.3.7
exotic object
object that has some alternative behaviour for one or more of the internal methods that must be supported by all ECMAScript objects.

NOTE Any object that is not an ordinary object is an exotic object.

4.3.8
standard object
object whose semantics are defined by this specification.

4.3.9
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. A *built-in constructor* is a built-in object that is also a constructor.

4.3.10
undefined value
primitive value used when a variable has not been assigned a value

4.3.11
Undefined type
type whose sole value is the undefined value

4.3.12
null value
primitive value that represents the intentional absence of any object value

4.3.13
Null type
type whose sole value is the null value

4.3.14
Boolean value
member of the Boolean type

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

4.3.15
Boolean type
type consisting of the primitive values **true** and **false**

4.3.16
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 data property whose value is the Boolean value. A Boolean object can be coerced to a Boolean value.

4.3.17
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.

4.3.18
String type
set of all possible String values

4.3.19
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 data 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).

4.3.20
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.

4.3.21
Number type
set of all possible Number values including the special “Not-a-Number” (NaN) value, positive infinity, and negative infinity

4.3.22
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 data 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).

4.3.23
Infinity
number value that is the positive infinite Number value

4.3.24
NaN
number value that is a IEEE 754 “Not-a-Number” value

4.3.25
function
member of the Object type 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.

4.3.26
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.

4.3.27
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.

4.3.28
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.

4.3.29
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.

4.3.30
attribute
internal value that defines some characteristic of a property

4.3.31
own property
property that is directly contained by its object

4.3.32
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.

A *chain production* is a production that has exactly one nonterminal symbol on its right-hand side along with zero or more terminal symbols.

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 *Script*, that describe how sequences of tokens can form syntactically correct independent components of an ECMAScript programs.

When a stream of characters is to be parsed as an ECMAScript script, 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 script is syntactically in error if the tokens in the stream of input elements cannot be parsed as a single instance of the goal nonterminal *Script*, 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 scripts. 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 generalize productions that permit token sequences that are not valid ECMAScript scriptss. For example, this technique is used in with object literals and object destructuring patterns. In such cases a more restrictive *supplemental grammar* is provided that further restricts the acceptable token sequences. In certain contexts, when explicitly specific, the input elements corresponding to such a production are parsed again using a goal symbol of a supplemental grammar. The script 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 script 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 "$_{opt}$", 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$_{opt}$

is a convenient abbreviation for:

VariableDeclaration **:**
Identifier
Identifier Initialiser

and that:

IterationStatement **:**
for (***ExpressionNoIn*$_{opt}$ **; *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*

is a convenient abbreviation for:

IterationStatement **:**
for (; *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*
for (*ExpressionNoIn* **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*

which in turn is an abbreviation for:

IterationStatement **:**
for (; ; *Expression*$_{opt}$ **)** *Statement*
for (; *Expression* **;** *Expression*$_{opt}$ **)** *Statement*
for (*ExpressionNoIn* **; ;** *Expression*$_{opt}$ **)** *Statement*
for (*ExpressionNoIn* **;** *Expression* **;** *Expression*$_{opt}$ **)** *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 ∉ *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 ∉ {**1**, **3**, **5**, **7**, **9**}] *DecimalDigits*
DecimalDigit [lookahead ∉ *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 script 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 script.

The lexical grammar has multiple goal symbols and the appropriate goal symbol to use depends upon the syntactic grammar context. If a phrase of the form "[Lexical goal *LexicalGoalSymbol*]" appears on the right-hand-side of a syntactic production then the next token must be lexically recognized using the indicated goal symbol. In the absence of such a phrase the default lexical goal symbol is used.

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 character

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.

Algorithms may be explicitly parameterized, in which case the names and usage of the parameters must be provided as part of the algorithm's definition. 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 script souce code.

Unless explicitly specified otherwise, all chain productions have an implicit associated definition for every algorithm that is might be applied to that production's left-hand side nonterminal. The implicit simply reapplies the same algorithm name with the same parameters, if any, to the chain production's sole right-hand side nonterminal and then result. For example, assume there is a production

Block **:**
{ *StatementList* **}**

but there is no evalution algorithm that is explicitly specified for that production. If in some algorithm there is a statement of the form: "Return the result of evaluating *Block*" it is implicit that the algorithm has an evalution algorithm of the form:

Runtime Semantics: Evaluation

Block **:** **{** *StatementList* **}**

1. Return the result of evaluating *StatementList*

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 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 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 abs(k) < abs(y) and $x-k = q \times y$ for some integer q.

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

NOTE floor(x) = x–(x 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 script 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. 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 named 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, then
        i. Let *contained* be the result of Contains for *sym* with argument *symbol*.
        ii. 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 comforming implementation must, prior to the first evaluation of a *Script*, validate all of the early error rules of the productions used to parse that *Script*. If any of the early error rules are violated the *Script* is invalid and can not be evaluated.

6 Source Text

Syntax

SourceCharacter **::**
any Unicode character

The ECMAScript code is expressed using Unicode, version 5.1 or later. ECMAScript source text is a sequence of Unicode characters. The phrase “Unicode character” refers to the abstract linguistic or typographical unit represented by a single Unicode scalar value. The actual encodings used to store and interchange ECMAScript source text is not relevant to this specification. Any well-defined encoding such as UTF-32 or UTF-16 may be used. Source text might even be externally represented using a non-Unicode character encoding. Regardless of the external source text encoding, a conforming ECMAScript implementation processes the source text as if it was an equivalent sequence of *SourceCharacter* values. Each *SourceCharacter* being an abstract Unicode character with a corresponding Unicode scalar value. Conforming ECMAScript implementations are not required to perform any normalisation of text, or behave as though they were performing normalisation of text.

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,template literals and identifiers, any Unicode characters may also be expressed as a Unicode escape sequence that explicitly express a code point’s numeric value. Within a comment, such an escape sequence is effectively ignored as part of the comment. Within other contexts, such an escape sequence contextually contributes one Unicode character.

NOTE 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 Unicode 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 Unicode character to the literal and is never interpreted as a line terminator or as a quote mark that might terminate the string literal.

ECMAScript String values (8.4) are computational sequences of 16-bit integer values called “code units”. ECMAScript language constructs that generate string values from *SourceCharacter* sequences use UTF-16 encoding to generate the code unit values.

Static Semantics: UTF-16 Encoding

The UTF-16 Encoding of a numeric code point value, *cp*, is deterimined as follows:

1. Assert: $0 \leq cp \leq 0x10FFFF$
2. If $cp \leq 65535$, then return *cp*.
3. Let *cu1* be floor((*cp* – 65536) / 1024) + 55296. NOTE 55296 is 0xD800.
4. Let *cu2* be ((*cp* – 65536) modulo 1024) + 56320. NOTE 56320 is 0xDC00.
5. Return the code unit sequence consisting of *cu1* followed by *cu2*.

7 Lexical Conventions

The source text of an ECMAScript script 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 several situations where the identification of lexical input elements is sensitive to the syntactic grammar context that is consuming the input elements. This requires multiple goal symbols for the lexical grammar. The *InputElementDiv* goal symbol is the default goal symbol and is used in those syntactic grammar contexts where a leading division (**/**) or division-assignment (**/=**) operator is permitted. The *InputElementRegExp* goal symbol is used in all syntactic grammar contexts where a *RegularExpressionLiteral* is permitted. The *InputElementTemplateTail* goal is used in syntactic grammar contexts where a *TemplateLiteral* logically continues after a substitution element.

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
RightBracePunctuator

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

InputElementTemplateTail **::**
WhiteSpace
LineTerminator
Comment
Token
DivPunctuator
TemplateSubstitutionTail

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, template 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 Point	*Name*	*Formal Name*	*Usage*
U+200C	Zero width non-joiner	<ZWNJ>	*IdentifierPart*
U+200D	Zero width joiner	<ZWJ>	*IdentifierPart*
U+FEFF	Byte Order Mark	<BOM>	*Whitespace*

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 occur within a *StringLiteral*, a *RegularExpressionLiteral*, a *Template*, or a *TemplateSubstitutionTail* where they are considered significant characters forming part of a literal value. They may also occur 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 Point	*Name*	*Formal Name*
U+0009	Tab	<TAB>
U+000B	Vertical Tab	<VT>
U+000C	Form Feed	<FF>
U+0020	Space	<SP>
U+00A0	No-break space	<NBSP>
U+FEFF	Byte Order Mark	<BOM>
Other category “Zs”	Any other Unicode “space separator”	<USP>

ECMAScript implementations must recognise all of the white space characters defined in Unicode 5.1. 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*, *Template*, or *TemplateSubstitutionTail*. 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

Code Point	*Name*	*Formal Name*
U+000A	Line Feed	<LF>
U+000D	Carriage Return	<CR>
U+2028	Line separator	<LS>
U+2029	Paragraph separator	<PS>

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

Syntax

LineTerminator **::**
<LF>
<CR>
<LS>
<PS>

LineTerminatorSequence **::**
<LF>
<CR> [lookahead ∉ <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 Unicode 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 **::**
/* *MultiLineCommentChars*$_{opt}$ ***/**

MultiLineCommentChars **::**
MultiLineNotAsteriskChar MultiLineCommentChars$_{opt}$
***** *PostAsteriskCommentChars*$_{opt}$

PostAsteriskCommentChars **::**
MultiLineNotForwardSlashOrAsteriskChar MultiLineCommentChars$_{opt}$
***** *PostAsteriskCommentChars*$_{opt}$

MultiLineNotAsteriskChar **::**
SourceCharacter **but not ***

MultiLineNotForwardSlashOrAsteriskChar **::**
SourceCharacter **but not one of / or ***

SingleLineComment **::**
// *SingleLineCommentChars*$_{opt}$

SingleLineCommentChars **::**
SingleLineCommentChar SingleLineCommentChars$_{opt}$

SingleLineCommentChar **::**
SourceCharacter **but not** *LineTerminator*

7.5 Tokens

Syntax

Token **::**
IdentifierName
Punctuator
NumericLiteral
StringLiteral
Template

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

7.6 Identifier Names and Identifiers

IdentifierName, *Identifier*, and *ReservedWord* are tokens that are interpreted according to the Default Identifier Syntax given in Unicode Standard Annex #31, Identifier and Pattern Syntax, with some small modifications. *ReservedWord* is is an enumerated subset of *IdentifierName* and *Identifier* is an *IdentifierName* that is not a *ReservedWord* (see 7.6.1). The Unicode identifier grammar is based on character properties specified by the Unicode Standard. The Unicode characters in the specified categories in version 5.1.0 of the Unicode standard must be treated as in those categories by all conforming ECMAScript implementations. ECMAScript implementations may recognise identifier characters defined in later editions of the Unicode Standard.

NOTE 1 This standard specifies specific character additions: The dollar sign (**`U+0024`**) and the underscore (**`U+005f`**) are permitted anywhere in an *IdentifierName*, and the characters zero width non-joiner (U+200C) and zero width joiner (U+200D) are permitted anywhere after the first character of an *IdentifierName*.

Unicode escape sequences are permitted in an *IdentifierName*, where they contribute a single Unicode character to the *IdentifierName*. The code point of the contributed character is expressed by the *HexDigits* of the *UnicodeEscapeSequence* (see 7.8.4). The **** preceding the *UnicodeEscapeSequence* and the **`u`** and **`{ }`** characters, if they appear, do not contribute characters 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 the Unicode character it constributes, 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).

NOTE 2 If maximal portability is a concern, programmers should only employ the identifier characters that were defined in Unicode 3.0.

Syntax

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

IdentifierName **::**
IdentifierStart
IdentifierName IdentifierPart

IdentifierStart **::**
UnicodeIDStart
$
_
**** *UnicodeEscapeSequence*

IdentifierPart **::**
UnicodeIDContinue
$
_
**** *UnicodeEscapeSequence*
<ZWNJ>
<ZWJ>

UnicodeIDStart **::**
any Unicode character with the Unicode property “ID_Start”.

UnicodeIDContinue **::**
any Unicode character with the Unicode property “ID_Continue”

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

Static Semantics: StringValue

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

1. Return the StringValue of *IdentifierName*.

IdentifierName **::**
IdentifierStart
IdentifierName IdentifierPart

1. Return the String value consisting of the sequence of code units corresponding to *IdentifierName*. In determining the sequence any occurrences of **** *UnicodeEscapeSequence* are first replaced with the code point represented by the *UnicodeEscapeSequence* and and then the code points of the entire *IdentifierName* are converted to code units by UTF-16 Encoding (clause 6) each code point.

7.6.1 Reserved Words

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

Syntax

ReservedWord **::**
Keyword
FutureReservedWord
NullLiteral
BooleanLiteral

The *ReservedWord* definitions are specified as literal sequences of Unicode characters. However, any Unicode character in a *ReservedWord* can also be expressed by a **** *UnicodeEscapeSequence* that expresses that same Unicode character's code point. Use of such escape sequences does not change the meaning of the *ReservedWord*.

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	**delete**	**import**	**this**
case	**do**	**in**	**throw**
catch	**else**	**instanceof**	**try**
class	**export**	**let**	**typeof**
continue	**finally**	**new**	**var**
const	**for**	**return**	**void**
debugger	**function**	**super**	**while**
default	**if**	**switch**	**with**

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**

```
enum            extends
```

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:

```
implements                  private     public      yield
interface     package       protected   static
```

7.7 Punctuators

Syntax

Punctuator **:: one of**

```
{       (       )       [       ]
.       ;       ,       <       >       <=
>=      ==      !=      ===     !==
+       -       *       %       ++      --
<<      >>      >>>     &       |       ^
!       ~       &&      ||      ?       :
=       +=      -=      *=      %=      <<=
>>=     >>>=    &=      |=      ^=      =>
```

DivPunctuator **:: one of**

```
/       /=
```

RightBracePunctuator **::**

```
}
```

7.8 Literals

7.8.1 Null Literals

Syntax

NullLiteral **::**

```
null
```

7.8.2 Boolean Literals

Syntax

BooleanLiteral **::**

```
true
false
```

7.8.3 Numeric Literals

Syntax

NumericLiteral **::**
DecimalLiteral
BinaryIntegerLiteral
OctalIntegerLiteral
HexIntegerLiteral

DecimalLiteral **::**
DecimalIntegerLiteral **.** $DecimalDigits_{opt}$ $ExponentPart_{opt}$
. *DecimalDigits* $ExponentPart_{opt}$
DecimalIntegerLiteral $ExponentPart_{opt}$

DecimalIntegerLiteral **::**
0
NonZeroDigit $DecimalDigits_{opt}$

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

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

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

HexIntegerLiteral **::**
0x *HexDigits*
0X *HexDigits*

HexDigits **::**
HexDigit
HexDigits 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 *SourceCharacter* 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**.

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.

Static Semantics: **MV's**

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* **::** *BinaryIntegerLiteral* is the MV of *BinaryIntegerLiteral*.
- The MV of *NumericLiteral* **::** *OctalIntegerLiteral* is the MV of *OctalIntegerLiteral*.
- 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** or of *OctalDigit* **::** **0** or of *BinaryDigit* **::** **0** is 0.
- The MV of *DecimalDigit* **::** **1** or of *NonZeroDigit* **::** **1** or of *HexDigit* **::** **1** or of *OctalDigit* **::** **1** or of *BinaryDigit* **::** **1** is 1.

- The MV of *DecimalDigit* **::** `2` or of *NonZeroDigit* **::** `2` or of *HexDigit* **::** `2` or of *OctalDigit* **::** `2` is 2.
- The MV of *DecimalDigit* **::** `3` or of *NonZeroDigit* **::** `3` or of *HexDigit* **::** `3` or of *OctalDigit* **::** `3` is 3.
- The MV of *DecimalDigit* **::** `4` or of *NonZeroDigit* **::** `4` or of *HexDigit* **::** `4` or of *OctalDigit* **::** `4` is 4.
- The MV of *DecimalDigit* **::** `5` or of *NonZeroDigit* **::** `5` or of *HexDigit* **::** `5` or of *OctalDigit* **::** `5` is 5.
- The MV of *DecimalDigit* **::** `6` or of *NonZeroDigit* **::** `6` or of *HexDigit* **::** `6` or of *OctalDigit* **::** `6` is 6.
- The MV of *DecimalDigit* **::** `7` or of *NonZeroDigit* **::** `7` or of *HexDigit* **::** `7` or of *OctalDigit* **::** `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 *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 *BinaryIntegerLiteral* **::** `0b` *BinaryDigit* is the MV of *BinaryDigit*.
- The MV of *BinaryIntegerLiteral* **::** `0B` *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 *OctalIntegerLiteral* **::** `0o` *OctalDigit* is the MV of *OctalDigit*.
- The MV of *OctalIntegerLiteral* **::** `0O` *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 *HexIntegerLiteral* **::** `0x` *HexDigits* is the MV of *HexDigits*.
- The MV of *HexIntegerLiteral* **::** `0X` *HexDigits* is the MV of *HexDigits*.
- The MV of *HexDigits* **::** *HexDigit* is the MV of *HexDigit*.
- The MV of *HexDigits* **::** *HexDigits HexDigit* is (the MV of *HexDigits* 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.

7.8.4 String Literals

NOTE A string literal is zero or more Unicode code points enclosed in single or double quotes. Unicode code points may also be 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. String literals evaluate to ECAMScript String values. When generating these string values Unicode code points are UTF-16 encoded as defined in clause 6. Code points belonging to Basic Multilingual Plane are encoded as a single code unit element of the string. All other code points are encoded as two code unit elements of the string.

Syntax

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

DoubleStringCharacters **::**
DoubleStringCharacter DoubleStringCharacters$_{opt}$

SingleStringCharacters **::**
SingleStringCharacter *SingleStringCharacters*$_{opt}$

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 ∉ *DecimalDigit*]
HexEscapeSequence
UnicodeEscapeSequence

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

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*
u{ *HexDigits* **}**

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

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**.

Static Semantics

Static Semantics: Early Errors

UnicodeEscapeSequence **::** **u{** *HexDigits* **}**

- It is a Syntax Error if the MV of *HexDigits* > 1114111.

Static Semantics: **SV's and CV's**

A string literal stands for a value of the String type. The String value (SV) of the literal is described in terms of code unit values (CV) contributed by the various parts of the string literal. As part of this process, some Unicode 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 code unit sequence.
- The SV of *StringLiteral* **::** `''` is the empty code unit 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 or two code units that is the CV of *DoubleStringCharacter*.
- The SV of *DoubleStringCharacters* **::** *DoubleStringCharacter DoubleStringCharacters* is a sequence of one or two code units that is the CV of *DoubleStringCharacter* followed by all the code units in the SV of *DoubleStringCharacters* in order.
- The SV of *SingleStringCharacters* **::** *SingleStringCharacter* is a sequence of one or two code units that is the CV of *SingleStringCharacter*.
- The SV of *SingleStringCharacters* **::** *SingleStringCharacter SingleStringCharacters* is a sequence of one or two code units that is the CV of *SingleStringCharacter* followed by all the code units in the SV of *SingleStringCharacters* in order.
- The SV of *LineContinuation* **::** `\` *LineTerminatorSequence* is the empty code unit sequence.
- The CV of *DoubleStringCharacter* **::** *SourceCharacter* **but not one of** `"` **or** `\` **or** *LineTerminator* is the UTF-16 Encoding (clause 6) of the code point value of *SourceCharacter*.
- 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 UTF-16 Encoding (clause 6) of the code point value of *SourceCharacter* .
- 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 the code unit value 0.
- 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`	`0x0008`	backspace	<BS>
`\t`	`0x0009`	horizontal tab	<HT>
`\n`	`0x000A`	line feed (new line)	<LF>
`\v`	`0x000B`	vertical tab	<VT>
`\f`	`0x000C`	form feed	<FF>
`\r`	`0x000D`	carriage return	<CR>
`\"`	`0x0022`	double quote	"
`\'`	`0x0027`	single quote	'
`\\`	`0x005C`	backslash	\

- 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 UTF-16 Encoding (clause 6) of the code point value of *SourceCharacter* .
- The CV of *HexEscapeSequence* **::** **x** *HexDigit HexDigit* is the code unit value that 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 code unit value that 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*.
- The CV of *UnicodeEscapeSequence* **::** **u{** *HexDigits* **}** the is the UTF-16 Encoding (clause 6) of the MV of *HexDigits*.

7.8.5 Regular Expression Literals

NOTE 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 source code comprising the *RegularExpressionBody* and the *RegularExpressionFlags* are subsequently parsed using the more stringent ECMAScript Regular Expression grammar (15.10.1).

An implementation may extend the ECMAScript Regular Expression grammar defined in 15.10.1, but it must not extend the *RegularExpressionBody* and *RegularExpressionFlags* productions defined below 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: **/(?:)/**.

Static Semantics: BodyText

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

1. Return the source code that was recognized as *RegularExpressionBody*.

Static Semantics: FlagText

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

1. Return the source code that was recognized as *RegularExpressionFlags*.

7.8.6 Template Literal Lexical Components

Syntax

Template **::**
NoSubstitutionTemplate
TemplateHead

NoSubstitutionTemplate **::**
`** *TemplateCharacters*$_{opt}$ **`

TemplateHead **::**
` *TemplateCharacters*$_{opt}$ **${**

TemplateSubstitutionTail **::**
TemplateMiddle
TemplateTail

TemplateMiddle **::**
} *TemplateCharacters*$_{opt}$ **${**

TemplateTail **::**
} *TemplateCharacters*$_{opt}$ **`**

TemplateCharacters **::**
TemplateCharacter TemplateCharacters$_{opt}$

TemplateCharacter **::**
SourceCharacter **but not one of ` or \ or $**
$ [lookahead ∉ **{**]
**** *EscapeSequence*
LineContinuation

Static Semantics: TV**'s** and TRV**'s**

A template literal component is interpreted as a sequence of Unicode characters. The Template Value (TV) of a literal component is described in terms of code unit values (CV, 7.8.4) contributed by the various parts of the

template literal component. As part of this process, some Unicode characters within the template component are interpreted as having a mathematical value (MV, 7.8.3). In determining a TV, escape sequences are replaced by the code unit of the Unicode characters represented by the escape sequence. The Template Raw Value (TRV) is similar to a Template Value with the difference that in TRVs escape sequences are interpreted literally.

- The TV and TRV of *NoSubstitutionTemplate* :: ``` is the empty code unit sequence.
- The TV and TRV of *TemplateHead* :: `` `${ `` is the empty code unit sequence.
- The TV and TRV of *TemplateMiddle* :: `}${` is the empty code unit sequence.
- The TV and TRV of *TemplateTail* :: `}` is the empty code unit sequence.
- The TV of *NoSubstitutionTemplate* :: ` *TemplateCharacters* ` is the TV of *TemplateCharacters*.
- The TV of *TemplateHead* :: ` *TemplateCharacters* `${` is the TV of *TemplateCharacters*.
- The TV of *TemplateMiddle* :: `}` *TemplateCharacters* `${` is the TV of *TemplateCharacters*.
- The TV of *TemplateTail* :: `}` *TemplateCharacters* ` is the TV of *TemplateCharacters*.
- The TV of *TemplateCharacters*:: *TemplateCharacter* is the TV of *TemplateCharacter*.
- The TV of *TemplateCharacters* :: *TemplateCharacter TemplateCharacters* is a sequence consisting of the code units in the TV of *TemplateCharacter* followed by all the code units in the TV of *TemplateCharacters* in order.
- The TV of *TemplateCharacter* :: *SourceCharacter* **but not one of** ` **or** \ **or** $ is the UTF-16 Encoding (clause 6) of the code point value of *SourceCharacter*.
- The TV of *TemplateCharacter* :: $ [lookahead ∉ {] is the code unit value 0x0024.
- The TV of *TemplateCharacter* :: \ *EscapeSequence* is the CV of *EscapeSequence*.
- The TV of *TemplateCharacter* :: *LineContinuation* is the TV of *LineContinuation*.
- The TV of *LineContinuation* :: \ *LineTerminatorSequence* is the empty code unit sequence.
- The TRV of *NoSubstitutionTemplate* :: ` *TemplateCharacters* ` is the TRV of *TemplateCharacters*.
- The TRV of *TemplateHead* :: ` *TemplateCharacters* `${` is the TRV of *TemplateCharacters*.
- The TRV of *TemplateMiddle* :: `}` *TemplateCharacters* `${` is the TRV of *TemplateCharacters*.
- The TRV of *TemplateTail* :: `}` *TemplateCharacters* ` is the TRV of *TemplateCharacters*.
- The TRV of *TemplateCharacters*:: *TemplateCharacter* is the TRV of *TemplateCharacter*.
- The TRV of *TemplateCharacters*:: *TemplateCharacter TemplateCharacters* is a sequence consisting of the code units in the TRV of *TemplateCharacter* followed by all the code units in the TRV of *TemplateCharacters*, in order.
- The TRV of *TemplateCharacter* :: *SourceCharacter* **but not one of** ` **or** \ **or** $ is the UTF-16 Encoding (clause 6) of the code point value of *SourceCharacter*.
- The TRV of *TemplateCharacter* :: $ [lookahead ∉ {] is the code unit value 0x0024.
- The TRV of *TemplateCharacter* :: \ *EscapeSequence* is the sequence consisting of the code unit value 0x005C followed by the code units of TRV of *EscapeSequence*.
- The TRV of *TemplateCharacter*:: *LineContinuation* is the TRV of *LineContinuation*.
- The TRV of *EscapeSequence* :: *CharacterEscapeSequence* is the TRV of the *CharacterEscapeSequence*.
- The TRV of *EscapeSequence* :: 0 [lookahead ∉ *DecimalDigit*] is the code unit value 0x0030.
- The TRV of *EscapeSequence* :: *HexEscapeSequence* is the TRV of the *HexEscapeSequence*.
- The TRV of *EscapeSequence* :: *UnicodeEscapeSequence* is the TRV of the *UnicodeEscapeSequence*.
- The TRV of *CharacterEscapeSequence* :: *SingleEscapeCharacter* is the TRV of the *SingleEscapeCharacter*.
- The TRV of *CharacterEscapeSequence* :: *NonEscapeCharacter* is the CV of the *NonEscapeCharacter*.
- The TRV of *SingleEscapeCharacter* :: **one of** ' " \ b f n r t v is the CV of the *SourceCharacter* that is that single character.
- The TRV of *HexEscapeSequence* :: x *HexDigit HexDigit* is the sequence consisting of code unit value 0x0078 followed by TRV of the first *HexDigit* followed by the TRV of the second *HexDigit*.
- The TRV of *UnicodeEscapeSequence* :: u *HexDigit HexDigit HexDigit HexDigit* is the sequence consisting of code unit value 0x0075 followed by TRV of the first *HexDigit* followed by the TRV of the second *HexDigit* followed by TRV of the third *HexDigit* followed by the TRV of the fourth *HexDigit*.
- The TRV of *UnicodeEscapeSequence* :: u{ *HexDigits* } is the sequence consisting of code unit value 0x0075 followed by code unit value 0x007B followed by TRV of *HexDigits* followed by code unit value 0x007D.

- The TRV of *HexDigits* **::** *HexDigit* is the TRV of *HexDigit*.
- The TRV of *HexDigits* **::** *HexDigits HexDigit* is the sequence consisting of TRV of *HexDigits* followed by TRV of *HexDigit*.
- The TRV of a *HexDigit* is the CV of the *SourceCharacter* that is that *HexDigit*.
- The TRV of *LineContinuation* **::** **** *LineTerminatorSequence* is the sequence consisting of the code unit value 0x005C followed by the code units of TRV of *LineTerminatorSequence*.
- The TRV of *LineTerminatorSequence* **::** <LF> is the code unit value 0x000A.
- The TRV of *LineTerminatorSequence* **::** <CR> [lookahead ∉ <LF>] is the code unit value 0x000D.
- The TRV of *LineTerminatorSequence* **::** <LS> is the code unit value 0x2028.
- The TRV of *LineTerminatorSequence* **::** <PS> is the code unit value 0x2029.
- The TRV of *LineTerminatorSequence* **::** <CR><LF> is the sequence consisting of the code unit value 0x000D followed by the code unit value 0x000A.

NOTE TV excludes the code units of *LineContinuation* while TRV includes them.

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 script 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 script 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 *script*, then a semicolon is automatically inserted at the end of the input stream.

3. When, as the script 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.

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 ECMAScript Language 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. An ECMAScript language value is a value that is characterized by an ECMAScript language type.

8.1.1 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**.

8.1.2 The Null Type

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

8.1.3 The Boolean Type

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

8.1.4 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 UTF-16 code unit value. 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 index 0, the next element (if any) at index 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.

Where ECMAScript operations interpret String values, each element is interpreted as a single UTF-16 code unit. However, ECMAScript does not place any restrictions or requirements on the sequence of code units in a String value, so they may be ill-formed when interpreted as UTF-16 code unit sequences. Operations that do not interpret String contents treat them as sequences of undifferentiated 16-bit unsigned integers. No operations ensure that Strings are in a normalized form. Only operations that are explicitly specified to be language or locale sensitive produce language-sensitive results

NOTE The rationale behind this design was to keep the implementation of Strings as simple and high-performing as possible. If ECMAScript source code is in Normalised Form C, string literals are guaranteed to also be normalised, as long as they do not contain any Unicode escape sequences.

Some operations interpret String contents as UTF-16 encoded Unicode code points. In that case the interpretation is:

- A code unit in the range 0 to 0xD7FF or in the range 0xE000 to 0xFFFF is interpreted as a code point with the same value.
- A sequence of two code units, where the first code unit *c1* is in the range 0xD800 to 0xDBFF and the second code unit *c2* is in the range 0xDC00 to 0xDFFF, is a surrogate pair and is interpreted as a code point with the value (*c1* - 0xD800) × 0x400 + (*c2* – 0xDC00) + 0x10000.
- A code unit that is in the range 0xD800 to 0xDFFF, but is not part of a surrogate pair, is interpreted as a code point with the same value.

8.1.5 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 +∞ and –∞, 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 π) 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.

8.1.6 The Object Type

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

- A *data property* associates a key value with an ECMAScript language value and a set of Boolean attributes.
- A *accessor property* associates a key value 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.

Property are identified using key values. A key value is either an ECMAScript String value or an ECMAScript Symbol object (??.??.??).

Property keys are used to access properties and their values. There are two kinds of access for properties: *get* and *set*, corresponding to value retrieval and assignment, respectively. The properties accessable via get and set access includes both *own properties* that are a direct part of an object and *inherited properties* which are provided by another associated object via a property inheritance relationship. Inherited properties may be either own or inherited properites of the associated object.

All objects are logically collections of properties, but there are multiple forms of objacts that differ in their semanjtics for accessing and manipulating their properties. O*rdinary object* are the most common form of objects and have the default object semantics, An *exotic object* is any form of object whose property semantics differ in any way from the default semantics.

8.1.6.1 Property Attributes

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

Table 5 — Attributes of a Data Property

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

An accessor property associates a key value with the attributes listed in Table 6.

Table 6 — Attributes of a Named Accessor Property

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

If the initial values of a property's attributes are not explicitly specified by this specification, the default value defined in Table 7 is used.

Table 7 — Default Attribute Values

Attribute Name	Default Value
[[Value]]	**undefined**
[[Get]]	**undefined**
[[Set]]	**undefined**
[[Writable]]	**false**
[[Enumerable]]	**false**
[[Configurable]]	**false**

8.1.6.2 Object Internal Methods and Internal Data Properties

The actual semantics of ECMAScript objects are specified via algorithms called *internal methods*. Each object in an ECMAScript engine is associated with a set of internal methods that defines its runtime behaviour. These internal methods are not part of the ECMAScript language. They are defined by this specification purely for expository purposes. However, each object within an implementation of ECMAScript must behave as specified by the internal methods associated with it. The exact manner in which this is accomplished is determined by the implementation.

Internal methods are identified within this specification using names enclosed in double square brackets [[ ]]. Internal method names are polymorphic. This means that different ECMAScript object values may perform different algorithms when a common internal method name is invoke upon them. If, at runtime, the implementation of an algorithm attempts to use an internal method of an object that the object does not support, a **TypeError** exception is thrown.

Internal data properties correspond to internal state that is associated with objects and used by various ECMAScript specification algorithms. Depending upon the specific internal data property such state may consist of values of any ECMAScript language type or of specific ECMA specification type values. Unless explicitly specified otherwise, internal data properties may be dynamically added to ECMAScript objects.

Table 8**Error! Reference source not found.** summarises the *essential internal methods* used by this specification that are applicable to all ECMAScript objects. Every object must have algorithms for all of the essential internal methods. However, all objects do not necessarily use the same algorithms for those methods.

The “Signature” column of Table 8 and other similar tables describes the invocation pattern for each internal method. The invocation pattern always includes a parenthesised list of descriptive parameter names. If a parameter name is the same as an ECMAScript type name then the name describes the required type of the parameter value. If an internal method explicitly returns a value, its parameter list is followed by the symbol “→” and the type name of the returned value. The type names used in signatures 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. An internal method implicitly returns a Completion Record as described in 8.8. In addition to its parameters, an internal method always has access to the object upon which it is invoked as a method.

Table 8 — Essential Internal Methods

Internal Method	*Signature*	*Description*
[[GetInheritance]]	()→Object or Null	Determine the object that provides inherited properties for this object. A **null** value indicates that there are no inherited properties. an object.
[[SetInheritance]]	(*Object* or Null)	Associate with an object another object that provides inherited properties. Passing **null** indicates that there are no inherited properties.
[[IsExtensible]]	()→Boolean	Determine whether it is permitted to add additional properties to an object.
[[preventExtensions]]	()	Control whether new properties may be added to an object.
[[HasOwnProperty]]	(*propertyKey*) → Boolean	Returns a Boolean value indicating whether the object already has an own property whose key is *propertyKey*.
[[GetOwnProperty]]	(*propertyKey*) → Undefined or Property Descriptor	Returns a Property Descriptor for the own property of this object whose key is *propertyKey*, or **undefined** if no such property exists.
[[GetP]]	(*propertyKey, Receiver*) → any	Retrive the value of an object's property using the *propertyKey* parameter. If any ECMAScript code must be executed to retrieve the property value, *Receiver* is used as the **this** value when evaluating the code.
[[SetP]]	(*propertyKey*,*value*, *Receiver*) → *Boolean*	Try to set the value of an object's property indentified by *propertyKey* to *value*. If any ECMAScript code must be executed to set the property value, *Receiver* is used as the **this** value when evaluating the code. Returns **true** indicating that the property value was set or **false** indicating that it could not be set.
[[Delete]]	(*propertyKey*) → Boolean	Removes the own property indentified by the *propertyKey* parameter from the object. Return **false** is the property was not deleted because its [[Configurable]] attribute is **false**. Otherwise return **true**.
[[DefineOwnProperty]]	(*propertyKey, PropertyDescriptor*) → Boolean	Creates or alters the named own property to have the state described by a Property Descriptor. Returns **true** indicating that the property was successfully created/updated or **false** indicating that the property could not be created or updated.
[[Enumerate]]	()→Object	Returns an iterator object that over the string values of the keys of the enumerable properties of the object.
[[Keys]]	()→List of String	Returns an Array containing all of the enumerable own property keys for the object that are Strings.
[[OwnPropertyKeys]]	()→List of (String or Symbol)	Returns an Array containing all of the own property keys for the object except those that are private Symbols.
[[Freeze]]	() → *Boolean*	
[[Seal]]	() → *Boolean*	
[[IsFrozen]]	() → *Boolean*	
[[IsSealed]]	() → *Boolean*	

Table 9 summarises additional essential internal methods that must be supported by all objects that may be called as functions..

Table 9 — Addional Essential Internal Method of Function Objects

Internal Method	*Value Type Domain*	*Description*
[[Call]]	(*any*, a List of *any*) → *any* or Reference	Executes code associated with the object. Invoked via a function call expression. The arguments to the internal method are a **this** value and a list containing the arguments passed to the function by a call expression. Objects that implement this internal method are *callable*. Only callable objects that are host objects may return Reference values.
[[Construct]]	(a List of *any*) → Object	Creates an object. Invoked via the **`new`** operator. The arguments to the internal are the arguments passed to the **new** operator. Objects that implement this internal method are called *constructors*.

8.1.6.2 Invariants of the Essential Internal Methods

Current this section is just a bunch of material merged together from the ES5 spec. and from the wiki Proxy pages. It need to be completely reworked.

The intent is that it lists all invariants of the Essential Internal Methods. This includes both invariants that are enforced for Proxy objects and other invariants that may not be enfored.

Definitions:

- The *target* of an internal method is the object the internal method is called upon.
- A *sealed property* is a non-configurable own property of a target.
- A *frozen property* is a non-configurable non-writable own property of a target.
- A new property is a property that does not exist on a non-extensible target.
- Two property descriptors *desc1* and *desc2* for a property key value are incompatible if:
  1. *Descl* is produced by calling [[GetOwnPropertyDescriptor]] of *target* with *key*, and
  2. Calling [[DefineOwnProperty]] of *target* with arguments *key* and *desc2* would throw a *TypeError* exception.

Exotic objects may define additional constraints upon their [[SetP]] internal method behavior. If possible, exotic objects should not allow [[SetP]] operations in situations where this definition of [[CanPut]] returns false.

[[GetInheritance]]

Every [[Prototype]] chain must have finite length (that is, starting from any object, recursively accessing the [[Prototype]] internal data property must eventually lead to a **null** value).

getOwnPropertyDescriptor

Non-configurability invariant: cannot return incompatible descriptors for sealed properties
Non-extensibility invariant: must return undefined for new properties
Invariant checks:
if trap returns undefined, check if the property is configurable
if property exists on target, check if the returned descriptor is compatible
if returned descriptor is non-configurable, check if the property exists on the target and is also non-configurable

defineProperty

Non-configurability invariant: cannot succeed (return true) for incompatible changes to sealed properties
Non-extensibility invariant: must reject (return false) for new properties
Invariant checks:
on success, if property exists on target, check if existing descriptor is compatible with argument descriptor
on success, if argument descriptor is non-configurable, check if the property exists on the target and is also non-configurable

getOwnPropertyNames

Non-configurability invariant: must report all sealed properties
Non-extensibility invariant: must not list new property names
Invariant checks:
check whether all sealed target properties are present in the trap result
If the target is non-extensible, check that no new properties are listed in the trap result

deleteProperty

Non-configurability invariant: cannot succeed (return true) for sealed properties
Invariant checks:
on success, check if the target property is configurable

getPrototypeOf

Invariant check: check whether the target's prototype and the trap result are identical (according to the egal operator)

freeze | seal | preventExtensions

Invariant checks:
on success, check if isFrozen(target), isSealed(target) or !isExtensible(target)

isFrozen | isSealed | isExtensible

Invariant check: check whether the boolean trap result is equal to isFrozen(target), isSealed(target) or isExtensible(target)

hasOwn

Non-configurability invariant: cannot return false for sealed properties
Non-extensibility invariant: must return false for new properties
Invariant checks:
if false is returned, check if the target property is configurable
if false is returned, the property does not exist on target, and the target is non-extensible, throw a TypeError

has

Non-configurability invariant: cannot return false for sealed properties
Invariant checks:
if false is returned, check if the target property is configurable

get

Non-configurability invariant: cannot return inconsistent values for frozen data properties, and must return undefined for sealed accessors with an undefined getter

Invariant checks:

if property exists on target as a data property, check whether the target property's value and the trap result are identical (according to the egal operator)

if property exists on target as an accessor, and the accessor's get attribute is undefined, check whether the trap result is also undefined.

set

Non-configurability invariant: cannot succeed (return true) for frozen data properties or sealed accessors with an undefined setter

Invariant checks:

on success, if property exists on target as a data property, check whether the target property's value and the update value are identical (according to the egal operator)

on success, if property exists on target as an accessor, check whether the accessor's set attribute is not undefined

keys

Non-configurability invariant: must report all enumerable sealed properties

Non-extensibility invariant: must not list new property names

Invariant checks:

Check whether all enumerable sealed target properties are listed in the trap result

If the target is non-extensible, check that no new properties are listed in the trap result

enumerate

Non-configurability invariant: must report all enumerable sealed properties

Invariant checks:

Check whether all enumerable sealed target properties are listed in the trap result

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.

~~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 standard ECMAScript objects are ordinary objects and behave as described in 8.3. Some standard objects are exotic objects and have behaviour defined in 8.4..

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

The [[GetOwnProperty]] internal method of all objects must conform to the following invariants for each property of the 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 result of calling an object's [[IsExtensible]] internal method 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 all objects must not permit the addition of a new property to an object if the [[Extensible]] internal method of that object has been observed by ECMAScript code to be **false**.

If the result of calling the [[IsExtensible]] internal method of an object has been observed by ECMAScript code to be **false** then it must not subsequently become **true**.

8.2 ECMAScript Specification Types

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, Environment Record, and Data Block. 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.

8.2.1 Data Blocks

This section is a placeholder for describing the Data Block internal type. The following material is verbatium from the the Binary Data ES wiki proposal. The material has not yet been reviewed or integrated with the rest of this spec.

This spec introduces a new, spec-internal block datatype, intuitively representing a contiguously allocated block of binary data. Blocks are not ECMAScript language values and appear only in the program store (aka heap).

A block is one of:

- a number-block
- an array-block[t, n]
- a struct-block[t1, ..., tn]

A number-block is one of:

- an unsigned-integer; i.e., one of uint8, uint16, uint32, or uint64
- a signed-integer; i.e., one of int8, int16, int32, or int64
- a floating-point; i.e., one of float32 or float64

A uintk is an integer in the range [0, 2k). An intk is an integer in the range [-2k-1, 2k-1). A floatk is a floating-point number representable as a k-bit IEE754 value.

An array-block[t, n] is an ordered sequence of n blocks of homogeneous block type t. Each element of the array is stored at in independently addressable location in the program store, and multiple Data objects may contain references to the element.

A struct-block[t1, ..., tn] is an ordered sequence of n blocks of heterogeneous types t1 to tn, respectively. Each field of the struct is stored at in independently addressable location in the program store, and multiple Data objects may contain references to the field.

The spec also introduces a datatype of Data objects, which are ECMAScript objects that encapsulate references to block data in the program store. Every Data object has the following properties:

[[Class]] = “Data”

[[Value]] : reference[block] – a reference to a block in the program store

[[DataType]] : reference[Type] – a reference to a Type object describing this object’s data block

8.2.2 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 an ECMAScript value or 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 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 R is the record shown in the previous paragraph then R.[[field2]] is shorthand for "the field of R named [[field2]]".

Schema for commonly used Record field combinations 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**}.

8.2.3 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 whole fields are defined as by Table 10.

Table 10 — Completion Record Fields

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

The term "abrupt completion" refers to any completion with a [[type]] value other than **normal**.

8.2.3.1 NormalCompletion

The abstract operation NormalCompletion with a single *argument*, such as:

1. Return NormalCompletion(*argument*).

Is a short hand that is defined as follows:

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

8.2.3.2 Implicit Completion Values

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"`**.

means the same thing as:

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

A "return" statement without a value in an algoritm step means the same thing as:

1. Return NormalCompletion(*argument*).

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.

8.2.3.3 Throw an Exception

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}.

8.2.3.4 ReturnIfAbrupt

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]].

8.2.4 The Reference Specification Type

NOTE The Reference type is used to explain the behaviour of such operators as **`delete`**, **`typeof`**, the assignment operators, the **`super`** keyword and other lanauge features. For example, the left-hand operand of an assignment is expected to produce a reference.

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 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.

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

8.2.4.1 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 **true**, then
        i. Asset: In this case, *base* will never be **null** or **undefined**.
        ii. Set *base* to ToObject(*base*).
    b. Return the result of calling the [[GetP]] internal method of *base* passing GetReferencedName(*V*) and GetThisValue(*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.

NOTE The object that may be created in step 5.a.ii is not accessible outside of the above method. An implementation might choose to avoid the actual creation of the object.

8.2.4.2 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. Let *globalObj* be the result of the abstraction operation GetGlobalObject.
    c. Return the result of calling Put(*globalObj*,GetReferencedName(*V*), *W*, **false**).
6. Else if IsPropertyReference(*V*), then
    a. If HasPrimitiveBase(*V*) is **true**, then
        i. Asset: In this case, *base* will never be **null** or **undefined**.
        ii. Set *base* to ToObject(*base*).
    b. Let *succeeded* be the result of calling the [[SetP]] internal method of *base* passing GetReferencedName(*V*), *W*, and GetThisValue(*V*) as arguments).
    c. ReturnIfAbrupt(*succeeded*).
    d. If *succeeded* is **false** and IsStrictReference(*V*) is **true**, then throw a **TypeError** exception.
    e. Return.
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.

NOTE The object that may be created in step 6.a.ii is not accessible outside of the above algorithm. An implementation might choose to avoid the actual creation of that transient object.

8.2.4.3 GetThisValue (V)

1. ReturnIfAbrupt(*V*).
2. If Type(*V*) is not Reference, return *V*.
3. If IsUnresolvableReference(*V*), throw a **ReferenceError** exception.
4. If IsSuperReference(*V*), then
    a. Return the value of the *thisValue* component of the reference *V*.
5. Return GetBase(*V*).

8.2.5 The Property Descriptor 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 8.1.6.1. 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 8.1.6.1 Table 5 or Table 6.

A Property Descriptor may be derived from an ECMAScript object that has properties that directly correspond to the fields of a Property Descriptor. Such a derived Property Descriptor has an additional field named [[Origin]] whose value is the object from which the Property Descriptor was derived.

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

8.2.5.1 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**.

8.2.5.2 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**.

8.2.5.3 IsGenericDescriptor (Desc)

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

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

8.2.5.4 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. If *Desc* has an [[Origin]] field, then return *Desc*.[[Origin]].
3. Let *obj* be the result of the abstract operation ObjectCreate.
4. Assert: *obj* is an extensible ordinary object with no own properties.
5. If *Desc* has a [[Value]] field, then
    a. Call OrdinaryDefineOwnProperty with arguments *obj*, "**`value`**", and Property Descriptor {[[Value]]: *Desc*.[[Value]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}

6. If *Desc* has a [[Writable]] field, then
    a. Call OrdinaryDefineOwnProperty with arguments *obj*, "**`writable`**", and Property Descriptor {[[Value]]: *Desc*.[[Writable]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
7. If *Desc* has a [[Get]] field, then
    a. Call OrdinaryDefineOwnProperty with arguments *obj*, "**`get`**", and Property Descriptor {[[Value]]: *Desc*.[[Set]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
8. If *Desc* has a [[Get]] field, then
    a. Call OrdinaryDefineOwnProperty with arguments *obj*, "**`set`**", and Property Descriptor {[[Value]]: *Desc*.[[Set]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
9. If *Desc* has a [[Enumerable]] field, then
    a. Call OrdinaryDefineOwnProperty with arguments *obj*, "**`enumerable`**", and Property Descriptor {[[Value]]: *Desc*.[[Enumerable]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
10. If *Desc* has a [[Configurable]] field, then
    a. Call OrdinaryDefineOwnProperty with arguments *obj* , "**`configurable`**", and Property Descriptor {[[Value]]: *Desc*.[[Configurable]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
11. Return *obj*.

8.2.5.5 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 HasProperty(*Obj*, "**`enumerable`**") is **true**, then
    a. Let *enum* be the result of Get(*Obj*, "**`enumerable`**").
    b. ReturnIfAbrupt(*enum*).
    c. Set the [[Enumerable]] field of *desc* to ToBoolean(*enum*).
5. If the result of HasProperty(*Obj*, "**`configurable`**") is **true**, then
    a. Let *conf* be the result of Get(*Obj*, "**`configurable`**").
    b. ReturnIfAbrupt(*conf*).
    c. Set the [[Configurable]] field of *desc* to ToBoolean(*conf*).
6. If the result of HasProperty(*Obj*, "**`value`**") is **true**, then
    a. Let *value* be the result of Get(*Obj*, "**`value`**").
    b. ReturnIfAbrupt(*value*).
    c. Set the [[Value]] field of *desc* to *value*.
7. If the result of HasProperty(*Obj*, "**`writable`**") is **true**, then
    a. Let *writable* be the result of Get(*Obj*, "**`writable`**").
    b. ReturnIfAbrupt(*writable*).
    c. Set the [[Writable]] field of *desc* to ToBoolean(*writable*).
8. If the result of HasProperty(*Obj*, "**`get`**") is **true**, then
    a. Let *getter* be the result of Get(*Obj*, "**`get`**").
    b. ReturnIfAbrupt(*getter*).
    c. If IsCallable(*getter*) is **false** and *getter* is not **undefined**, then throw a **TypeError** exception.
    d. Set the [[Get]] field of *desc* to *getter*.
9. If the result of HasProperty(*Obj*, "**`set`**") is **true**, then
    a. Let *setter* be the result of Get(*Obj*, "**`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. Set the [[Origin]] field of *desc* to *Obj*.
12. Return *desc*.

8.2.5.6 CompletePropertyDescriptor (Desc, LikeDesc)

When the abstract operation CompletePropertyDescriptor is called with Property Descriptor Record *Desc*, the following steps are taken:

1. Assert: *LikeDesc* is either a Property Descriptor Record or **undefined**.
2. ReturnIfAbrupt(*Desc*).
3. Assert: *Desc* is a Property Descriptor Record
4. If *LikeDesc* is **undefined**, then set *LikeDesc* to Record{[[Value]]: **undefined**, [[Writable]]: **false**, [[Get]]: **undefined**, [[Set]]: **undefined**, [[Enumerable]]: **false**, [[Configurable]]: **false**).
5. If either IsGenericDescriptor(Desc) or IsDataDescriptor(*Desc*) is **true**, then
    a. If *Desc* does not have a [[Value]] field, then set *Desc*.[[Value]] to *LikeDesc*.[[Value]].
    b. If *Desc* does not have a [[Writable]] field, then set *Desc*.[[Writable]] to *LikeDesc*.[[Writable]].
6. Else,
    a. If *Desc* does not have a [[Get]] field, then set *Desc*.[[Get]] to *LikeDesc*.[[Get]].
    b. If *Desc* does not have a [[Set]] field, then set *Desc*.[[Set]] to *LikeDesc*.[[Set]].
7. If *Desc* does not have a [[Enumerable]] field, then set *Desc*.[[Enumerable]] to *LikeDesc*.[[Enumerable]].
8. If *Desc* does not have a [[Configurable]] field, then set *Desc*.[[Configurable]] to *LikeDesc*.[[Configurable]].
9. Return *desc*.

8.2.6 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.

8.3 Ordinary Object Internal Methods and Internal Data Properties

Sections 8.3-8.5 will eventually be subsectons of a new toplevel section that follow the current section 10

All ordinary objects have an internal data property called [[Prototype]]. The value of this property is either **null** or an object and is used for implementing inheritance. 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 set access. Accessor properties are inherited for both get access and set access.

Every ordinary ECMAScript object has a Boolean-valued [[Extensible]] internal data property that controls whether or not properties may be added to the object. If the value of the [[Extensible]] internal data 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 data properties of the object may not be modified. Once the value of an object's [[Extensible]] internal data property has been set to **false** it may not be subsequently changed to **true**.

In the following algorithm descriptions, assume *O* is an ordinary ECMAScript object, *P* is a property key value, *V* is any ECMAScript language value, Desc is a Property Description record, and *B* is a Boolean flag.

8.3.1 [[GetInheritance]] ( )

When the [[GetInheritance]] internal method of *O* is called the following steps are taken:

1. Return the value of the [[Prototype]] internal data property of *O*.

8.3.2 [[SetInheritance]] (V)

When the [[SetInheritance]] internal method of *O* is called with argument *V* the following steps are taken:

1. Assert: Either Type(*V*) is Object or Type(*V*) is Null.
2. Let *extensible* be the value of the [[Extensible]] internal data propertry of *O*.
3. If *extensible* is **false**, then return **false**.
4. Set the value of the [[Prototype]] internal data property of *O* to *V*.
5. Return **true**.

8.3.3 [[IsExtensible]] ( )

When the [[IsExtensible]] internal method of *O* is called the following steps are taken:

1. Return the value of the [[Extensible]] internal data property of *O*.

8.3.4 [[PreventExtensions]] ( )

When the [[PreventExtensions]] internal method of *O* is called the following steps are taken:

1. Assert: Type(*B*) is Boolean.
2. Set the value of the [[Extensible]] internal data property of *O* to **false**.
3. Return NormalCompletion(empty).

8.3.5 [[HasOwnProperty]] (P)

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

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. If *O* does not have an own property with key *P*, return **false**
3. Return **true**.

8.3.6 [[GetOwnProperty]] (P)

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

1. Return the result of OrdinaryGetOwnProperty with arguments *O* and *P*.

8.3.6.1 OrdinaryGetOwnProperty (O, P)

When the abstract operation OrdinaryGetOwnProperty is called with Object *O* and with property key *P*, the following steps are taken:

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. If *O* does not have an own property with key *P*, return **undefined**.
3. Let *D* be a newly created Property Descriptor with no fields.
4. Let *X* be *O*'s own property whose key is *P*.
5. 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
6. 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.
7. Set *D*.[[Enumerable]] to the value of *X*'s [[Enumerable]] attribute.
8. Set *D*.[[Configurable]] to the value of *X*'s [[Configurable]] attribute.
9. Return *D*.

8.3.7 [[GetP]] (P, Receiver)

When the [[GetP]] internal method of *O* is called with property key *P* and ECMAScipt language value *Receiver* the following steps are taken:

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. Let *desc* be the result of calling OrdinaryGetOwnProperty with arguments *O* and *P*.
3. ReturnIfAbrupt(*desc*).
4. If *desc* is **undefined**, then
    a. Let *parent* be the result of calling the [[GetInheritance]] internal method of *O*.
    b. ReturnIfAbrupt(*parent*).
    c. If *parent* is **null**, then return **undefined.**
    d. Return the result of calling the [[GetP]] internal methods of *parent* with arguments *P* and *Receiver*.
5. If IsDataDescriptor(*desc*) is **true**, return *desc*.[[Value]].
6. Otherwise, IsAccessorDescriptor(*desc*) must be **true** so, let *getter* be *desc*.[[Get]].
7. If *getter* is **undefined**, return **undefined**.

8. Return the result of calling the [[Call]] internal method of *getter* with *targetThis* as the *thisArgument* and an empty List as *argumentsList*.

8.3.8 [[SetP] (P, V, Receiver)

When the [[SetP]] internal method of *O* is called with property key *P*, value *V*, and ECMAScipt language value *Receiver*, the following steps are taken:

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. Let *ownDesc* be the result of calling OrdinaryGetOwnProperty with arguments *O* and *P*.
3. ReturnIfAbrupt(*ownDesc*).
4. If *desc* is **undefined**, then
    a. Let *parent* be the result of calling the [[GetInheritance]] internal method of *O*.
    b. ReturnIfAbrupt(*parent*).
    c. If *parent* is not **null**, then
        i. Return the result of calling the [[SetP]] internal methods of *parent* with arguments *P*, *V*, and *Receiver*.
    d. Else,
        i. If Type(*Receiver*) is not Object, return **false**.
        ii. Return the result of performing CreateOwnDataProperty(*Receiver*, *P*, *V*).
5. If IsDataDescriptor(*ownDesc*) is **true**, then
    a. If *ownDesc*.[[Writable]] is **false**, return **false**.
    b. If SameValue(*O*, *Receiver*) is **true**, then
        i. Let *valueDesc* be the Property Descriptor {[[Value]]: *V*}.
        ii. Return the result of calling OrdinaryDefineOwnProperty with arguments *O*, *P*, and *valueDesc*.
    c. Else O and *Receiver* are different values,
        i. If Type(*Receiver*) is not Object, return **false**.
        ii. Return the result of performing CreateOwnDataProperty(*Receiver*, *P*, *V*).
6. If IsAccessorDescriptor(*desc*) is **true**, then
    a. Let *setter* be *desc*.[[Set]].
    b. If *setter* is **undefined**, return **false**.
    c. Let *setterResult* be the result of calling the [[Call]] internal method of *setter* providing *Receiver* as *thisArgument* and a new List containing *V* as *argumentsList*.
    d. ReturnIfAbrupt(*setterResult*).
    e. Return **true**.

~~**8.3.9 [[HasProperty]] (P)**~~

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

~~1. Assert: *P* is a valid property key, either a String or a Symbol Object.~~
~~2. Let *has* be the result of calling the [[HasOwnProperty]] internal method of *O* with property key *P*.~~
~~3. ReturnIfAbrupt(*proto*).~~
~~4. If *has* is **true**, return *has*.~~
~~5. Let *proto* be the result of calling the [[GetInheritance]] internal method of *O*.~~
~~6. ReturnIfAbrupt(*proto*).~~
~~7. If *proto* is **null**, then return **false**.~~
~~8. Return the result of calling the [[HasProperty]] internal method of *proto* with argument *P*.~~

~~**8.3.4 [[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,~~

~~i. 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]].~~

~~Exotic objects may define additional constraints upon their [[SetP]] internal method behavior. If possible, exotic objects should not allow [[SetP]] operations in situations where this definition of [[CanPut]] returns false.~~

8.3.9 [[Delete]] (P)

When the [[Delete]] internal method of *O* is called with property key *P* the following steps are taken:

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. Let *desc* be the result of calling OrdinaryGetOwnProperty with arguments *O* and *P*.
3. If *desc* is **undefined**, then return **true**.
4. If *desc*.[[Configurable]] is **true**, then
    a. Remove the own property with name *P* from *O*.
    b. Return **true**.
5. Return **false**.

8.3.10 [[DefineOwnProperty]] (P, Desc)

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

1. Return the result of OrdinaryDefineOwnProperty with arguments *O*, *P*, and *Desc*.

8.3.10.1 OrdinaryDefineOwnProperty (O, P, Desc)

When the abstract operation OrdinaryDefineOwnProperty is called with Object *O*, property key *P*, and property descriptors *Desc* the following steps are taken:

1. Let *current* be the result of calling OrdinaryGetOwnProperty with arguments *O* and *P*.
2. Let *extensible* be the value of the [[Extensible]] internal data propertry of *O*.
3. Return the result of ValidateAndApplyPropertyDescriptor with arguments *O*, *P*, *extensible*, *Desc*, and *current*.

8.3.10.2 IsCompatableDescriptor (Extensible, Desc, Current)

When the abstract operation IsCompatablePropertyDesriptor is called with Boolean value *Extensible*, and property descriptors *Desc*, and *Current* the following steps are taken:

1. Return the result of ValidateAndApplyPropertyDescriptor with arguments **undefined**, **undefined**, *Extensible*, *Desc*, and *Current*.

8.3.10.3 ValidateAndApplyPropertyDescriptor (O, P, extensible, Desc, current)

When the abstract operation ValidateAndApplyPropertyDescriptor is called with Object *O*, property key *P*, Boolean value *extensible*, and property descriptors *Desc*, and *current* the following steps are taken:

This 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**.

NOTE If **undefined** is passed as the *O* argument only validation is performed and not object updates are preformed.

1. Assert: If *O* is not **undefined** then *P* is a valid property key.
2. If *current* is **undefined** and *extensible* is **false**, then return **false**.
3. If *current* is **undefined** and *extensible* is **true**, then
    a. If IsGenericDescriptor(*Desc*) or IsDataDescriptor(*Desc*) is **true**, then
        i. If *O* is not **undefined**, then 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,
        i. If *O* is not **undefined**, then 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**.
4. Return **true**, if every field in *Desc* is absent.
5. 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).
6. If the [[Configurable]] field of *current* is **false** then
    a. Return **false**, if the [[Configurable]] field of *Desc* is **true**.
    b. Return **false**, if the [[Enumerable]] field of *Desc* is present and the [[Enumerable]] fields of *current* and *Desc* are the Boolean negation of each other.
7. If IsGenericDescriptor(*Desc*) is **true**, then no further validation is required.
8. Else if IsDataDescriptor(*current*) and IsDataDescriptor(*Desc*) have different results, then
    a. Return **false**, if the [[Configurable]] field of *current* is **false**.
    b. If IsDataDescriptor(*current*) is **true**, then
        i. If *O* is not **undefined**, then 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. If *O* is not **undefined**, then 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.
9. Else if IsDataDescriptor(*current*) and IsDataDescriptor(*Desc*) are both **true**, then
    a. If the [[Configurable]] field of *current* is **false**, then
        i. Return **false**, 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. Return **false**, 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.
10. Else IsAccessorDescriptor(*current*) and IsAccessorDescriptor(*Desc*) are both **true**,
    a. If the [[Configurable]] field of *current* is **false**, then
        i. Return **false**, if the [[Set]] field of *Desc* is present and SameValue(*Desc*.[[Set]], *current*.[[Set]]) is **false**.
        ii. Return **false**, if the [[Get]] field of *Desc* is present and SameValue(*Desc*.[[Get]], *current*.[[Get]]) is **false**.
11. If *O* is not **undefined**, then
    a. 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.
12. Return **true**.

However, if *O* has an [[BuiltinBrand]] internal data property whose value is BuiltinArray *O* also has a more elaborate [[DefineOwnProperty]] internal method defined in 15.4.5.1.

NOTE Step 10.b allows any field of Desc to be different from the corresponding field of current if current's [[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**.

~~**8.3.11 [[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 Get(*O*, "**`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 *thisArgument* and an empty List as *argumentsList*.~~
~~b. ReturnIfAbrupt(*str*).~~
~~c. If *str* is a primitive value, return *str*.~~
~~4. Let *valueOf* be the result of Get(*O*, "**`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 *thisArgument* and an empty argument list.~~
~~b. ReturnIfAbrupt(*val*).~~
~~c. If *val* is a primitive value, return *val*.~~
~~7. Throw a **TypeError** exception.~~

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

~~1. Let *valueOf* be the result of Get(*O*, "**`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 *thisArgument* and an empty List as *argumentsList*.~~
~~b. ReturnIfAbrupt(*val*).~~
~~c. If *val* is a primitive value, return *val*.~~
~~4. Let *toString* be the result of Get(*O*, "**`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 *thisArgument* and an empty List as *argumentsList*.~~
~~b. ReturnIfAbrupt(*str*).~~
~~c. If *str* is a primitive value, return *str*.~~
~~7. Throw a **TypeError** exception.~~

~~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 ordinary objects can return only primitive values. If an exotic object implements its own [[DefaultValue]] internal method, it must ensure that its [[DefaultValue]] internal method can return only primitive values.~~

8.3.11 [[Enumerate]] ()

When the [[Enumerate]] internal method of *O* is called 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*. The mechanics and order of enumerating the properties is not specified but must conform to the rules specified below.

Enumerated properties do not include properties whose property key is a Symbol. 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 result of calling the [[GetInheritance]] internal method of *O* with no arguments.
3. ReturnIfAbrupt(*proto*).
4. If *proto* is the value **null**, then
    a. Let *propList* be a new empty List.
5. Else
    a. Let *propList* be the result of calling the [[Enumerate]] internal method of *proto*.
6. ReturnIfAbrupt(*propList*).
7. For each *name* that is the property key of an own property of *O*
    a. If Type(*name*) is String, then
        i. Let *desc* be the result of calling OrdinaryGetOwnProperty with arguments *O* and *name*.
        ii. If *name* is an element of *propList*, then remove *name* as an element of *propList*.
        iii. If *desc*.[[Enumerable]] is **true**, then add *name* as an element of *propList*.
8. Order the elements of *propList* in an implementation defined order.
9. Return *propList*.

8.3.12 [[Keys]] ( )

When the [[Keys]] internal method of *O* is called the following steps are taken:

1. Let *result* be a new empty List.
2. If Type(*O*) is not Object, return *result*.
3. For each own property key *P* of *O*
    a. If Type(*P*) is String, then
        i. Let *desc* be the result of calling OrdinaryGetOwnProperty with arguments *O* and *P*.
        ii. If *desc*.[[Enumerable]] is **true**, then
            1. Add *P* as the last element of *result*.
4. Return *result*.

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.

8.3.13 [[OwnPropertyKeys]] ( )

When the [[OwnPropertyKeys]] internal method of *O* is called the following steps are taken:

1. Let *result* be a new empty List.
2. If Type(*O*) is not Object, return *result*.
3. For each own property key *P* of *O*
    a. If *P* is not a private Symbol, then
        i. Add *P* as the last element of *result*.
4. Return *result*.

8.3.14 [[Freeze]] ( )

When the [[Freeze]] internal method of *O* is called the following steps are taken:

1. Return the result of MakeObjectSecure(*O*, **true**).

8.3.15 [[Seal]] ( )

When the [[Seal]] internal method of *O* is called the following steps are taken:

1. Return the result of MakeObjectSecure(*O*, **false**).

8.3.16 [[IsFrozen]] ( )

When the [[IsFrozen]] internal method of *O* is called the following steps are taken:

1. Return the result of TestIfSecureObject (*O*, **true)**.

8.3.17 [[IsSealed]] ( )

When the [[IsSealed]] internal method of *O* is called the following steps are taken:

1. Return the result of TestIfSecureObject (*O*, **false)**.

8.3.18 ObjectCreate Abstract Operation

The abstract operation ObjectCreate with optional argument *proto* (an object or null) is used to specify the runtime creation of new ordinary 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 *obj*'s essential internal methods to the default ordinary object definitions specified in 8.3.
4. Set the [[Prototype]] internal data property of *obj* to *proto*.
5. Set the [[Extensible]] internal data property of *obj* to **true**.
6. Return *obj*.

8.3.19 Ordinary Function Objects

Ordinary function objects encapsulate parameterized ECMAScript code closed over a lexical environment and support the dynamic evaluation of that code. An ordinary function object is an ordinary object and has the same internal data properties and (except as noted below) the same internal methods as other ordinary objects.

Ordinary function objects have the additional internal data properties listed in Table 11. They also have a [[BuiltinBrand]] internal data property whose value is **BuiltinFunction**.

Ordinary function objects provide alternative definitions for the [[GetP]] and [[GetOwnProperty]] internal methods. These alternatives prevent the value of strict mode function from being revealed as the value of a function object property named "**`caller`**". These alternative definitions exist sole to preclude a non-standard legacy feature of some ECMAScript implementations from revealing information about strict mode callers. If an implementation does not provide such a feature, it need not implement these alternative internal methods for ordinary function objects.

Table 11 -- Internal Data Properties of Ordinary Function Objects

Internal Data Property	*Type*	*Description*
[[Scope]]	Lexical Environment	The Lexical Environment that the function was closed over. Is used as the outer environment when evaluating the code of the function.
[[FormalParameters]]	Parse Node	The root parse node of the source code that defines the function's formal parameter list.
[[Code]]	Parse Node	The root parse node of the source code that defines the function's body.
[[Realm]]	Realm Record	The Code Realm in which the function was created and which provides any intrinsic objects that are accessed when evaluating the function.
[[ThisMode]]	(lexical, strict, global)	Defines how **`this`** references are interpreted within the formal parameters and code body of the function. **lexical** means that **`this`** refers to the **this** value of a lexically enclosing function. **strict** means that the **this** value is used exactly as provided by an invocation of the function. **global** means that a this value of **undefined** is interpreted as a reference to the global object.
[[Strict]]	Boolean	**true** if this is a strict mode function, **false** this is not a strict mode function.
[[Home]]	Object	If the function uses **`super`**, this is the object whose [[Inheritance]] provides the object where super property lookups begin. Not present for functions that don't reference **`super`**.
[[MethodName]]	String or Symbol	If the function uses **`super`**, this is the property keys that is used for unqualified references to **`super`**. Not present for functions that don't reference **`super`**.

Ordinary function objects all have the [[Call]], [[GetP]] and [[GetOwnProperty]] internal methods defined here. Oridinary functions that are also constructors in addition have the [[Construct]] internal method.

8.3.19.1 [[Call]] Internal Method

The [[Call]] internal method for an ordinary Function object *F* is called with parameters *thisArgument* and *argumentsList*, a List of ECMAScript language values. The following steps are taken:

1. Let *callerContext* be the running execution context.
2. If, *callerContext* is not already suspended, then Suspend *callerContext*.
3. Let *calleeContext* be a new ECMAScript Code execution context.
4. Let *calleeRealm* be the value of *F's* [[Realm]] internal data property.
5. Set *calleeContext's* Realm *calleeRealm*.
6. Let *thisMode* be the value of *F*'s [[ThisMode]] internal data property.
7. If *thisMode* is lexical, then
    a. Let *localEnv* be the result of calling NewDeclarativeEnvironment passing the value of the [[Scope]] internal data property of *F* as the argument.
8. Else,
    a. If *thisMode* is strict, set *thisValue* to *thisArgument*.
    b. Else
        i. if *thisArgument* is **null** or **undefined**, then
            1. Set *thisValue* to *calleeRealm*.[[globalThis]].
        ii. Else if Type(*thisArgument*) is not Object, set the *thisValue* to ToObject(*thisArgument*).
        iii. Else set the *thisValue* to *thisArgument*.
    c. Let *localEnv* be the result of calling NewFunctionEnvironment passing *F* and *thisValue* as the arguments.
9. Set the LexicalEnvironment of *calleeContext* to *localEnv*.
10. Set the VariableEnvironment of *calleeContext* to *localEnv*.

11. Push calleeContext on to the execution context stack; *calleeContext* is now the running execution context.
12. Let *status* be the result of performing Function Declaration Instantiation using the function *F*, *argumentsList* , and *localEnv* as described in 10.5.3.
13. If *status* is an abrupt completion, then
    a. Remove calleeContext from the execution context stack and restore *callerContext* as the running execution context.
    b. Return *status*.
14. Let *result* be the result of evaluating the *FunctionBody* that is the value of *F*'s [[Code]] internal data property.
15. Remove calleeContext from the execution context stack and restore *callerContext* as the running execution context.
16. If *result*.type is return then return NormalCompletion(*result*.[[value]]).
17. Return *result*.

8.3.19.2 [[Construct]] Internal Method

The [[Construct]] internal method for an ordinary Function object *F* is called with a single parameter *argumentsList* which is a possibly empty List of ECMAScript language values. The following steps are taken:

1. Let *proto* be the result of Get(*F*, **"prototype"**).
2. ReturnIfAbrupt(*proto*).
3. If Type(*proto*) is Object, let *obj* be the result of the abstract operation ObjectCreate with argument *proto*.
4. Else, let *obj* be the result of the abstract operation ObjectCreate.
5. Let *result* be the result of calling the [[Call]] internal method of *F*, providing *obj* and *argumentsList* as the arguments.
6. ReturnIfAbrupt(*result*).
7. If Type(*result*) is Object then return *result*.
8. Return NormalCompletion(*obj*).

8.3.19.3 [[GetP]] (P, Receiver)

When the [[GetP]] internal method of ordinary function object *F* is called with property key *P* and ECMAScipt language value *Receiver* the following steps are taken:

1. Let *v* be the result of calling the default ordinary object [[GetP]] internal method (8.3.7) on *F* passing *P* and *Receiver* as arguments.
2. ReturnIfAbrupt(*v*).
3. If *P* is **"caller"** and *v* is a strict mode Function object, return **undefined**.
4. Return *v*.

8.3.19.4 [[GetOwnProperty]] (P)

When the [[GetOwnProperty]] internal method of ordinary function object *F* is called with property key *P*, the following steps are taken:

1. Let *v* be the result of calling the default ordinary object [[GetOwnProperty]] internal method (8.3.6) on *F* passing *P* as the argument.
2. ReturnIfAbrupt(*v*).
3. If IsDataDescriptor(*v*) is **true**, then
    a. If *P* is **"caller"** and *v*.[[Value]] is a strict mode Function object, then
        i. Set *v*.[[Value]] to **undefined** *v*.[[Value]].
4. Return *v*.

8.4 Built-in Exotic Object Internal Methods and Data Fields

This specification define several kinds of built-in exotic objects. These objects generally behave similar to ordinary objects except for a few specific situtations. The following exotic objects use the ordinary object internal methods except where it is explicitly specified wise below:

8.4.1 Bound Function Exotic Objects

A *bound function* is an exotic object that wrappers another function object. A bound function is callable (it has [[Call]] and [[Construct]] internal methods). Calling a bound function generally results in a call of its wrappered function.

Bound function objects do not have the internal data properties of ordinary function objects defined in Table 11. Instead they have the internal data properties defined in Table 12. They also have a [[BuiltinBrand]] internal data property whose value is **BuiltinFunction**.

Table 12 -- Internal Data Properties of Exotic Bound Function Objects

Internal Data Property	*Type*	*Description*
[[BoundTargetFunction]]	Callable Object	The wrappered function object.
[[BoundThis]]	Any	The value that is always passed as the **this** value when calling the wrappered function.
[[BoundArguments]]	List of Any	A list of values that whose elements are used as the first arguments to any call to the wrappered function.

Unlike ordinary function objects, bound function objects do not use alternative definitions of the [[GetP]] and [[GetOwnPropety]] internal methods. Bound function objects provide all of the essential internal methods as specified in 8.3. However, they use the following definitions for the essential internal methods of function objects.

8.4.1.1 [[Call]]

When the [[Call]] internal method of an exotic bound function object, *F*, which was created using the bind function is called with parameters *thisArgument* and *argumentsList*, a List of ECMAScript language values, the following steps are taken:

1. Let *boundArgs* be the value of *F's* [[BoundArguments]] internal data property.
2. Let *boundThis* be the value of *F's* [[BoundThis]] internal data property.
3. Let *target* be the value of *F's* [[BoundTargetFunction]] internal data 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 *argumentsList* in the same order.
5. Return the result of calling the [[Call]] internal method of *target* providing *boundThis* as *thisArgument* and providing *args* as *argumentsList*.

8.4.1.2 [[Construct]]

When the [[Construct]] internal method of an exotic bound 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* [[BoundTargetFunction]] internal data property.
2. If *target* has no [[Construct]] internal method, a **TypeError** exception is thrown.
3. Let *boundArgs* be the value of *F's* [[BoundArguments]] internal data 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.

8.4.1.3 BoundFunctionCreate Abstract Operation

The abstract operation BoundFunctionCreate with arguments *targetFunction*, *boundThis* and *boundArgs* is used to specify the creation of new Object objects. It performs the following steps:

1. Let *proto* be the the intrinsic %FunctionPrototype%.
2. Let *obj* be a newly created ECMAScript object.
3. Set *obj*'s essential internal methods to the default ordinary object definitions specified in 8.3.
4. Set the [[Call]] internal method of *obj* as described in 8.4.1.1.
5. Set the [[Construct]] internal method of *F* as described in 8.4.1.2.

6. Set the [[Prototype]] internal data property of *obj* to *proto*.
7. Set the [[Extensible]] internal data property of *obj* to **true**.
8. Set the [[BoundTargetFunction]] internal data property of *obj* to *targetFunction*.
9. Set the [[BoundThis]] internal data property of *obj* to the value of *boundThis*.
10. Set the [[BoundArguments]] internal data property of *obj* to *boundArgs*.
11. Add the [[BuiltinBrand]] internal data property with value BuiltinFunction to *obj*.
12. Return *obj*.

8.4.2 Array Exotic Objects

An *Array object* is an exotic object 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 ToString(ToUint32(*P*)) is equal to *P* and 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.

Exotic Array objects always have a non-configurable property named **`"length"`**.

Exotic Array objects have the same internal data properties as ordinary objects. They also have a [[BuiltinBrand]] internal data property whose value is **BuiltinArray**.

Exotic Array objects provide alternative definitions for the [[SetP]] and [[DefineOwnProperty]] internal methods. Except for these two internal methods, exotic Array objects provide all of the other essential internal methods as specified in 8.3.

8.4.2.1 [[SetP] (P, V, Receiver)

When the [[SetP]] internal method of an an exotic Array object *O* is called with property key *P*, value *V*, and ECMAScipt language value *Receiver*, the following steps are taken:

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. Let *ownDesc* be the result of calling OrdinaryGetOwnProperty with arguments *O* and *P*.
3. ReturnIfAbrupt(*ownDesc*).
4. If *desc* is **undefined**, then
    a. Let *parent* be the result of calling the [[GetInheritance]] internal method of *O*.
    b. ReturnIfAbrupt(*parent*).
    c. If *parent* is not **null**, then
        i. Return the result of calling the [[SetP]] internal methods of *parent* with arguments *P*, *V*, and *Receiver*.
    d. Else,
        i. If Type(*Receiver*) is not Object, return **false**.
        ii. Return the result of performing CreateOwnDataProperty(*Receiver*, *P*, *V*).
5. If IsDataDescriptor(*ownDesc*) is **true**, then
    a. If *ownDesc*.[[Writable]] is **false**, return **false**.
    b. If SameValue(*O*, *Receiver*) is **true**, then
        i. Let *valueDesc* be the Property Descriptor {[[Value]]: *V*}.
        ii. If *P* is **`"length"`**, then
            1. Return the result of calling ArraySetLength with arguments *O*, and *valueDesc*.
        iii. Else,
            1. Return the result of calling OrdinaryDefineOwnProperty with arguments *O*, *P*, and *valueDesc*.
    c. Else O and *Receiver* are different values,
        i. If Type(*Receiver*) is not Object, return **false**.

ii. Return the result of performing CreateOwnDataProperty(*Receiver*, *P*, *V*).

6. If IsAccessorDescriptor(*desc*) is **true**, then
   a. Let *setter* be *desc*.[[Set]].
   b. If *setter* is **undefined**, return **false**.
   c. Let *setterResult* be the result of calling the [[Call]] internal method of *setter* providing *Receiver* as *thisArgument* and a new List containing *V* as *argumentsList*.
   d. ReturnIfAbrupt(*setterResult*).
   e. Return **true**.

NOTE This algorithm differs from the ordinary object [[SetP]] algorithm only in the handling of properties with the name**"length"** in step 5.b.ii.1.

8.4.2.2 [[DefineOwnProperty]] (P, Desc)

When the [[DefineOwnProperty]] internal method of an exotic Array object *A* is called with property *P*, and Property Descriptor *Desc* the following steps are taken:

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. If *P* is **"length"**, then
   a. Return the result of calling ArraySetLength with arguments *A*, and *Desc*.
3. Else if *P* is an array index, then
   a. 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.
   b. Let *oldLen* be *oldLenDesc*.[[Value]].
   c. Let *index* be ToUint32(*P*).
   d. ReturnIfAbrupt(*index*).
   e. Reject if *index* ≥ *oldLen* and *oldLenDesc*.[[Writable]] is **false**.
   f. Let *succeeded* be the result of calling OrdinaryDefineOwnProperty passing *A*, *P*, and *Desc* as arguments.
   g. ReturnIfAbrupt(*succeeded*).
   h. If *succeeded* is **false**, then return **false**.
   i. If *index* ≥ *oldLen*
      i. Set *oldLenDesc*.[[Value]] to *index* + 1.
      ii. Let *succeeded* be the result of calling OrdinaryDefineOwnProperty passing *A*, **"length"**, and *oldLenDesc* as arguments.
      iii. ReturnIfAbrupt(*succeeded*).
   j. Return **true**.
4. Return the result of calling OrdinaryDefineOwnProperty passing *A*, *P*, and *Desc* as arguments.

8.4.2.3 ArrayCreate Abstract Operation

The abstract operation ArrayCreate with argument *length* (a positive integer) is used to specify the creation of new exotic Array objects. It performs the following steps:

1. Let *A* be a newly created ECMAScript object.
2. Set *A*'s essential internal methods to the default ordinary object definitions specified in 8.3.
3. Set the [[SetP]] internal method of *A* as specified in 8.4.2.1.
4. Set the [[DefineOwnProperty]] internal method of *A* as specified in 8.4.2.2.
5. Set the [[Prototype]] internal data property of *A* to the intrinsic object %ArrayPrototype%.
6. Set the [[BuiltinBrand]] internal data property of *A* to the value BuiltinArray.
7. Set the [[Extensible]] internal data property of *A* to **true**.
8. Call OrdinaryDefineOwnProperty with arguments *O*, **"length"** and Property Descriptor {[[Value]]: *length*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false**}.
9. Return A.

8.4.2.3 ArraySetLength Abstract Operation

When the abstract operation ArraySetLength is called with an exotic Array object *A*, and Property Descriptor *Desc* 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 the [[Value]] field of *Desc* is absent, then
    a. Return the result of calling OrdinaryDefineOwnProperty passing *A*, **"length"**, and *Desc* as arguments.
4. Let *newLenDesc* be a copy of *Desc*.
5. Let *newLen* be ToUint32(*Desc*.[[Value]]).
6. If *newLen* is not equal to ToNumber(*Desc*.[[Value]]), throw a **RangeError** exception.
7. Set *newLenDesc*.[[Value]] to *newLen*.
8. If *newLen* ≥*oldLen*, then
    a. Return the result of calling OrdinaryDefineOwnProperty passing *A*, **"length"**, and *newLenDesc* as arguments.
9. If *oldLenDesc*.[[Writable]] is **false**, then return **false**.
10. If *newLenDesc*.[[Writable]] is absent or has the value **true**, let *newWritable* be **true**.
11. Else,
    a. Need to defer setting the [[Writable]] attribute to **false** in case any elements cannot be deleted.
    b. Let *newWritable* be **false**.
    c. Set *newLenDesc*.[[Writable]] to **true**.
12. Let *succeeded* be the result of calling OrdinaryDefineOwnProperty passing *A*, **"length"**, and *newLenDesc* as arguments.
13. ReturnIfAbrupt(*succeeded*).
14. If *succeeded* is **false**, return **false**.
15. While *newLen* < *oldLen* repeat,
    a. Set *oldLen* to *oldLen* – 1.
    b. Let *deleteSucceeded* be the result of calling the [[Delete]] internal method of *A* passing ToString(*oldLen*).
    c. ReturnIfAbrupt(*succeeded*).
    d. If *deleteSucceeded* is **false**, then
        i. Set *newLenDesc*.[[Value]] to *oldLen+1*.
        ii. If *newWritable* is **false**, set *newLenDesc*.[[Writable]] to **false**.
        iii. Let *succeeded* be the result of calling OrdinaryDefineOwnProperty passing *A*, **"length"**, and *newLenDesc* as arguments.
        iv. ReturnIfAbrupt(*succeeded*).
        v. Return **false**.
16. If *newWritable* is **false**, then
    a. Call OrdinaryDefineOwnProperty passing *A*, **"length"**, and Property Descriptor{[[Writable]]: **false**} as arguments. This call will always return **true**.
17. Return **true**.

8.4.3 String Exotic Objects

A *String object* is an exotic object that encapsulates a String value and exposes virtual array index data properties corresponding to the individual code unit elements of the string value. Exotic String objects always have a data property named **"length"** whose value is the number of code unit elements in the encapsulated String value. Both the code unit data properties and the **"length"** property are non-writable and non-configurable.

Exotic String objects have the same internal data properties as ordinary objects. They also have a [[StringData]] internal data property and a [[BuiltinBrand]] internal data property whose value is **BuiltinStringWrapper**.

Exotic String objects provide alternative definitions for the following internal methods. All of the other exotic String object essential internal methods that are not defined below are as specified in 8.3.

8.4.3.1 [[HasOwnProperty]] (P)

When the [[HasOwnProperty]] internal method of exotic String object *O* is called with property key *P*, the following steps are taken:

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. Let *has* be the result of calling the ordinary object [[HasOwnProperty]] internal method (8.3.5) on *O* with argument *P*.
3. ReturnIfAbrupt(*has*).
4. If *has* is **true**, then return **true**.
5. Let *index* be ToInteger(*P*).
6. ReturnIfAbrupt(*index*).
7. Let *absIntIndex* be ToString(abs(*index*)).
8. ReturnIfAbrupt(*absIntIndex*).
9. If SameValue(*absIntIndex*, *P*) is **false** return **false**.
10. Let *str* be the String value of the [[StringData]] internal property of *O*.
11. Let *len* be the number of elements in *str*.
12. If *len* ≤ *index*, return **false**.
13. Return **true**.

8.4.3.2 [[GetOwnProperty]] (P)

When the [[GetOwnProperty]] internal method of an an exotic String object *S* is called with property key *P* the following steps are taken:

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. Let *desc* be the result of OrdinaryGetOwnProperty(*S*, *P*).
3. ReturnIfAbrupt(*desc*).
4. If *desc* is not **undefined** return *desc*.
5. Let *index* be ToInteger(*P*).
6. ReturnIfAbrupt(*index*).
7. Let *absIntIndex* be ToString(abs(*index*)).
8. ReturnIfAbrupt(*absIntIndex*).
9. If SameValue(*absIntIndex*, *P*) is **false** return **undefined**.
10. Let *str* be the String value of the [[StringData]] internal data property of *S*.
11. Let *len* be the number of elements in *str*.
12. If *len* ≤ *index*, return **undefined**.
13. Let *resultStr* be a String value of length 1, containing one code unit from *str*, specifically the code unit at position *index*, where the first (leftmost) element in *str* is considered to be at position 0, the next one at position 1, and so on.
14. Return a Property Descriptor { [[Value]]: *resultStr*, [[Enumerable]]: **true**, [[Writable]]: **false**, [[Configurable]]: **false** }.

8.4.3.3 [[GetP]] (P, Receiver)

When the [[GetP]] internal method of exotic String object *O* is called with property key *P* and ECMAScipt language value *Receiver* the following steps are taken:

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument *P*.
3. ReturnIfAbrupt(*desc*).
4. If *desc* is **undefined**, then
    a. Let *parent* be the result of calling the [[GetInheritance]] internal method of *O*.
    b. ReturnIfAbrupt(*parent*).
    c. If *parent* is **null**, then return **undefined.**
    d. Return the result of calling the [[GetP]] internal methods of *parent* with arguments *P* and *Receiver*.
5. If IsDataDescriptor(*desc*) is **true**, return *desc*.[[Value]].
6. Otherwise, IsAccessorDescriptor(*desc*) must be **true** so, let *getter* be *desc*.[[Get]].
7. If *getter* is **undefined**, return **undefined**.

8. Return the result of calling the [[Call]] internal method of *getter* with *targetThis* as the *thisArgument* and an empty List as *argumentsList*.

NOTE This algorithm differs from the ordinary object [[SetP]] algorithm only in invocation of [[GetOwnProperty]] in step 2.

8.4.3.4 [[Delete]] (P)

When the [[Delete]] internal method of exotic String object *O* is called with property name *P* the following steps are taken:

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument *P*.
3. If *desc* is **undefined**, then return **true**.
4. If *desc*.[[Configurable]] is **true**, then
    a. Remove the own property with name *P* from *O*.
    b. Return **true**.
5. Return **false**.

NOTE This algorithm differs from the ordinary object [[Delete]] algorithm only in invocation of [[GetOwnProperty]] in step 2.

8.4.3.5 [[DefineOwnProperty]] (P, Desc)

When the [[DefineOwnProperty]] internal method of an exotic String object *O* is called with property *P*, and Property Descriptor *Desc* the following steps are taken:

1. Let *current* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument *P*.
2. Let *extensible* be the result of calling the [[GetExtensible]] internal method of *O*.
3. Return the result of ValidateAndApplyPropertyDescriptor with arguments *O*, *P*, *extensible*, *Desc*, and *current*.

NOTE This algorithm differs from the ordinary object OrdinaryDefineOwnProperty abstract operation algorithm only in invocation of [[GetOwnProperty]] in step 1.

8.4.3.6 [[Enumerate]] ()

When the [[Enumerate]] internal method of an exotic String object *O* is called the following steps are taken:

8.4.3.7 [[Keys]] ( )

When the [[Keys]] internal method of an exotic String object *O* is called the following steps are taken:

8.4.3.8 [[OwnPropertyKeys]] ( )

When the [[OwnPropertyKeys]] internal method of an exotic String object *O* is called the following steps are taken:

8.4.4 Exotic Symbol Objects

An *Symbol object* is an exotic object that may be used as a property key. Symbol exotic objects are unique in that they are always immutable and never observably reference any other object.

Exotic String objects have the a single internal data properties named [[Private]] that is set when the object is created and never modified.

Exotic Symbol objects provide alternative definitions for all of the essential internal methods.

8.4.4.1 [[GetInheritance]] ( )

When the [[GetInheritance]] internal method of an exotic Symbol object *O* is called the following steps are taken:

1. Return **null**.

8.4.4.2 [[SetInheritance]] (V)

When the [[SetInheritance]] internal method of an exotic Symbol object *O* is called with argument *V* the following steps are taken:

1. Assert: Either Type(*V*) is Object or Type(*V*) is Null.
2. Return **false**.

8.4.4.3 [[IsExtensible]] ( )

When the [[IsExtensible]] internal method of an exotic Symbol object *O* is called the following steps are taken:

1. Return **false**.

8.4.4.4 [[PreventExtensions]] ( )

When the [[PreventExtensions]] internal method of an exotic Symbol object an exotic Symbol object *O* is called the following steps are taken:

1. Return NormalCompletion(empty).

8.4.4.5 [[HasOwnProperty]] (P)

When the [[HasOwnProperty]] internal method of an exotic Symbol object *O* is called with property key *P*, the following steps are taken:

1. Return **false**.

8.4,4.6 [[GetOwnProperty]] (P)

When the [[GetOwnProperty]] internal method of an exotic Symbol object *O* is called with property key *P*, the following steps are taken:

1. Return **undefined**.

8.4.4.7 [[GetP]] (P, Receiver)

When the [[GetP]] internal method of an exotic Symbol object *O* is called with property key *P* and ECMAScipt language value *Receiver* the following steps are taken:

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. If *P* is **"`toString`"**, then
    a. Let *ctx* be the running execution context.
    b. Let *ctxRealm* be *ctx*'s Realm component.
    c. Return *ctxRealm*.[[intrinsics]].% ObjProto_toString %.
3. Return **undefined**.

8.4.4.8 [[SetP] (P, V, Receiver)

When the [[SetP]] internal method of an exotic Symbol object *O* is called with property key *P*, value *V*, and ECMAScipt language value *Receiver*, the following steps are taken:

1. Return **false**.

8.4.4.9 [[Delete]] (P)

When the [[Delete]] internal method of an exotic Symbol object *O* is called with property key *P* the following steps are taken:

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. Return **true**.

8.4.4.10 [[DefineOwnProperty]] (P, Desc)

When the [[DefineOwnProperty]] internal method of an exotic Symbol object *O* is called with property key *P* and property descriptor *Desc*, the following steps are taken:

1. Return **false**.

8.4.4.11 [[Enumerate]] ()

When the [[Enumerate]] internal method of an exotic Symbol object *O* is called the following steps are taken:

1. Return an Iterator object (reference xxxx) whose next method immediately throws %StopIteration% and forms no other action..

8.4.4.12 [[Keys]] ( )

When the [[Keys]] internal method of an exotic Symbol object *O* is called the following steps are taken:

1. Return a new empty List.

8.4.4.13 [[OwnPropertyKeys]] ( )

When the [[OwnPropertyKeys]] internal method of an exotic Symbol object *O* is called the following steps are taken:

1. Return a new empty List.

8.4.4.14 [[Freeze]] ( )

When the [[Freeze]] internal method of an exotic Symbol object *O* is called the following steps are taken:

1. Return **true**.

8.4.4.15 [[Seal]] ( )

When the [[Seal]] internal method of an exotic Symbol object *O* is called the following steps are taken:

1. Return **true**

8.4.4.16 [[IsFrozen]] ( )

When the [[IsFrozen]] internal method of an exotic Symbol object *O* is called the following steps are taken:

1. Return **true**.

8.4.4.17 [[IsSealed]] ( )

When the [[IsSealed]] internal method of an exotic Symbol object *O* is called the following steps are taken:

1. Return **true**.

8.4.5 Exotic Arguments Objects

An *arguments object* is an exotic object whose array index properties map to the formal parameters of a non-strict function invocation.

Exotic arguments objects have the same internal data properties as ordinary objects. They also have a [[ParsmeterMap]] internal data property and a [[BuiltinBrand]] internal data property whose value is **BuiltinArguments**.

Exotic arguments objects provide alternative definitions for the following internal methods. All of the other exotic arguments object essential internal methods that are not defined below are as specified in 8.3.

8.4.5 Typed Array Exotic Objects

8.4.6 Built-in Function Objects

The function objects specified in Clause 15 may be implemented as either ordinary function objects whose behaviour is provided using ECMAScript code or as implementation provided exotic function objects whose behaviour is provide in some other manner. In either case, the effect of calling such functions must be that specified for each one in Clause 15.

If an implementation provided exotic object is used, the objects must have non-function the ordinary object behaviour specified in 8.3 except for [[GetP]] and [[GetOwnProperty]] which must be as specified in 8.3.19. All such exotic function objects also have [[Prototype]] and [[Extensible]] internal data properties and a [[BuiltinBrand]] internal data property whose value is **BuiltinFunction**.

[[Calll]] and [[Construct]]

8.5 Proxy Object Internal Methods and Internal Data Properties

A proxy object is an exotic object whose essential internal methods are partially implemented using ECMAScript code. Every proxy objects has an internal data property called [[ProxyHandler]]. The value of [[ProxyHandler]] is always an object, called the proxy's *handler object*. Methods of a handler object may be used to augment the implementation for one or more of the proxy object's internal methods. Every proxy object also has an internal data property called [[ProxyTarget]] whose value is usually an object. This object is called the proxy's *target object*.

When a handler method is called to provide the implementation of a proxy object internal method, the handler method is passed the proxy's target object as a parameter. A proxy's handler object does not necessarily have a method corresponding to every essential internal method. Invoking an internal method on the proxy results in the invocation of the corresponding internal method on the proxy's target object is the handler object does not have a method corresponding to the internal trap.

The [[ProxyHandler]] and [[ProxyTarget]] internal data properties of a proxy object are always initialized when the object is created and typically may not be modified. Some proxy objects are created in a manner that permits them to be subsequent *revoked*. When a proxy is revoked, its [[ProxyHander]] internal data property is set to a special revoked proxy handler object and its [[ProxyTarget]] internal data property is set to **null**.

Because proxy permit arbitrary ECMAScript code to be used to in the implementation of internal methods, it is possible to define a proxy object that violates the invariants defined in 8.1.6.2. An ECMAScript implementation must be robust in the presence of such violations. Some of the internal method invariants defined in 8.1.6.2 are essential integrity invariants. These invariants are explicitly enforced by the proxy internal methods specified in this section.

In the following algorithm descriptions, assume O is an ECMAScript proxy object, P is a property key value, V is any ECMAScript language value, Desc is a Property Description record, and B is a Boolean flag.

8.5.1 [[GetInheritance]] ( )

When the [[GetInheritance]] internal method of proxy object *O* is called the following steps are taken:

1. Let *handler* the value of the [[ProxyHandler]] internal data property of *O*.
2. Let *target* the value of the [[ProxyTarget]] internal data property of *O*.
3. Let *trap* be the result of GetMethod(*handler*, "**`getPrototypeOf`**").
4. ReturnIfAbrupt(*trap*).
5. If *trap* is **undefined**, then
    a. Return the result of calling the [[GetInheritance]] internal method of *target*.
6. Let *handlerProto* be the result of calling *trap* with *handler* as the **this** value and a new List containing *target*.
7. ReturnIfAbrupt(*trapResult*).
8. Let *targetProto* be the result of calling the [[GetInheritance]] internal method of *target*.
9. ReturnIfAbrupt(*targetProto*).
10. If SameValue(*handlerProto*, *targetProto*) is **false**, then throw a TypeError exception.
11. Return *handlerProto*.

NOTE [[GetInheritance] for proxy objects enforces the following invariant:

- [[GetInheritance] applied to the proxy object must return the same value as [[GetInheritance] applied to the proxy object's handler object.

8.5.2 [[SetInheritance]] (V)

When the [[SetInheritance]] internal method of proxy object *O* is called with argument *V* the following steps are taken:

1. Assert: Either Type(*V*) is Object or Type(*V*) is Null.
2. Let *handler* the value of the [[ProxyHandler]] internal data property of *O*.
3. Let *target* the value of the [[ProxyTarget]] internal data property of *O*.
4. Let *trap* be the result of GetMethod(*handler*, "**`setPrototypeOf`**").
5. ReturnIfAbrupt(*trap*).
6. If *trap* is **undefined**, then
    a. Return the result of calling the [[SetInheritance]] internal method of *target* with argument *V*.
7. Let *trapResult* be the result of calling *trap* with *handler* as the **this** value and a new List containing *target* and *V*.
8. ReturnIfAbrupt(*trapResult*).
9. Let *getProtoTrap* be the result of GetMethod(*handler*, "**`getPrototypeOf `**").
10. ReturnIfAbrupt(*getProtoTrap*).
11. If *getProtoTrap* is **undefined**, then
    a. Return *trapResult*.
12. Let *getProtoResult* be the result of calling *getProtoTrap* with *handler* as the **this** value and a new List containing *target*.
13. ReturnIfAbrupt(*getProtoResult*).
14. Let *targetProto* be the result of calling the [[GetInheritance]] internal method of *target*.
15. ReturnIfAbrupt(*targetProto*).
16. If SameValue(*getProtoResult*, *targetProto*) is **false**, then throw a TypeError exception.
17. Return *trapResult*.

NOTE [[SetInheritance] for proxy objects enforces the following invariant:

- After a [[SetInheritance]] call, [[GetInheritance] applied to the proxy object must return the same value as [[GetInheritance] applied to the proxy object's handler object.

8.5.3 [[IsExtensible]] ( )

When the [[IsExtensible]] internal method of proxy object *O* is called the following steps are taken:

1. Let *handler* the value of the [[ProxyHandler]] internal data property of *O*.
2. Let *target* the value of the [[ProxyTarget]] internal data property of *O*.
3. Let *trap* be the result of GetMethod(*handler*, "**`isExtensible`**").

4. ReturnIfAbrupt(*trap*).
5. If *trap* is **undefined**, then
    a. Return the result of calling the [[IsExtensible]] internal method of *target*.
6. Let *trapResult* be the result of calling *trap* with *handler* as the **this** value and a new List containing *target*.
7. ReturnIfAbrupt(*trapResult*).
8. Let *proxyIsExtensible* be ToBoolean(*trapResult*).
9. Let *targetIsExtensible* be the result of calling the [[IsExtensible]] internal method of *target*.
10. ReturnIfAbrupt(*targetIsExtensible*).
11. If SameValue(*proxyIsExtensible*, *targetIsExtensible*) is **false**, then throw a TypeError exception.
12. Return *proxyIsExtensible*.

NOTE [[IsExtensible] for proxy objects enforces the following invariant:

- [[IsExtensible] applied to the proxy object must return the same value as [[IsExtensible] applied to the proxy object's handler object.

8.5.4 [[PreventExtensions]] ( )

When the [[PreventExtensions]] internal method of proxy object *O* is the following steps are taken:

1. Let *handler* the value of the [[ProxyHandler]] internal data property of *O*.
2. Let *target* the value of the [[ProxyTarget]] internal data property of *O*.
3. Let *trap* be the result of GetMethod(*handler*, "**`preventExtensions`**").
4. ReturnIfAbrupt(*trap*).
5. If *trap* is **undefined**, then
    a. Return the result of calling the [[PreventExtensions]] internal method of *target*.
6. Let *trapResult* be the result of calling *trap* with *handler* as the **this** value and a new List containing *target*.
7. ReturnIfAbrupt(*trapResult*).
8. Let *isTrap* be the result of GetMethod(*handler*, "**`isExtensible`**").
9. ReturnIfAbrupt(*isTrap*).
10. If *isTrap* is **undefined**, then
    a. Return NormalCompletion(empty).
11. Let *isTrapResult* be the result of calling *isTrap* with *handler* as the **this** value and a new List containing *target*.
12. ReturnIfAbrupt(*isTrapResult*).
13. Let *proxyIsExtensible* be ToBoolean(*isTrapResult*).
14. Let *targetIsExtensible* be the result of calling the [[IsExtensible]] internal method of *target*.
15. ReturnIfAbrupt(*targetIsExtensible*).
16. If SameValue(*proxyIsExtensible*, *targetIsExtensible*) is **false**, then throw a TypeError exception.
17. Return NormalCompletion(empty).

NOTE [[PreventExtensions] for proxy objects enforces the following invariant:

- After a [[PreventExtensions]] call, [[IsExtensible] applied to the proxy object must return the same value as [[IsExtensible] applied to the proxy object's handler object.

8.5.5 [[HasOwnProperty]] (P)

When the [[HasOwnProperty]] internal method of proxy object *O* is called with property key *P*, the following steps are taken:

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. Let *handler* the value of the [[ProxyHandler]] internal data property of *O*.
3. Let *target* the value of the [[ProxyTarget]] internal data property of *O*.
4. Let *trap* be the result of GetMethod(*handler*, "**`hasOwn`**").
5. ReturnIfAbrupt(*trap*).
6. If *trap* is **undefined**, then
    a. Return the result of calling the [[HasOwnProperty]] internal method of *target* with argument *P*.
7. Let *trapResult* be the result of calling *trap* with *handler* as the **this** value and a new List containing *target* and *P*.
8. ReturnIfAbrupt(*trapResult*).
9. Let *success* be ToBoolean(*trapResult*).

10. If *success* is **false**, then
    a. Let *targetDesc* be the result of calling the [[GetOwnProperty]] internal method of *target* with argument *P*.
    b. ReturnIfAbrupt(*targetDesc*).
    c. If *targetDesc* is not **undefined**, then
        i. If *targetDesc*.[[Configurable]] is **false**, then throw a **TypeError** exception.
        ii. Let *extensibleTarget* be the result of calling the [[IsExtensible]] internal method of *target*.
        iii. ReturnIfAbrupt(*extensibleTarget*).
        iv. If ToBoolean(*extensibleTarget*) is **false**, then throw a **TypeError** exception.
11. Else *success* is **true**,
    a. If *targetDesc* is **undefined**, then
        i. Let *extensibleTarget* be the result of calling the [[IsExtensible]] internal method of *target*.
        ii. ReturnIfAbrupt(*extensibleTarget*).
        iii. If ToBoolean(*extensibleTarget*) is **false**, then throw a **TypeError** exception.
12. Return *success*.

NOTE [[HasOwnProerty] for proxy objects enforces the following invariants:

- A property cannot be reported as non-existent, if it exists as a non-configurable own property of the target object.
- A property cannot be reported as non-existent, if it exists as a own property of the target object and the target object is not extensible.
- A property cannot be reported as existent, if it does not exists as a own property of the target object and the target object is not extensible.

8.5.6 [[GetOwnProperty]] (P)

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

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. Let *handler* the value of the [[ProxyHandler]] internal data property of *O*.
3. Let *target* the value of the [[ProxyTarget]] internal data property of *O*.
4. Let *trap* be the result of GetMethod(*handler*, "**`getOwnPropertyDescriptor`**").
5. ReturnIfAbrupt(*trap*).
6. If *trap* is **undefined**, then
    a. Return the result of calling the [[GetOwnProperty]] internal method of *target* with argument *P*.
7. Let *trapResultObj* be the result of calling *trap* with *handler* as the **this** value and a new List containing *target* and *P*.
8. ReturnIfAbrupt(*trapResultObj*).
9. If Type(*trapResultObj*) is neither Object or **undefined**, then throw a **TypeError** exception.
10. Let *targetDesc* be the result of calling the [[GetOwnProperty]] internal method of *target* with argument *P*.
11. ReturnIfAbrupt(*targetDesc*).
12. If *trapResult* is **undefined**, then
    a. If *targetDesc* is **undefined**, then return **undefined**.
    b. If *targetDesc*.[[Configurable]] is **false**, then throw a **TypeError** exception.
    c. Let *extensibleTarget* be the result of calling the [[IsExtensible]] internal method of *target*.
    d. ReturnIfAbrupt(*extensibleTarget*).
    e. If ToBoolean(*extensibleTarget*) is **false**, then throw a **TypeError** exception.
    f. Return **undefined**.
13. Let *extensibleTarget* be the result of calling the [[IsExtensible]] internal method of *target*.
14. ReturnIfAbrupt(*extensibleTarget*).
15. Set *extensibleTarget* to ToBoolean(*extensibleTarget*),
16. Let *resultDesc* be ToPropertyDescriptor(*trapResultObj*).
17. ReturnIfAbrupt(*resultDesc*).
18. Call CompletePropertyDescriptor(*resultDesc*, *targetDesc*).
19. Let *valid* be the result of IsCompatablePropertyDesriptor (*extensibleTarget*, *resultDesc*, *targetDesc*).
20. If *valid* is **false**, then throw a **TypeError** exception.
21. If *resultDesc*.[[Configurable]] is **false**, then
    a. If *targetDesc* is not **undefined** and *targetDesc*.[[Configurable]] is **true**,then
        i. Throw a **TypeError** exception.
22. Return *resultDesc*.

NOTE [[GetOwnProerty] for proxy objects enforces the following invariants:

- The result of [[GetOwnProperty]] must be either an Object or **undefined**.
- A property cannot be reported as non-existent, if it exists as a non-configurable own property of the target object.
- A property cannot be reported as non-existent, if it exists as a own property of the target object and the target object is not extensible.
- A property cannot be reported as existent, if it does not exists as a own property of the target object and the target object is not extensible.
- A property cannot be reported as non-configurable, if it does not exists as a own property of the target object or if it exists as a configurable own property of the target object.
- The result of [[GetOwnProperty]] can be applied to the target object using [[DefineOwnPropery]] and will not throw an exception.

8.5.7 [[GetP]] (P, Receiver)

When the [[GetP]] internal method of proxy object *O* is called with property key *P* and ECMAScipt language value *Receiver* the following steps are taken:

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. Let *handler* the value of the [[ProxyHandler]] internal data property of *O*.
3. Let *target* the value of the [[ProxyTarget]] internal data property of *O*.
4. Let *trap* be the result of GetMethod(*handler*, "**`get`**").
5. ReturnIfAbrupt(*trap*).
6. If *trap* is **undefined**, then
    a. Return the result of calling the [[GetP]] internal method of *target* with arguments *P* and *Receiver*.
7. Let *trapResult* be the result of calling *trap* with *handler* as the **this** value and a new List containing *target*, *P*, and *Receiver*.
8. ReturnIfAbrupt(*trapResult*).
9. Let *targetDesc* be the result of calling the [[GetOwnProperty]] internal method of *target* with argument *P*.
10. ReturnIfAbrupt(*targetDesc*).
11. If *targetDesc* is not **undefined**, then
    a. If IsDataDescriptor(*targetDesc*) and *targetDesc*.[[Configurable]] is **false** and *targetDesc*.[[Writable]] is **false**, then
        i. If SameValue(*trapResult*, *targetDesc*.[[Value]]) is **false**, then throw a **TypeError** exception.
    b. If IsAccessorDescriptor(*targetDesc*) and *targetDesc*.[[Configurable]] is **false** and *targetDesc*.[[Get]] is **undefined**, then
        i. If *trapResult* is not **undefined**, then throw a **TypeError** exception.
12. Return *trapResult*.

NOTE [[GetP] for proxy objects enforces the following invariants:

- The value reported for a property must be the same as the value of the corresponding target object property if the.target object property is a non-writable, non-configurable data property.
- The value reported for a property must be **undefined** if the corresponding corresponding target object property is non-configurable accessor property that has **undefined** as its [[Get]] attribute.

8.5.8 [[SetP] (P, V, Receiver)

When the [[SetP]] internal method of proxy object *O* is called with property key *P*, value *V*, and ECMAScipt language value *Receiver*, the following steps are taken:

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. Let *handler* the value of the [[ProxyHandler]] internal data property of *O*.
3. Let *target* the value of the [[ProxyTarget]] internal data property of *O*.
4. Let *trap* be the result of GetMethod(*handler*, "**`set`**").
5. ReturnIfAbrupt(*trap*).
6. If *trap* is **undefined**, then
    a. Return the result of calling the [[SetP]] internal method of *target* with arguments *P*, *V*, and *Receiver*.
7. Let *trapResult* be the result of calling *trap* with *handler* as the **this** value and a new List containing *target*, *P*, *V*, and *Receiver*.
8. ReturnIfAbrupt(*trapResult*).
9. If ToBoolean(*trapResult*) is **false**, then return **false**.

10. Let *targetDesc* be the result of calling the [[GetOwnProperty]] internal method of *target* with argument *P*.
11. ReturnIfAbrupt(*targetDesc*).
12. If *targetDesc* is not **undefined**, then
    a. If IsDataDescriptor(*targetDesc*) and *targetDesc*.[[Configurable]] is **false** and *targetDesc*.[[Writable]] is **false**, then
       i. If SameValue(*V*, *targetDesc*.[[Value]]) is **false**, then throw a **TypeError** exception.
    b. If IsAccessorDescriptor(*targetDesc*) and *targetDesc*.[[Configurable]] is **false**, then
       i. If *targetDesc*.[[Set]] is **undefined**, then throw a **TypeError** exception.
13. Return **true**.

NOTE [[SetP] for proxy objects enforces the following invariants:

- Cannnot change the value of a property to be different from the value of the corresoponding target object property if the corresponding target object property is a non-writable, non-configurable data property.
- Cannot set the value of a property if the corresponding corresponding target object property is a non-configurable accessor property that has **undefined** as its [[Set]] attribute.

8.5.9 [[Delete]] (P)

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

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. Let *handler* the value of the [[ProxyHandler]] internal data property of *O*.
3. Let *target* the value of the [[ProxyTarget]] internal data property of *O*.
4. Let *trap* be the result of GetMethod(*handler*, "**`deleteProperty`**").
5. ReturnIfAbrupt(*trap*).
6. If *trap* is **undefined**, then
   a. Return the result of calling the [[Delete]] internal method of *target* with argument *P*.
7. Let *trapResult* be the result of calling *trap* with *handler* as the **this** value and a new List containing *target* and *P*.
8. ReturnIfAbrupt(*trapResult*).
9. If ToBoolean(*trapResult*) is **false**, then return **false**.
10. Let *targetDesc* be the result of calling the [[GetOwnProperty]] internal method of *target* with argument *P*.
11. ReturnIfAbrupt(*targetDesc*).
12. If *targetDesc* is **undefined**, then return **true**.
13. If *desc*.[[Configurable]] is **false**, then throw a **TypeError** exception.
14. Return **true**.

NOTE [[Delete]] for proxy objects enforces the following invariant:

- A property cannot be deleted, if it exists as a non-configurable own property of the target object.

8.5.10 [[DefineOwnProperty]] (P, Desc)

When the [[DefineOwnProperty]] internal method of proxy object *O* is called with property key *P* and property descriptor *Desc*, the following steps are taken:

1. Assert: *P* is a valid property key, either a String or a Symbol Object.
2. Let *handler* the value of the [[ProxyHandler]] internal data property of *O*.
3. Let *target* the value of the [[ProxyTarget]] internal data property of *O*.
4. Let *trap* be the result of GetMethod(*handler*, "**`defineProperty`**").
5. ReturnIfAbrupt(*trap*).
6. If *trap* is **undefined**, then
   a. Return the result of calling the [[DefineOwnProperty]] internal method of *target* with arguments *P* and *Desc*.
7. Let *descObj* be FromPropertyDescriptor(*Desc*).
8. NOTE If *Desc* was originally generated from an object using ToPropertyDescriptor, then *descObj* will be that original object.
9. Let *trapResult* be the result of calling *trap* with *handler* as the **this** value and a new List containing *target*, *P*, and *descObj*.
10. ReturnIfAbrupt(*trapResult*).
11. If ToBoolean(*trapResult*) is **false**, then return **false**.

12. Let *targetDesc* be the result of calling the [[GetOwnProperty]] internal method of *target* with argument *P*.
13. ReturnIfAbrupt(*targetDesc*).
14. Let *extensibleTarget* be the result of calling the [[IsExtensible]] internal method of *target*.
15. ReturnIfAbrupt(*extensibleTarget*).
16. Set *extensibleTarget* to ToBoolean(*extensibleTarget*),
17. If *targetDesc* is **undefined**, then
    a. If *extensibleTarget* is **false**, then throw a **TypeError** exception.
    b. If *Desc*.[[Configurable]] is **false**, then throw a **TypeError** exception.
18. Else *targetDesc* is not **undefined,**
    a. If IsCompatableDescriptor(*extensibleTarget*, *Desc* , *targetDesc*) is **false**, then throw a **TypeError** exception.
    b. If *Desc*.[[Configurable]] is **false** and *targetDesc*.[[Configurable]] is **true**, then throw a **TypeError** exception.
19. Return **true**.

NOTE [[GetOwnProerty] for proxy objects enforces the following invariants:

- A property cannot be added, if the target object is not extensible.
- A property cannot be added as or modified to be non-configurable, if it does not exists as a non-configurable own property of the target object.
- A property may not be non-configurable, if is corresponding configurable property of the target object exists.
- If a property has a corresponding target object property then apply the property descriptor of the property to the target object using [[DefineOwnPropery]] will not throw an exception.

8.5.11 [[Enumerate]] ()

When the [[Enumerate]] internal method of of proxy object *O* is called the following steps are taken:

1. Let *handler* the value of the [[ProxyHandler]] internal data property of *O*.
2. Let *target* the value of the [[ProxyTarget]] internal data property of *O*.
3. Let *trap* be the result of GetMethod(*handler*, "**`enumerate`**").
4. ReturnIfAbrupt(*trap*).
5. If *trap* is **undefined**, then
    a. Return the result of calling the [[Enumerate]] internal method of *target*.
6. Let *trapResult* be the result of calling *trap* with *handler* as the **this** value and a new List containing *target*.
7. ReturnIfAbrupt(*trapResult*).
8. If Type(*trapResult*) is not Object, then throw a **TypeError** exception.
9. TODO: we may need to add a lot of additional invariant checking here according to the wiki spec. But maybe it really isn't necessary
10. Return *trapResult*.

NOTE [[Enumerate] for proxy objects enforces the following invariants:

- The result of [[Enumerate]] must be an Object.

8.5.12 [[Keys]] ( )

When the [[Keys]] internal method of proxy object *O* is called the following steps are taken:

11. Let *handler* the value of the [[ProxyHandler]] internal data property of *O*.
12. Let *target* the value of the [[ProxyTarget]] internal data property of *O*.
13. Let *trap* be the result of GetMethod(*handler*, "**`keys`**").
14. ReturnIfAbrupt(*trap*).
15. If *trap* is **undefined**, then
    a. Return the result of calling the [[Keys]] internal method of *target*.
16. Let *trapResult* be the result of calling *trap* with *handler* as the **this** value and a new List containing *target*.
17. ReturnIfAbrupt(*trapResult*).
18. If Type(*trapResult*) is not Object, then throw a **TypeError** exception.
19. TODO: we may need to add a lot of additional invariant checking here according to the wiki spec. But maybe it really isn't necessary
20. Return *trapResult*.

NOTE [[Keys] for proxy objects enforces the following invariants:

- The result of [[Keys]] must be an Object.

8.5.13 [[OwnPropertyKeys]] ( )

When the [[OwnPropertyKeys]] internal method of proxy object *O* is called the following steps are taken:

1. Let *handler* the value of the [[ProxyHandler]] internal data property of *O*.
2. Let *target* the value of the [[ProxyTarget]] internal data property of *O*.
3. Let *trap* be the result of GetMethod(*handler*, "**`getOwnPropertyNames`**").
4. ReturnIfAbrupt(*trap*).
5. If *trap* is **undefined**, then
    a. Return the result of calling the [[OwnPropertyKeys]] internal method of *target*.
6. Let *trapResult* be the result of calling *trap* with *handler* as the **this** value and a new List containing *target*.
7. ReturnIfAbrupt(*trapResult*).
8. If Type(*trapResult*) is not Object, then throw a **TypeError** exception.
9. TODO: we may need to add a lot of additional invariant checking here according to the wiki spec. But maybe it really isn't necessary
10. Return *trapResult*.

NOTE [[OwnPropertyKeys] for proxy objects enforces the following invariants:

- The result of [[OwnPropertyKeys]] must be an Object.

8.5.14 [[Freeze]] ( )

When the [[Freeze]] internal method of of proxy object *O* is called the following steps are taken:

1. Return the result of MakeObjectSecure(*O*, **false**).

8.5.15 [[Seal]] ( )

When the [[Seal]] internal method of *O* is called the following steps are taken:

1. Return the result of MakeObjectSecure(*O*, **false**).

8.5.16 [[IsFrozen]] ( )

When the [[IsFrozen]] internal method of *O* is called the following steps are taken:

1. Return the result of TestIfSecureObject (*O*, **true)**.

8.5.17 [[IsSealed]] ( )

When the [[IsSealed]] internal method of *O* is called the following steps are taken:

1. Return the result of TestIfSecureObject (*O*, **false)**.

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. Other, more specialized abstract operations are defined throughout this specification.

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 13:

Table 13 — ToPrimitive Conversions

Input Type	*Result*
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return ToPrimitive(*argument*.[[value]]) also passing the optional hint *PreferredType*.
Undefined	Return *argument* (no conversion).
Null	Return *argument* (no conversion).
Boolean	Return *argument* (no conversion).
Number	Return *argument* (no conversion).
String	Return *argument* (no conversion).
Object	Perform the steps given following this table.

When the *InputType* is Object, the following steps are taken:

1. If *PreferredType* was not passed, let *hint* be "**`default`**".
2. Else if *PreferredType* is hint Sring, let *hint* be "**`string`**".
3. Else *PreferredType* is hint Number, let *hint* be "**`number`**".
4. Let *exoticToPrim* be the result of Get(*O*, @@ToPrimitive).
5. ReturnIfAbrupt(*exoticToPrim*).
6. If *exoticToPrim* is not **undefined**, then
    a. If IsCallable(*toString)* is **false**, then throw a **TypeError** exception.
    b. Let *result* be the result of calling the [[Call]] internal method of *exoticToPrim*, with *O* as *thisArgument* and a List containing *hint* as *argumentsList*.
    c. ReturnIfAbrupt(*result*).
    d. If *result* is an ECMAScript language value and Type(*resul*) is not Object, then return *result*.
    e. Else, throw a **TypeError** exception.
7. If *hint* is be "**`default`**" then, let *hint* be "**`number`**".
8. Return the result of OrdinaryToPrimitive(*O*,*hint*).

When the OrdinaryToPrimitive is called with arguments *O* and *hint*, the following steps are taken:

1. Assert: Type(*O*) is Object
2. Assert: Type(*hint*) is String and its values is either "**`string`**" or "**`number`**".
3. If *hint* is "**`string`**", then
    a. Let *tryFirst* be "**`toString`**".
    b. Let *trySecond* be "**`valueOf`**".
4. Else,
    a. Let *tryFirst* be "**`valueOf`**".
    b. Let *trySecond First* be "**`toString`**".
5. Let *first* be the result of Get(*O*, *tryFirst*).
6. ReturnIfAbrupt(*first*).
7. If IsCallable(*first)* is **true** then,
    a. Let *result* be the result of calling the [[Call]] internal method of *first*, with *O* as *thisArgument* and an empty List as *argumentsList*.
    b. ReturnIfAbrupt(*result*).
    c. If *result* is an ECMAScript language value and Type(*resul*) is not Object, then return *result*.
    d. Else, throw a **TypeError** exception.
8. Let *second* be the result of Get(*O*, *trySecond*).

9. ReturnIfAbrupt(*second*).
10. If IsCallable(*second)* is **true** then,
    a. Let *result* be the result of calling the [[Call]] internal method of *second*, with *O* as *thisArgument* and an empty argument list.
    b. ReturnIfAbrupt(*result*).
    c. If *result* is an ECMAScript language value and Type(*resul*) is not Object, then return *result*.
11. Throw a **TypeError** exception.

NOTE When ToPrimitive is called with no hint, then it generally behaves as if the hint were Number. However, objects may over-ride this behaviour by defining a @@ToPrimitve method. Of the objects defined in this specification only Date objects (see 15.9.6) over-ride the default ToPrimitive behaviour. Date objects treat no hint as if the hint were String.

9.1.2 ToBoolean

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

Table 14 — ToBoolean Conversions

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

9.1.3 ToNumber

The abstract operation ToNumber converts its *argument* to a value of type Number according to Table 15:

Table 15 — ToNumber Conversions

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

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 **:::**
StrWhiteSpace$_{opt}$
StrWhiteSpace$_{opt}$ *StrNumericLiteral* *StrWhiteSpace*$_{opt}$

StrWhiteSpace **:::**
StrWhiteSpaceChar *StrWhiteSpace*$_{opt}$

StrWhiteSpaceChar **:::**
WhiteSpace
LineTerminator

StrNumericLiteral **:::**
StrDecimalLiteral
HexIntegerLiteral

StrDecimalLiteral **:::**
StrUnsignedDecimalLiteral
+ *StrUnsignedDecimalLiteral*
- *StrUnsignedDecimalLiteral*

StrUnsignedDecimalLiteral **:::**
`Infinity`
DecimalDigits **.** *DecimalDigits*$_{opt}$ *ExponentPart*$_{opt}$
. *DecimalDigits* *ExponentPart*$_{opt}$
DecimalDigits *ExponentPart*$_{opt}$

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`

NOTE 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**.

- **`Infinity` and `-Infinity`** are recognized as a *StringNumericLiteral* but not as a *NumericLiteral*.

Runtime Semantics

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* **:::** $StrWhiteSpace_{opt}$ *StrNumericLiteral* $StrWhiteSpace_{opt}$ is the MV of *StrNumericLiteral*, no matter whether white space is present or not.
- The MV of *StrNumericLiteral* **:::** *StrDecimalLiteral* is the MV of *StrDecimalLiteral*.
- The MV of *StrNumericLiteral* **:::** *HexIntegerLiteral* is the MV of *HexIntegerLiteral*.
- The MV of *StrDecimalLiteral* **:::** *StrUnsignedDecimalLiteral* is the MV of *StrUnsignedDecimalLiteral*.
- The MV of *StrDecimalLiteral* **:::** **+** *StrUnsignedDecimalLiteral* is the MV of *StrUnsignedDecimalLiteral*.
- The MV of *StrDecimalLiteral* **:::** **-** *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 *StrUnsignedDecimalLiteral***:::** **`Infinity`** is 10^{10000} (a value so large that it will round to **+∞**).
- The MV of *StrUnsignedDecimalLiteral***:::** *DecimalDigits* **.** is the MV of *DecimalDigits*.
- The MV of *StrUnsignedDecimalLiteral***:::** *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 *StrUnsignedDecimalLiteral***:::** *DecimalDigits* **.** *ExponentPart* is the MV of *DecimalDigits* times 10^{e}, where *e* is the MV of *ExponentPart*.
- The MV of *StrUnsignedDecimalLiteral***:::** *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 *StrUnsignedDecimalLiteral***:::** **.** *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 sign(*number*) × floor(abs(*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 *number* be the result of calling ToNumber on the input argument.
2. ReturnIfAbrupt(*number*).
3. If *number* is **NaN**, **+0**, **–0**, **+∞**, or **–∞**, return **+0**.
4. Let *int* be sign(*number*) × floor(abs(*number*)).
5. Let *int32bit* be *int* 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 *int* and *k* is mathematically an integer multiple of 2^{32}.
6. If *int32bit* is greater than or equal to 2^{31}, return *int32bit* – 2^{32}, otherwise return *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(ToUint32(*x*)) is equal to ToInt32(*x*) for all values of *x*. (It is to preserve this latter property that +∞ and –∞ 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 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**, +0, −0, +∞, or −∞, return **+0**.
4. Let *int* be sign(*number*) × floor(abs(*number*)).
5. Let *int32bit* be *int* 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 *int* and *k* is mathematically an integer multiple of 2^{32}.
6. Return *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(ToInt32(*x*)) is equal to ToUint32(*x*) for all values of *x*. (It is to preserve this latter property that **+∞** and **–∞** 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 *number* be the result of calling ToNumber on the input argument.
2. ReturnIfAbrupt(*number*).
3. If *number* is **NaN**, +0, −0, +∞, or −∞, return **+0**.
4. Let *int* be sign(*number*) × floor(abs(*number*)).
5. Let *int16bit* be *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 *int* and *k* is mathematically an integer multiple of 2^{16}.
6. Return *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 16:

Table 16 — ToString Conversions

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

9.1.8.1 ToString Applied to the Number Type

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

1. If *m* is **NaN**, return the String **"NaN"**.
2. If *m* is **+0** or **–0**, return the String **"0"**.
3. If *m* is less than zero, return the String concatenation of the String **"-"** and ToString(–*m*).
4. If *m* is infinity, return the String **"Infinity"**.
5. 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.
6. 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**'.
7. 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*.
8. 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*.
9. 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).
10. 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:

> 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 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 17:

Table 17 — ToObject Conversions

Argument Type	*Result*
Completion Record	If *argument* is an abrupt completion, return *argument*. Otherwise return ToObject(*argument*.[[value]])
Undefined	Throw a **TypeError** exception.
Null	Throw a **TypeError** exception.
Boolean	Return a new Boolean object whose [[BooleanData]] internal data property is set to the value of *argument*. See 15.6 for a description of Boolean objects.
Number	Return a new Number object whose [[NumberData]] internal data property is set to the value of *argument*. See 15.7 for a description of Number objects.
String	Return a new String object whose [[StringData]] internal data property is set to the value of *argument*. See 15.5 for a description of String objects.
Object	Return *argument* (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* is an exotic String object, 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 18:

Table 18 — CheckObjectCoercible Results

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

9.2.2 IsCallable

The abstract operation IsCallable determines if its *argument*, which must be an ECMAScript language value or a Completion Record, is a callable function Object according to Table 19:

Table 19 — IsCallable Results

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

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
    a. If *x* and *y* are exactly the same sequence of characters (same length and same characters in corresponding positions) return **true**; otherwise, return **false**.
8. If Type(*x*) is Boolean, then
    a. If *x* and *y* are both **true** or both **false**, then return **true**; otherwise, return **false**.
9. Return **true** if *x* and *y* are the same Object value. Otherwise, return **false**.

9.2.4 IsConstructor

The abstract operation IsConstructor determines if its *argument*, which must be an ECMAScript language value or a Completion Record, is a function object with a [[Construct]] internal method.

1. ReturnIfAbrupt(*argument*).
2. If Type(*argument*) is not Object, return **false**.
3. If *argument* has a [[Construct]] internal method, return **true**.
4. Return **false**.

9.3 Operations on Objects

9.3.1 Get (O, P)

The abstract operation Get is used to retrieve the value of an specific property of an object. The operation is called with arguments *O* and *P* where *O* is the object and *P* is the property key. This abstract operation performs, the following steps:

1. Asset: Type(*O*) is Object.
2. Assert: Either Type(*P*) is String or Type(*O*) is Object and *O* is a Symbol object.
3. Return the result of calling the [[GetP]] internal method of *O* passing *P* and *O* as the arguments.

9.3.2 Put (O, P, V, Throw)

The abstract operation Put is used to set the value of an specific property of an object. The operation is called with arguments *O*, *P*, *V*, and *Throw* where *O* is the object, *P* is the property key, *V* is the new value for the property and *Throw* is a Boolean flag. This abstract operation performs, the following steps:

1. Asset: Type(*O*) is Object.
2. Assert: Either Type(*P*) is String or Type(*O*) is Object and *O* is a Symbol object.
3. Asset: Type(*Throw*) is Boolean.
4. Let *success* be the result of calling the [[SetP]] internal method of *O* passing *P*, *V*, and *O* as the arguments.
5. ReturnIfAbrupt(*success*).
6. If *success* is **false** and *Throw* is **true**, then throw a **TypeError** exception.
7. Return *success*.

9.3.3 CreateOwnDataProperty (O, P, V)

The abstract operation CreateOwnProperty is used to create a new own property of an object. The operation is called with arguments *O*, *P*, and *V* where *O* is the object, *P* is the property key, and *V* is the new value for the property. This abstract operation performs, the following steps:

1. Asset: Type(*O*) is Object.
2. Assert: Either Type(*P*) is String or Type(*O*) is Object and *O* is a Symbol object.
3. Asset: *O* does not have an own property whose key is *P*..
4. Let *extensible* be the result of calling the [[GetExtensible]] internal method of *O*.
5. ReturnIfAbrupt(*extensible*).
6. If *extensible* is **false**, then return **false**.
7. Let *newDesc* be the Property Descriptor.
   {[[Value]]: *V*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
8. Return the result of calling the [[DefineOwnProperty]] internal method of *O* passing *P* and *newDesc* as arguments.

9.3.4 DefinePropertyOrThrow (O, P, desc)

The abstract operation DefinePropertyOrThrow is used to call the [[DefineOwnProperlty]] internal method of an object in a manner that will throw a TypeError exception if the requested property update can not be performed. The operation is called with arguments *O*, *P*, and *desc* where *O* is the object, *P* is the property key, and *desc* is Property Descriptor record for the property. This abstract operation performs, the following steps:

1. Asset: Type(*O*) is Object.
2. Assert: Either Type(*P*) is String or Type(*O*) is Object and *O* is a Symbol object.
3. Let *success* be the result of calling the [[DefineOwnProperty]] internal method of *O* passing *P* and *desc* as arguments.
4. ReturnIfAbrupt(*success*).
5. If *success* is **false**, then throw a **TypeError** exception.
6. Return *success*.

9.3.5 DeletePropertyOrThrow (O, P)

The abstract operation Put is used to remove a specific own property of an object. It throws an exception is the property is not configurable. The operation is called with arguments *O* and *P* where *O* is the object and *P* is the property key. This abstract operation performs, the following steps:

1. Asset: Type(*O*) is Object.
2. Assert: Either Type(*P*) is String or Type(*O*) is Object and *O* is a Symbol object.
3. Let *success* be the result of calling the [[Delete]] internal method of *O* passing *P* as the argument.
4. ReturnIfAbrupt(*success*).

5. If *success* is **false**, then throw a **TypeError** exception.
6. Return *success*.

9.3.6 HasProperty (O, P)

The abstract operation HasProperty is used to determine whether an object has a property with the specified property key. The property may be either an own or inherited. A Boolean value is return. The operation is called with arguments *O* and *P* where *O* is the object and *P* is the property key. This abstract operation performs, the following steps:

1. Asset: Type(*O*) is Object.
2. Assert: Either Type(*P*) is String or Type(*O*) is Object and *O* is a Symbol object.
3. Let *obj* be *O*.
4. Repeat,
    a. If *obj* is **null**, then return **false.**
    b. If Type(*obj*) is not Object, then throw a **TypeError** exception.
    c. Let *has* be the result of calling the [[HasOwnProperty]] internal method of *obj* argument *P*.
    d. ReturnIfAbrupt(*has*).
    e. If *has* is **true**, return *has*.
    f. Set *obj* to the result of calling the [[GetInheritance]] internal method of *obj*.
    g. ReturnIfAbrupt(*obj*).

9.3.7 GetMethod (O, P)

The abstract operation GetMethod is used to get the value of an specific property of an object when the value of the property is expected to be a function. The operation is called with arguments *O* and *P* where *O* is the object, *P* is the property key. This abstract operation performs, the following steps:

1. Asset: Type(*O*) is Object.
2. Assert: Either Type(*P*) is String or Type(*O*) is Object and *O* is a Symbol object.
3. Let *func* be the result of calling the [[GetP]] internal method of *O* passing *P* and *V* as the arguments.
4. ReturnIfAbrupt(*func*).
5. If *func* is **undefined**, then return **undefined**.
6. If IsCallable(*func*) is **false**, then throw a **TypeError** exception.
7. Return *func*.

9.3.8 Invoke(O,P [,args])

The abstract operation Invoke is used to call a method property of an object. The operation is called with arguments *P*, *O*, and optionally *args* where *P* is the property key, *O* serves as both the lookup point for the property and the **this** value of the call, and *args* is the list of arguments values passed to the method. If args is not present, an empty List is used as its value. This abstract operation performs, the following steps:

1. Assert: *P* is a valid property key.
2. If *args* was not passed, then let *args* be a new empty List.
3. Let *obj* be ToObject(*O*).
4. ReturnIfAbrupt(*obj*).
5. Let *func* be the result of GetMethod(*obj*, *P*).
6. ReturnIfAbrupt(*func*).
7. If *func* is **undefined**, throw a **TypeError** exception.
8. Return the result of calling the [[Call]] internal method of *func* passing *O* as *thisArgument* and *args* as *argumentsList*.

9.3.9 MakeObjectSecure (O, immutable)

The abstract operation MakeObjectSecure is used to fix the set of own properties of an object. If the Boolean argument *immutable* is **true** all own data properties are also made non-writable. This abstract operation performs, the following steps:

1. Asset: Type(*O*) is Object.
2. Assert: Type(*immutable*) is Boolean.

3. Let *keys* be the result of calling [[OwnPropertyKeys]] internl method of *O*.
4. ReturnIfAbrupt(*keys*).
5. Let *pendingException* be **undefined**.
6. If *immutable* is **false**, then
    a. Repeat for each element *k* of *keys*,
        i. Let *status* be the result of DefinePropertyOrThrow(*O*, *k*, PropertyDescriptor{ [[Configurable]]: **false**}).
        ii. If *status* is an Abrupt Completion, then
            1. If *pendingException* is **undefined**, then set *pendingException* to *status*.
7. Else,
    a. Repeat for each element *k* of *keys*,
        i. Let *status* be the result of calling the [[GetOwnProperty]] internal method of *O* with *k*.
        ii. If *status* is an Abrupt Completion, then
            1. If *pendingException* is **undefined**, then set *pendingException* to *status*.
        iii. Else,
            1. Let *currentDesc* be *status*.[[value]].
            2. If *currentDesc* is not **undefined**, then
                a. If IsAccessorDescriptor(*currentDesc*) is **true**, then
                    i. Let *desc* be PropertyDescriptor{[[Configurable]]: **false**}.
                b. Else,
                    i. Let *desc* be PropertyDescriptor [[Configurable]]: **false**, [[Writable]]: **false** }.
                c. Let *status* be the result of DefinePropertyOrThrow(*O*, *k*, *desc*).
                d. If *status* is an Abrupt Completion, then
                    i. If *pendingException* is **undefined**, then set *pendingException* to *status*.
8. If *pendingException* is not **undefined**, then return *pendingException*.
9. Return the result of calling the[[PreventExtensions]] internal method of *O*.

9.3.10 TestIfSecureObject (O, immutable)

The abstract operation TestIfSecureObject is used to determine the set of own properties of an object are fixed. If the Boolean argument *immutable* is **true** a check is also made to determine whether all own data properties are non-writable. This abstract operation performs, the following steps:

1. Asset: Type(*O*) is Object.
2. Assert: Type(*immutable*) is Boolean.
3. Let *status* be the result of calling the [[GetExtensible]] internal method of *O*.
4. ReturnIfAbrupt(*status*).
5. If *status* is **true**, then return **false**
6. NOTE If the object is extensible, none of its properties are examined.
7. Let *keys* be the result of calling [[OwnPropertyKeys]] internl method of *O*.
8. ReturnIfAbrupt(*keys*).
9. Let *pendingException* be **undefined**.
10. If *immutable* is **false**, then
11. Let *configurable* be **false**.
12. Let *writable* be **false**.
13. Repeat for each element *k* of *keys*,
    a. Let *status* be the result of calling the [[GetOwnProperty]] internal method of *O* with *k*.
    b. If *status* is an Abrupt Completion, then
        i. If *pendingException* is **undefined**, then set *pendingException* to *status*.
        ii. Let *configurable* be **true**.
    c. Else,
        i. Let *currentDesc* be *status*.[[value]].
        ii. If *currentDesc* is not **undefined**, then
            1. Set *configurable* to *configurable* logically ored with *currentDesc*.[[Configurable]].
            2. If IsDataDescriptor(*currentDesc*) is **true**, then
                a. Set *writable* to *writable* logically ored with *currentDesc*.[[Writable]].
14. If *pendingException* is not **undefined**, then return *pendingException*.

15. If *immutable* is **true** and *writable* is **true**, then return **false**.
16. If *configurable* is **true**, then return **false**.
17. Return **true**.

9.3.11 CreateArrayFromList (elements)

The abstract operation CreateArrayFromList is used to create an Array object whose elements are provided by an internal List. This abstract operation performs, the following steps:

1. Assert: *elements* is a List whose elements are all ECMAScript language values.
2. Let *array* be the result of the abstract operation ArrayCreate with argument 0.
3. Let *n* be 0.
4. For element *e* of *elements*
    a. Call CreateOwnDataProperty(*array*, ToString(*n*), *e*).
    b. Asset: the call in step 4.a will never result in an abrupt completion.
    c. Increment *n* by 1.
5. Return *array*.

9.3.12 OrdinaryHasInstance (C, O)

The abstract operation OrdinaryHasInstance implements the default algorithm for determining if an object *O* inherits from the inheritance path used by constructor *C*. This abstract operation performs, the following steps:

1. If IsCallable(*C*) is **false**, return **false**.
2. If *C* has a [[BoundTargetFunction]] internal data property
    a. Return the result of the instanceOfOperator abstract operation (11.8) with *C* and *O* as arguments.
3. If Type(*O*) is not an Object, return **false**.
4. Let *P* be the result of Get(*C*, **"prototype"**).
5. ReturnIfAbrupt(*P*).
6. If Type(*P*) is not Object, throw a **TypeError** exception.
7. Repeat
    a. Set *O* to the result of calling the [[GetInheritance]] internal method of *O* with no arguments.
    b. ReturnIfAbrupt(*O*).
    c. If *O* is **null**, return **false**.
    d. If SameValue(*P*, *O*) is **true**, return **true**.

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 *Script*. The global code of a particular *Script* does not include any source text that is parsed as part of a *FunctionBody*, *ConciseBody*, *ClassBody*, or *ModuleBody*.

- *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 *Script*. The eval code for a particular invocation of **eval** is the global code portion of that *Script*.

    - *Function code* is source text that is parsed to supply the value of the [[Code]] internal data property (see 13.6) 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 *FunctionBody*, *ConciseBody*, or *ClassBody*..

    - *Generator code* is source text that is parsed to supply the value of the [[Code]] internal data property (see 13.5) of generator objects. The *generator code* of a particular generator does not include any source text that is parsed as the function code of a nested *FunctionBody*, *ConciseBody*, or *ClassBody*. All generator code is also considered to

be function code, but only function code that is defined within a generator is generator code.

- *Module code* is source text that is parse code that is provided as a *ModuleBody*. It is the code that is directly evaluated when a module is initialized. The module code of a particular module does not include any source text that is parsed as part of a nested *FunctionBody*, *ConciseBody*, *ClassBody*, or *ModuleBody*..

NOTE Function code is generally provided as the bodies of Function Definitions (13.1), Arrow Function Defindditions (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). Generator code is provided as the bodies of Generator Definitions 13.4 and Generator Expressions (11.????).

10.1.1 Strict Mode Code

An ECMAScript *Script* 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).
- Module code is always strict code.
- 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.

10.1.2 Non-ECMAScript Functions

An ECMAScript implementation may support the evaluation of function objects whose evaluative behaviour is expressed in some implementation defined form of executable code other than via ECMAScript code. Whether a function object is an ECMAScript code function or a non-ECMAScript function is not semantically observable from the perspective of an ECMAScript code function that calls or is called by such a non-ECMAScript function.

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 evaluation 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`** binding. As 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 **`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 primary 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 *WithStatement* that associate identifier bindings with the properties of some object. Global Environment Records and Function Environment Records are specializations that are used for specifically for *Script* global declarations and for top-level declarations within funtions.

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 three concrete subclasses, declarative environment record, object environment record, and global environment record. Function environment records are a subclass of declarative environment record. The abstract class includes the abstract specification methods defined in Table 20. These abstract methods have distinct concrete algorithms for each of the concrete subclasses.

Table 20 — Abstract Methods of Environment Records

Method	*Purpose*
HasBinding(N)	Determine if an environment record has a binding for an identifier. Return **true** if it does and **false** if it does not. The String value *N* is the text of the identifier.
CreateMutableBinding(N, D)	Create a new but uninitialised mutable binding in an environment record. The String value *N* is the text of the bound name. If the optional Boolean argument *D* is **true** the binding is may be subsequently deleted.
CreateImmutableBinding(N)	Create a new but uninitialised immutable binding in an environment record. The String value N 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 N is the text of the bound name. V is the value for the binding and is a value of any ECMAScript language type.
SetMutableBinding(N,V, S)	Set the value of an already existing mutable binding in an environment record. The String value *N* is the text of the bound name. *V* is the value for the binding and may be a value of any ECMAScript language type. *S* is a Boolean flag. If *S* is **true** and the binding cannot be set throw a **TypeError** exception. *S* 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 *N* is the text of the bound name. *S* is used to identify strict mode references. If *S* is **true** and the binding does not exist or is uninitialised throw a **ReferenceError** exception.
DeleteBinding(N)	Delete a binding from an environment record. The String value *N* is the text of the bound name If a binding for *N* exists, remove the binding and return **true**. If the binding exists but cannot be removed return **false**. If the binding does not exist return **true**.
HasThisBinding()	Determine if an environment record establishes a **`this`** binding. Return **true** if it does and **false** if it does not.
HasSuperBinding()	Determine if an environment record establishes a **`super`** method binding. Return **true** if it does and **false** if it does not.
WithBaseObject ()	If this environment record is associated with a **`with`** statement, return the with object. Otherwise, return **undefined**.

10.2.1.1 Declarative Environment Records

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

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 uninitialised. 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.
4. Return NormalCompletion(empty)

10.2.1.1.3 CreateImmutableBinding (N)

The concrete Environment Record method CreateImmutableBinding for declarative environment records creates a new immutable binding for the name *N* that is uninitialised. 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.

10.2.1.1.4 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.

10.2.1.1.5 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. If the binding for *N* in *envRec* is a mutable binding, change its bound value to *V*.
4. Else if binding for *N* in *envRec* has not yet been initialized throw a ReferenceError exception.
5. Else this must be an attempt to change the value of an immutable binding so if *S* is **true** throw a **TypeError** exception.
6. Return NormalCompletion(empty).

10.2.1.1.6 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
    a. If *S* is **false**, return the value **undefined**, otherwise throw a **ReferenceError** exception.

4. Else,
    a. Return the value currently bound to *N* in *envRec*.

10.2.1.1.7 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* cannot be deleted, return **false**.
4. Remove the binding for *N* from *envRec*.
5. Return **true**.

10.2.1.1.8 HasThisBinding ()

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

1. Return **false**.

10.2.1.1.9 HasSuperBinding ()

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

1. Return **false**.

10.2.1.1.10 WithBaseObject()

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

1. Return **undefined**.

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 HasProperty(*bindings N*).

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 HasProperty(*bindings*, *N*), is **false**.
4. If *D* is **true** then let *configValue* be **true** otherwise let *configValue* be **false**.
5. Return the result of DefinePropertyOrThrow(*bindings*, *N*, Property Descriptor {[[Value]]:**undefined**, [[Writable]]: **true**, [[Enumerable]]: **true** , [[Configurable]]: *configValue*}).

10.2.1.2.3 CreateImmutableBinding (N)

The concrete Environment Record method CreateImmutableBinding is never used within this specification in association with Object environment records.

10.2.1.2.4 InitializeBinding (N,V)

The concrete Environment Record method InitializeBinding for object 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 object environment record for which the method was invoked.
2. Assert: *envRec* must have an uninitialised binding for N.
3. Record that the binding for *N* in *envRec* has been initialised.
4. Call the SetMutableBinding concrete method of *envRec* with *N*, *V*, and **false** as arguments.

10.2.1.2.5 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* normally already exists 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. Return the result of Put(*bindings*, *N*, *V*, and *S*).

10.2.1.2.6 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 HasProperty(*bindings*, *N*).
4. ReturnIfAbrupt(*value*).
5. If *value* is **false**, then
    a. If *S* is **false**, return the value **undefined**, otherwise throw a **ReferenceError** exception.
6. Return the result of Get(*bindings*, *N*).

10.2.1.2.7 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* as the argument.

10.2.1.2.8 HasThisBinding ()

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

1. Return **false**.

10.2.1.2.9 HasSuperBinding ()

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

1. Return **false**.

10.2.1.2.10 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**.

10.2.1.3 Function Environment Records

A function 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 function environment record contains the **`this`** value used within its scope. If such a function references **`super`**, its function environment record also contains the state that is used to perform **`super`** method invocations from within the function.

Function 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 20 and share the same specifications for all of those methods except for HasThisBinding and HasSuperBinding. In addition, declarative environment records support the methods listed in Table 21:

Table 21 — Additional Methods of Function Environment Records

Method	***Purpose***
GetThisBinding()	Return the value of this environment record's **`this`** binding.
GetSuperBase()	Return the object that is the base for **`super`** property accesses bound in this environment record. The object is derived from this environment record's HomeObject binding. If the value is Empty, return **undefined**.
GetMethodName()	Return the value of this environment record's MethodName binding.

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

10.2.1.3.1 HasThisBinding ()

Function 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 GetSuperBase ()

1. Let *home* be the value of this environment record's *HomeObject*.
2. If *home* has the value Empty, then return **undefined**.
3. Assert Type(*home*) is Object.
4. Return the result of calling *home's* [[GetInheritance]] internal method..

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 record is used to represent the outer most scope that is shared by all of the ECMAScript *Script* elements that are processed in a common Realm (10.3). 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 *Script* productions.

A global environment record is logically a single record but it is specified as a composite encapsulating an object environment record and a declarative environment record. The object environment record has as its base object the global object of the associated Realm. This global object is also the value of the global environment record's *thisValue*. The object environment record component of a global environment record contains the bindings for all built-in globals (15.1) and all bindings introduced by a *FunctionDeclaration* or *VariableStatement* contained in global code. The bindings for all other ECMAScript declarations in global code are contained in the declarative environment record component of the global environment record.

Properties may be created directly on a global object. Hence, the object environment record component of a global environment record may contain both bindings created explicitly by *FunctionDeclaration* or *VariableStatement* declarations and binding created implicitly as properties of the global object. In order to identify which bindings were explicitly created using declarations, a global environment record maintains a list of the names bound using its CreateGlobalVarBindings and CreateGlobalFunctionBindings concrete methods.

Global environment records have the additional state components listed in Table 22 and the additional methods listed in Table 23.

Table 22 -- Components of Global Environment Records

Component	*Purpose*
ObjectEnvirornment	A Object Environment Record whose base object is the global object. Contains global built-in bindings as well as bindings for *FunctionDeclaration* or *VariableStatement* declarations in global code for the associated Realm.
DeclarativeEnvironment	A Declarative Environment Record that contains bindings for all declarations in global for the associated Realm code except for *FunctionDeclaration* and *VariableStatement* declarations.
VarNames	A List containing the string names bound by *FunctionDeclaration* or *VariableStatement* declarations in global code for the associated Realm.

Table 23 — Additional Methods of Global Environment Records

Method	*Purpose*
GetThisBinding()	Return the value of this environment record's **`this`** binding.
HasVarDeclaration (N)	Determines if the argument identifier has a binding in this environment record that was created using a *VariableStatement* or a *FunctionDeclaration*.
HasLexicalDeclaration (N)	Determines if the argument identifier has a binding in this environment record that was created using a lexical declaration such as a *LexicalDeclaration* or a *ClassDeclaration*:
CanDeclareGlobalVar (N)	Determines if a corresponding CreateGlobalVarBinding call would succeed if called for the same argument *N*.
CanDeclareGlobalFunction (N)	Determines if a corresponding CreateGlobalFunctionBinding call would succeed if called for the same argument *N*.
CreateGlobalVarBinding(N, D)	Used to create global **`var`** bindings in the ObjectEnvironmentComponent of the environment record. The binding will be a mutable binding. The corresponding global object property will have attribute values approate for a **`var`**. The String value *N* is the text of the bound name. *V* is the initial value of the binding If the optional Boolean argument *D* is **true** the binding is may be subsequently deleted. This is logically equivalent to CreateMutableBinding but it allows var declarations to receive special treatment.
CreateGlobalFunctionBinding(N, V, D)	Used to create and initialize global **`function`** bindings in the ObjectEnvironmentComponent of the environment record. The binding will be a mutable binding. The corresponding global object property will have attribute values approate for a **`function`**.The String value *N* is the text of the bound name. If the optional Boolean argument *D* is **true** the binding is may be subsequently deleted. This is logically equivalent to CreateMutableBinding followed by a SetMutableBinding but it allows function declarations to receive special treatment.

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

10.2.1.4.1 HasBinding(N)

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

1. Let *envRec* be the global environment record for which the method was invoked.
2. Let *DclRec* be *envRec's* DeclarativeEnvironment.
3. If the result of calling *DclRec's* HasBinding concrete method with argument *N* is **true**, return **true**.
4. Let *ObjRec* be *envRec's* ObjectEnvironment.
5. Return the result of calling *ObjRec's* HasBinding concrete method with argument *N*.

10.2.1.4.2 CreateMutableBinding (N, D)

The concrete environment record method CreateMutableBinding for global environment records creates a new mutable binding for the name *N* that is uninitialised. The binding is created in the associated DeclarativeEnvironment. A binding for *N* must not already exist in the DeclarativeEnvironment. 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 global environment record for which the method was invoked.
2. Let *DclRec* be *envRec's* DeclarativeEnvironment.
3. Assert: *DclRec* does not already have a binding for *N*.
4. Create a mutable binding in *DclRec* 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.
5. Return NormalCompletion(empty)

10.2.1.4.3 CreateImmutableBinding (N)

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

1. Let *envRec* be the global 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.

10.2.1.4.4 InitializeBinding (N,V)

The concrete Environment Record method InitializeBinding for global 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 global environment record for which the method was invoked.
2. Let *DclRec* be *envRec's* DeclarativeEnvironment.
3. If the result of calling *DclRec's* HasBinding concrete method with argument *N* is **true**, then
    a. Return the result of calling *DclRec's* InitializeBinding concrete method with arguments *N* and *V*.
4. Let *ObjRec* be *envRec's* ObjectEnvironment.
5. If the result of calling *ObjRec's* HasBinding concrete method with argument *N* is **true**, then
    a. Set the bound value for *N* in *envRec* to *V*.
    b. Record that the binding for *N* in *envRec* has been initialised.

10.2.1.4.5 SetMutableBinding (N,V,S)

The concrete Environment Record method SetMutableBinding for global 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*. If the binding is an immutable binding, a **TypeError** is thrown if *S* is **true**. A property named *N* normally already exists 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 declarative environment record for which the method was invoked.
2. Let *DclRec* be *envRec's* DeclarativeEnvironment.
3. If the result of calling *DclRec's* HasBinding concrete method with argument *N* is **true**, then
    a. Return the result of calling the SetMutableBinding concrete method of *DclRec* with arguments *N*, *V*, and *S*.
4. Let *ObjRec* be *envRec's* ObjectEnvironment.
5. Return the result of calling the SetMutableBinding concrete method of *ObjRec* with arguments *N*, *V*, and *S*.

10.2.1.4.6 GetBindingValue(N,S)

The concrete Environment Record method GetBindingValue for global environment records simply returns the value of its bound identifier whose name is the value of the argument *N*. If *S* is **true** and the binding is an uninitialised binding throw a **ReferenceError** exception. A property named *N* normally already exists 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 declarative environment record for which the method was invoked.
2. Let *DclRec* be *envRec's* DeclarativeEnvironment.
3. If the result of calling *DclRec's* HasBinding concrete method with argument *N* is **true**, then
    a. Return the result of calling the GetBindingValue concrete method of *DclRec* with arguments *N*, and *S*.
4. Let *ObjRec* be *envRec's* ObjectEnvironment.
5. Return the result of calling the GetBindingValue concrete method of *ObjRec* with arguments *N*, and *S*.

10.2.1.4.7 DeleteBinding (N)

The concrete Environment Record method DeleteBinding for global 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. Let *DclRec* be *envRec's* DeclarativeEnvironment.
3. If the result of calling *DclRec's* HasBinding concrete method with argument *N* is **true**, then
    a. Return the result of calling the DeleteBinding concrete method of *DclRec* with argument *N*.
4. Let *ObjRec* be *envRec's* ObjectEnvironment.
5. If the result of calling *ObjRec's* HasBinding concrete method with argument *N* is **true**, then
    a. Let *status* be the result of calling the DeleteBinding concrete method of *DclRec* with argument *N*.
    b. ReturnIfAbrupt(*status*).
    c. If *status* is **true**, then
        i. Let *varNames* be *envRec's* VarNames List.
        ii. If *N* is an element of *varNames*, then remove that element from the *varNames*.
    d. Return *status*.
6. Return **true**.

10.2.1.4.8 HasThisBinding ()

Global Environment Records always provide a **`this`** binding whose value is the associated global object.

1. Return **true**.

10.2.1.4.9 HasSuperBinding ()

1. Return **false**.

10.2.1.4.10 WithBaseObject()

Global Environment Records always return **undefined** as their WithBaseObject.

1. Return **undefined**.

10.2.1.4.11 GetThisBinding ()

2. Let *envRec* be the global environment record for which the method was invoked.
3. Let *ObjRec* be *envRec's* ObjectEnvironment.
4. Let *bindings* be the binding object for *ObjRec*.
5. Return *bindings*.

10.2.1.4.12 HasVarDeclaration (N)

The concrete environment record method HasVarDeclaration for global environment records determines if the argument identifier has a binding in this record that was created using a *VariableStatement* or a *FunctionDeclaration*:

1. Let *envRec* be the global environment record for which the method was invoked.
2. Let *varDeclaredNames* be *envRec's* VarNames List.
3. If *varDeclaredNames* contains the value of *N*, return **true**.
4. Return **false**.

10.2.1.4.13 HasLexicalDeclaration (N)

The concrete environment record method HasLexicalDeclaration for global environment records determines if the argument identifier has a binding in this record that was created using a lexical declaration such as a *LexicalDeclaration* or a *ClassDeclaration*:

1. Let *envRec* be the global environment record for which the method was invoked.
2. Let *DclRec* be *envRec's* DeclarativeEnvironment.
3. Return the result of calling *DclRec's* HasBinding concrete method with argument *N*.

10.2.1.4.14 CanDeclareGlobalVar (N)

The concrete environment record method CanDeclareGlobalVar for global environment records determines if a corresponding CreateGlobalVarBinding call would succeed if called for the same argument *N*. Redundent var declarations and var declarations for pre-existing global object properties are allowed.

1. Let *envRec* be the global environment record for which the method was invoked.
2. Let *ObjRec* be *envRec's* ObjectEnvironment.
3. If the result of calling *ObjRec's* HasBinding concrete method with argument *N* is **true**, return **true**.
4. Let *bindings* be the binding object for *ObjRec*.
5. If the result of calling the [[GetExtensible]] internal method of *bindings* is **true**, return **true**.
6. Return **false**.

10.2.1.4.15 CanDeclareGlobalFunction (N)

The concrete environment record method CanDeclareGlobalVar for global environment records determines if a corresponding CreateGlobalFunctionBinding call would succeed if called for the same argument *N*.

1. Let *envRec* be the global environment record for which the method was invoked.
2. Let *ObjRec* be *envRec's* ObjectEnvironment.
3. Let *globalObject* be the binding object for *ObjRec*.
4. Let *extensible* be the result of calling the [[GetExtensible]] internal method of *globalObject*.
5. If the result of calling *ObjRec's* HasBinding concrete method with argument *N* is **false**, then return *extensible*.
6. Let *existingProp* be the result of calling the [[GetOwnProperty]] internal method of *globalObject* with argument *N*.
7. If *existingProp* is **undefined**, then return *extensible*.
8. If *existingProp*.[[Configurable]] is **true**, then return **true**.
9. If IsDataDescriptor(*existingProp*) is **true** and *existingProp* has attribute values {[[Writable]]: **true**, [[Enumerable]]: **true**}, then return **true**.
10. Return **false**.

10.2.1.4.16 CreateGlobalVarBinding (N, D)

The concrete Environment Record method CreateVarBinding for global environment records creates a mutable binding in the associated object environment record and records the bound name in the associated VarNames List. If a binding already exists, it is reused.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Let *ObjRec* be *envRec's* ObjectEnvironment.
3. Assert: The result of calling *envRec's* CanDeclareGlobalVar concrete method with argument *N* is **true**.
4. If the result of calling *ObjRec's* HasBinding concrete method with argument *N* is **false**, then
    a. Call the CreateMutableBinding concrete method of *ObjRec* with arguments *N* and *D*.
5. Let *varDeclaredNames* be *envRec's* VarNames List.
6. If *varDeclaredNames* does not contain the value of *N*, then
    a. Append *N* to *varDeclaredNames*.
7. Return.

10.2.1.4.17 CreateGlobalFunctionBinding (N, V, D)

The concrete Environment Record method CreateFunctionBinding for global environment records creates a mutable binding in the associated object environment record and records the bound name in the associated VarNames List. If a binding already exists, it is replaced.

1. Let *envRec* be the declarative environment record for which the method was invoked.
2. Let *ObjRec* be *envRec's* ObjectEnvironment.
3. Assert: The result of calling *envRec's* CanDeclareGlobalFunction concrete method with argument *N* is **true**.
4. Let *globalObject* be the binding object for *ObjRec*.
5. Let *existingProp* be the result of calling the [[GetOwnProperty]] internal method of *globalObject* with argument *N*.
6. If *existingProp* is **undefined** or *existingProp*.[[Configurable]] is **true**, then
    a. Call the [[DefineOwnProperty]] internal method of *globalObject* passing *N* and Property Descriptor {[[Value]]:***V***, [[Writable]]: **true**, [[Enumerable]]: **true** , [[Configurable]]: *D*} as arguments.
7. Else,
    a. Call the [[DefineOwnProperty]] internal method of *globalObject* passing *N* and Property Descriptor {[[Value]]:***V*** } as arguments.
8. NOTE The assertion in step 3 means that the above [[DefineOwnProperty]] calls will never return an abrupt completion.
9. Let *varDeclaredNames* be *envRec's* VarNames List.
10. If *varDeclaredNames* does not contain the value of *N*, then
    a. Append *N* to *varDeclaredNames*.
11. Return.

NOTE Global unction declarations are always represented as a own property of the global object. If possible, an existing own property is reconfigured to have a standard set of attribute values.

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:

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 reference 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 reference 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 NewFunctionEnvironment (F, T)

When the abstract operation NewFunctionEnvironment 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 Function environment record containing containing no bindings.
3. Set *envRec's thisValue* to *T*.
4. If *F* has a [[HomeObject]] internal data property, then
    a. Set *envRec's HomeObject* to the value of *F's* [[HomeObject]] internal data property.
    b. Set *envRec's MethodName* to the value of *F's* [[MethodName]] internal data property.
5. Else,
    a. Set *envRec's HomeObject* 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 data 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 and resources.

A Realm is specified as a Record with the fields specified in Table 24:

Table 24 — Realm Record Fields

Field Name	*Value*	*Meaning*
[[intrinsics]]	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
[[globalThis]]	An ECMAScript object	The global object for this Realm
[[globalEnv]]	A ECMAScript environment	The global environment for this Realm
[[loader]]	any ECMAScript identifier or **empty**	The Loader object that can associate ECMAScript code with this Realm

The intrinsic objects associated with a code Realm are specified by Table 25. Within this specification a reference such as %name% means the value with this intrinsic name in the [[intinsics]] record of the Realm of the running execution context.

Table 25 — Intrinsic Objects with Realm Specific Bindings

Intrinsic Name	*ECMAScript Language Association*
%Object%	The initial value of the global object property named **"Object"**.
%ObjectPrototype%	The initial value of the **"prototype"** data property of the intrinsic %Object%.
%ObjProto_toString%	The initial value of the **"toString"** data property of the intrinsic %ObjectPrototype%.
%Function%	The initial value of the global object property named **"Function"**.
%FunctionPrototype%	The initial value of the **"prototype"** data property of the intrinsic %Function%.
%Array%	The initial value of the global object property named **"Array"**.
%ArrayPrototype%	The initial value of the **"prototype"** data property of the intrinsic %Array%.
%ArrayIteratorPrototype%	The prototype object used for interator objects created by the CreateArrayIterator abstract operation.
%Map%	The initial value of the global object property named **"Map"**.
%MapPrototype%	The initial value of the **"prototype"** data property of the intrinsic %Map%.
%MapIteratorPrototype%	The prototype object used for interator objects created by the CreateMapIterator abstract operation
%WeakMap%	The initial value of the global object property named **"WeakMap"**.
%WeakMapPrototype%	The initial value of the **"prototype"** data property of the intrinsic %WeakMap%.
%Set%	The initial value of the global object property named **"Set"**.
%SetPrototype%	The initial value of the **"prototype"** data property of the intrinsic %Set%.
%SetIteratorPrototype%	The prototype object used for interator objects created by the CreateSetterator abstract operation
%StopIteration%	
???	

10.4 Execution Contexts

An *execution context* is a specification device that is used to track the runtime evaluation of code by an ECMAScript implementation. At any point in time, there is at most one execution context that is actually executing code. This is known as the *running* execution context. A stack is used to track execution contexts. The running execution context is always the top element of this stack. 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. Each execution context has the state components listed in Table 26 .

Table 26 —State Components for All Execution Contexts

Component	*Purpose*
code evaluation state	Any state needed to perform, suspend, and resume evaluation of the code associated with this execution context.
Realm	The Realm from which associated code accesses ECMAScript resources.

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

Execution contexts for ECMAScript code have the additional state components listed in Table 27.

Table 27 —Additional State Components for ECMAScript Code Execution Contexts

Component	*Purpose*
LexicalEnvironment	Identifies the Lexical Environment used to resolve identifier references made by code within this execution context.
VariableEnvironment	Identifies the Lexical Environment whose environment record holds bindings created by *VariableStatements* within this execution context.

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 directly access or observe 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 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 StringValue of *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.

10.4.2 GetThisEnvironment

The abstract operation GetThisEnviroment finds the lexical environment that currently supplies the binding of the keyword **`this`**. GetThisEnviroment performs the following steps:

1. Let *lex* be the running 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*.

NOTE The loop in step 4 will always terminate because the llst of environment always end with the global environment which has a **`this`** binding.

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 running 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*.

10.4.4 GetGlobalObject

The abstract operation GetGlobalObject returns the global object used by the currently running execution context. GetGlobalObject Performs the following steps:

1. Let *ctx* be the running execution context.
2. Let *currentRealm* be *ctx's* Realm.
3. Return *currentRealm*.[[globalThis]].

10.5 Declaration Binding Instantiation

10.5.1 Global Declaration Instantiation

NOTE When an execution context is established for evaluating scripts, declarations are instantiated in the current global environment. Each global binding declared in the code is instantiated.

Global Declaration Instantiation is performed as follows using arguments *script*, *env*, and *deletableBindings*. *script* is the *ScriptBody* that for which the execution context is being established. *env* is the global environment record in which bindings are to be created. *deletableBindings* is **true** if the bindings that are created should be deletable.

1. Let *strict* be IsStrict of *script*.
2. Let *lexNames* be the LexicallyDeclaredNames of *script*.
3. Let *varNames* be the VarDeclaredNames of *script*.
4. For each *name* in *lexNames*, do
    a. If the result of calling *env's* HasVarDeclaration concrete method passing *name* as the argument is **true**, throw a SyntaxError exception.
    b. If the result of calling *env's* HasLexicalDeclaration concrete method passing *name* as the argument is **true**, throw a SyntaxError exception.

5. For each *name* in *varNames*, do
    a. If the result of calling *env's* HasLexicalDeclaration concrete method passing *name* as the argument is **true**, throw a SyntaxError exception.
6. Let *varDeclarations* be the VarScopedDeclarations of *script*.
7. Let *functionsToInitialize* be an empty List.
8. Let *declaredFunctionNames* be an empty List.
9. For each *d* in *varDeclarations*, in reverse list order do
    a. If *d* is a *FunctionDeclaration* then
        i. NOTE If there are multiple *FunctionDeclarations* for the same name, the last declaration is used.
        ii. Let *fn* be the sole element of the BoundNames of *d*.
        iii. If *fn* is not an element of *declaredFunctionNames*, then
            1. Let *fnDefinable* be the result of calling *env's* CanDeclareGlobalFunction concrete method passing *fn* as the argument.
            2. If *fnDefinable* is **false**, throw TypeError exception.
            3. Append *fn* to *declaredFunctionNames*.
            4. Append *d* to *functionsToInitialize*.
10. Let *declaredVarNames* be an empty List.
11. For each *d* in *varDeclarations*, do
    a. If *d* is a *VariableStatement* then
        i. For each String *vn* in the BoundNames of *d*, do
            1. If *vn* is not an element of *declaredFunctionNames*, then
                a Let *vnDefinable* be the result of calling *env's* CanDeclareGlobalVar concrete method passing *vn* and *deletableBindings* as the arguments.
                b If *vnDefinable* is **false**, throw TypeError exception.
                c If *vn* is not an element of *declaredVarNames*, then
                    i. Append *vn* to *declaredVarNames*.
12. NOTE: No abnormal terminations occur after this algorithm step.
13. For each *FunctionDeclaration f* in *functionsToInitialize*, do
    a. Let *fn* be the sole element of the BoundNames of *f*.
    b. Let *fo* be the result of performing InstantiateFunctionObject for *f* with argument *env*.
    c. Call *env*'s CreateGlobalFunctionBinding concrete method passing *fn*, *fo*, and *deletableBindings* as the arguments.
14. For each String *vn* in *declaredVarNames*, in list order do
    a. Call *env's* CreateGlobalVarBinding concrete method passing *vn* and *deletableBindings* as the argument.
15. Let *lexDeclarations* be the LexicallyScopedDeclarations of *script*.
16. For each element *d* in *lexDeclarations* do
    a. NOTE Lexically declared names are only instantiated here but not initialized.
    b. 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.
17. Return NormalCompletion(empty)

NOTE Early errors specified in 14.1 prevent name conflicts between function/var declarations and let/const/class/module declarations as well as redeclaration of let/const/class/module bindings for declaration contained within a single *Script*. However, such conflicts and redeclarations that span more than one *Script* are detected as runtime errors during Global Declaration Instantiation. If any such errors are detected, no bindings are instantiated for the script.

Unlike explicit var or function declarations, properties that are directly created on the global object result in global bindings that may be shadowed by let, const, class, and module declarations.

10.5.2 Module Declaration Instantiation

10.5.3 Function Declaration Instantiation

This version reflects the concensus as of the Sept. 2012 TC39 meeting. However, it now appears that the binding semantics of formal parameters is like to change again.

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 data property of *func*.
2. Let *strict* be the value of the [[Strict]] internal data property of *func*.
3. Let *formals* be the value of the [[FormalParameters]] internal data property of *func*.
4. Let *parameterNames* be the BoundNames of *formals*.
5. Let *varDeclarations* be the VarScopedDeclarations of *code*.
6. Let *functionsToInitialize* be an emptyList.
7. Let *argumentsObjectNotNeeded* be **false**.
8. For each *d* in *varDeclarations*, in reverse list order do
    a. If *d* is a *FunctionDeclaration* then
        i. NOTE If there are multiple *FunctionDeclarations* for the same name, the last declaration is used.
        ii. Let *fn* be the sole element of the BoundNames of *d*.
        iii. If *fn* is "`arguments`", then let *argumentsObjectNotNeeded* be **true**.
        iv. Let *alreadyDeclared* be the result of calling *env's* HasBinding concrete method passing *fn* as the argument.
        v. If *alreadyDeclared* is **false**, then
            1. Let *status* be the result of calling *env's* CreateMutableBinding concrete method passing *fn* as the argument.
            2. Assert: *status* is never an Abrupt Completion.
            3. Append *d* to *functionsToInitialize*.
9. For each String *paramName* in *parameterNames*, do
    a. Let *alreadyDeclared* be the result of calling *env's* HasBinding concrete method passing *paramName* as the argument.
    b. NOTE Duplicate parameter names can only occur in non-strict functions. Parameter names that are the same as function declaration names do not get initialized to **undefined**.
    c. If *alreadyDeclared* is **false**, then
        i. If *paramName* is "`arguments`", then let *argumentsObjectNotNeeded* be **true**.
        ii. Let *status* be the result of calling *env's* CreateMutableBinding concrete method passing *paramName* as the argument.
        iii. Assert: *status* is never an Abrupt Completion
        iv. Call *env*'s InitializeBinding concrete method passing *paramName*, and **undefined** as the arguments.
10. NOTE If there is a function declaration or formal parameter with the name `"arguments"` then an argument object is not created.
11. If *argumentsObjectNotNeeded* 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.

12. Let *varNames* be the VarDeclaredNames of *code*.
13. 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 initialied here if it is not also the name of a formal parameter or a *FunctionDeclarations*.
    c. If *alreadyDeclared* is **false**, then
        i. Call *env's* CreateMutableBinding concrete method passing *varName* as the argument.
14. Let *lexDeclarations* be the LexicalDeclarations of *code*.
15. For each element *d* in *lexDeclarations* do
    a. NOTE A lexically declared name can not be the same as a function declaration, formal parameter, or a var name. Lexically declarated names are only instantiated here but not initialized.
    b. 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.
16. For each *FunctionDeclaration f* in *functionsToInitialize*, do
    a. Let *fn* be the sole element of the BoundNames of *f*.
    b. Let *fo* be the result of performing InstantiateFunctionObject for *f* with argument *env*.
    c. Call *env*'s SetMutableBinding concrete method passing *fn*, *fo*, and **false** as the arguments.
17. NOTE Function declaration are initialised prior to parameter initialisation so that default value expressions may reference them. it is not extended code. "**`arguments`**" is not initialized until after parameter initialization.
18. Let *ao* be the result of InstantiateArgumentsObject with argument *argumentsList*.
19. NOTE If *argumentsObjectNotNeeded* 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.
20. Let *formalStatus* be the result of performing Binding Initialisation for *formals* with *ao* and **undefined** as arguments.
21. ReturnIfAbrupt(*formalStatus*).
22. If *argumentsObjectNotNeeded* is **false**, then
    a. If *strict* is **true**, then
        i. Perform the abstract operation CompleteStrictArgumentsObject with argument *ao*.
    b. Else,
        i. Perform the abstract operation CompleteMappedArgumentsObject with arguments *ao*, *func*, *formals*, and *env*.
    c. Call *env*'s InitializeBinding concrete method passing "**`arguments`**" and *ao* as arguments.
23. 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 created.

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 InstantiateFunctionObject for *f* with argument *env*.
c. Call *env*'s InitializeBinding concrete method passing *fn*, and *fo* as the arguments.

10.5.5 Eval Declaration Instantiation

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 *FunctionDeclaration* contained in the outermost *StatementList* of the function code.

The abstract operation InstantiateArgumentsObject called with an argument *args* performs the following steps:

1. Let *len* be the number of elements in *args*.
2. Let *obj* be the result of the abstract operation ObjectCreate.
3. Add the [[BuiltinBrand]] internal data property to *obj* with value BuiltinArguments.
4. Call the [[DefineOwnProperty]] internal method on *obj* passing **`"length"`** and the Property Descriptor {[[Value]]: *len*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true**} as arguments.
5. Let *indx* = *len* - 1.
6. Repeat while *indx* ≥ 0,
    a. Let *val* be the element of *args* at 0-origined list position *indx*.
    b. Call the [[DefineOwnProperty]] internal method on *obj* passing ToString(*indx*) and the Property Descriptor {[[Value]]: *val*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**} as arguments.
    c. Let *indx* = *indx* - 1
7. Return *obj*

The abstract operation *CompleteStrictArgumentsObject* called with argument *obj* which must have been previous created by the abstraction operation InstantiateArgumentsObject. The following steps are performed:

1. Perform the AddRestrictedFunctionProperties abstract operation with argument *obj*.
2. Return.

The abstract operation *Complete*MappedArgumentsObject is called with object *obj*, object *func*, grammar production *formals*, and environment record *env*. *obj* must have been previous created by the abstraction operation InstantiateArgumentsObject.The following steps are performed:

1. Let *len* be the result of Get(*obj*, **`"length"`**).
2. Let *mappedNames* be an empty List.
3. Let *numberOfNonRestFormals* be NumberOfParameters of *formals*.
4. Let *map* be the result of the abstract operation ObjectCreate.
5. Let *indx* = *len* - 1.
6. Repeat while *indx* ≥ 0,
    a. If *indx* is less than the *numberOfNonRestFormals*, then
        i. Let *param* be getParameter of *formals* with argument *indx*.
        ii. If *param* is a *BindingIdentifier*, then
            1. Let *name* be the sole element of BoundNames of *param*.
            2. If *name* is not an element of *mappedNames*, then
                a Add *name* as an element of the list *mappedNames*.
                b Let *g* be the result of calling the MakeArgGetter abstract operation with arguments *name* and *env*.
                c Let *p* be the result of calling the MakeArgSetter abstract operation with arguments *name* and *env*.
                d Call the [[DefineOwnProperty]] internal method of *map* passing ToString(*indx*) and the Property Descriptor {[[Set]]: *p*, [[Get]]: *g*, [[Configurable]]: **true**} as arguments.
    b. Let *indx* = *indx* - 1
7. If *mappedNames* is not empty, then
    a. Set the [[ParameterMap]] internal data property of *obj* to *map*.

   b. Set the [[GetP]], [[GetOwnProperty]], [[DefineOwnProperty]], and [[Delete]] internal methods of *obj* to the definitions provided below.
8. Call the [[DefineOwnProperty]] internal method on *obj* passing "**`callee`**" and the Property Descriptor {[[Value]]: *func*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true**} as arguments.
9. 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 calling the abstract operation FunctionCreate using Normal as the *kind*, *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.
5. Return the result of calling the abstract operation FunctionCreate using Normal as the *kind*, *parameters* as *FormalParameterList*, *body* for *FunctionBody*, *env* as *Scope*, and **true** for *Strict*.

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

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 data 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 Get(*map*, *P*).
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* and Property Descriptor *Desc* performs the following steps:

1. Let *map* be the value of the [[ParameterMap]] internal data 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 (8.3.9) on the arguments object passing *P* and *Desc* as the arguments.
4. ReturnIfAbrupt(allowed).
5. If *allowed* is **false**, then return **false**.
6. 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* as the argument.
    b. Else
        i. If *Desc*.[[Value]] is present, then
            1. Asset: the follow Put call will always succeed because formal parameters mapped by argument objects are always writable.
            2. Call Put(*map*, *P*, *Desc*.[[Value]], **false**).
        ii. If *Desc*.[[Writable]] is present and its value is **false**, then
            1. Call the [[Delete]] internal method of *map* passing *P* as the argument.
7. Return **true**.

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

1. Let *map* be the value of the [[ParameterMap]] internal data 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 for ordinary objects (8.3.10) on the arguments object passing *P* as the argument.
4. If *result* is **true** and the value of *isMapped* is not **undefined**, then
    a. Call the [[Delete]] internal method of *map* passing *P* as the argument.
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
ArrayInitialiser
ObjectLiteral
FunctionExpression
ClassExpression
GeneratorExpression
GeneratorComprehension
RegularExpressionLiteral
TemplateLiteral
CoverParenthesizedExpressionAndArrowParameterList

*CoverParenthesizedExpressionAndArrowParameterList***:**
(*Expression* **)**
( )
(... *Identifier* **)**
(*Expression* **, ...** *Identifier***)**

Supplemental Syntax

When processing the production *PrimaryExpression* **:** *CoverParenthesizedExpressionAndArrowParameterList* the following grammar is used to refine the interpretation of *CoverParenthesizedExpressionAndArrowParameterList*.

ParenthesizedExpression **:**
(*Expression* **)**

Static Semantics

Static Semantics: CoveredParenthesizedExpression

CoverParenthesizedExpressionAndArrowParameterList **: (** *Expression* **)**

1. Return the result of parsing the lexical token stream matched by *CoverParenthesizedExpressionAndArrowParameterList* using *ParenthesizedExpression* as the goal symbol.

Static Semantics: IsValidSimpleAssignmentTarget

PrimaryExpression **:**
this
Literal
ArrayInitialiser
ObjectLiteral
FunctionExpression
ClassExpression
GeneratorExpression
GeneratorComprehension
RegularExpressionLiteral
TemplateLiteral

1. Return **false**.

PrimaryExpression **:** *Identifier*

1. If this *PrimaryExpression* is contained in strict code and StringValue of *Identifier* is `"eval"` or `"arguments"`, then return **false**.
2. Return **true**.

PrimaryExpression **:** *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *expr* be CoveredParenthesizedExpression of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return IsValidSimpleAssignmentTarget of *expr*.

11.1.1 The `this` Keyword

Runtime Semantics: Evaluation

PrimaryExpression **:** `this`

1. Return the result of calling the ThisResolution abstract operation.

11.1.2 Identifier Reference

Runtime Semantics: Evaluation

PrimaryExpression **:** *Identifier*

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*.

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

11.1.3 Literals

Syntax

Literal **:**
NullLiteral
ValueLiteral

ValueLiteral **:**
BooleanLiteral
NumericLiteral
StringLiteral

Runtime Semantics

Runtime Semantics: Evaluation

Literal **:** *NullLiteral*

1. Return **null**.

ValueLiteral **:** *BooleanLiteral*

1. Return **false** if *BooleanLiteral* is the token *BooleanLiteral* **::** `false`
2. Return **true** if *BooleanLiteral* is the token *BooleanLiteral* **::** `true`

ValueLiteral **:** *NumericLiteral*

1. Return the number whose value is MV of *NumericLiteral* as defined in 7.8.3.

ValueLiteral **:** *StringLiteral*

1. Return the string whose elements are the SV of *StringLiteral* as defined in 7.8.4.

11.1.4 Array Initialiser

Syntax

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 **:**
[$Elision_{opt}$ **]**
[*ElementList* **]**
[*ElementList* **,** $Elision_{opt}$ **]**

ElementList **:**
$Elision_{opt}$ *AssignmentExpression*
$Elision_{opt}$ *SpreadElement*
ElementList **,** $Elision_{opt}$ *AssignmentExpression*
ElementList **,** $Elision_{opt}$ *SpreadElement*

Elision **:**
,
Elision **,**

SpreadElement **:**
... *AssignmentExpression*

Static Semantics

Static Semantics: Elision Width

Elision **:** **,**

1. Return the numeric value 1.

Elision **:** *Elision* **,**

1. Let *preceding* be the Elision Width of *Elision*.
2. Return *preceding*+1.

Runtime Semantics

Runtime Semantics: Array Accumulation

With parameters *array* and *nextIndex*.

ElementList **:** *Elision*$_{opt}$ *AssignmentExpression*

1. Let *padding* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
2. Let *initResult* be the result of evaluating *AssignmentExpression*.
3. Let *initValue* be GetValue(*initResult*).
4. ReturnIfAbrupt(*initValue*).
5. Call the [[DefineOwnProperty]] internal method of *array* with arguments ToString(ToUint32(*nextIndex+padding*)) and the Property Descriptor { [[Value]]: *initValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
6. Assert: the above call to [[DefineOwnProperty]] will return **false** or an abrupt completion value.
7. Return *nextIndex+padding*+1.

ElementList **:** *Elision*$_{opt}$ *SpreadElement*

1. Let *padding* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
2. Return the result of performing Array Accumulation for *SpreadElement* with arguments *array* and *nextIndex+padding*.

ElementList **:** *ElementList* **,** *Elision*$_{opt}$ *AssignmentExpression*

1. Let *postIndex* be the result of performing Array Accumulation for *ElementList* with arguments *array* and *nextIndex*.
2. ReturnIfAbrupt(*postIndex*).
3. Let *padding* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
4. Let *initResult* be the result of evaluating *AssignmentExpression*.
5. Let *initValue* be GetValue(*initResult*).
6. ReturnIfAbrupt(*initValue*).
7. Call the [[DefineOwnProperty]] internal method of *array* with arguments ToString(ToUint32(*postIndex+padding*)) and the Property Descriptor { [[Value]]: *initValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
8. Assert: the above call to [[DefineOwnProperty]] will return **false** or an abrupt completion value.
9. Return *postIndex+padding*+1.

ElementList **:** *ElementList* **,** *Elision*$_{opt}$ *SpreadElement*

1. Let *postIndex* be the result of performing Array Accumulation for *ElementList* with arguments *array* and *nextIndex*.
2. ReturnIfAbrupt(*postIndex*).
3. Let *padding* be the Elision Width of *Elision*; if *Elision* is 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. ReturnIfAbrupt(*spreadObj*).
5. Let *lenVal* be the result of calling Get(*spreadObj*, **"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 HasProperty(*spreadObj*, ToString(*n*)).
   b. ReturnIfAbrupt(*exists*).
   c. If *exists* is **true** then,

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)*) and Property Descriptor {[[Value]]: *v*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
d. Assert: the above call to [[DefineOwnProperty]] will return **false** or an abrupt completion value.
e. Let *n* = *n*+1.
f. 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 [[SetP]].

Runtime Semantics: Evaluation

ArrayLiteral : **[** *Elision*$_{opt}$ **]**

1. Let *array* be the result of the abstract operation ArrayCreate with argument 0.
2. Let *pad* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
3. Call Put(*array*, **"length"**, *pad*, **false**).
4. Return *array*.

ArrayLiteral : **[** *ElementList* **]**

1. Let *array* be the result of the abstract operation ArrayCreate with argument 0.
2. Let *len* be the result of performing Array Accumulation for *ElementList* with arguments *array* and 0.
3. ReturnIfAbrupt(*len*).
4. Call Put(*array*, **"length"**, *len*, **false**).
5. Return *array*.

ArrayLiteral : **[** *ElementList* **,** *Elision*$_{opt}$ **]**

1. Let *array* be the result of the abstract operation ArrayCreate with argument 0.
2. Let *len* be the result of performing Array Accumulation for *ElementList* with arguments *array* and 0.
3. ReturnIfAbrupt(*len*).
4. Let *padding* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
5. Call Put(*array*, **"length"**, ToUint32(*padding*+*len*), **false**).
6. Return *array*.

11.1.4.2 Array Comprehension

Syntax

ArrayComprehension :
[*AssignmentExpression* *ComprehensionForList* **]**
[*AssignmentExpression* *ComprehensionForList* **if** *Expression* **]**

ComprehensionForList :
ComprehensionFor
ComprehensionForList *ComprehensionFor*

ComprehensionFor :
for *ForBinding* **of** *Expression*

ForBinding :
BindingIdentifier
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 **:** *BindingPattern*

1. Let *obj* be ToObject(*value*).
2. ReturnIfAbrupt(*obj*).
3. Return the result of performing Binding Initialisation for *BindingPattern* passing *obj* and *environment* as the arguments.

Runtime Semantics: Evaluation

ToDo

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

ObjectLiteral **:**
{ }
{ *PropertyDefinitionList* **}**
{ *PropertyDefinitionList* **, }**

PropertyDefinitionList **:**
PropertyDefinition
PropertyDefinitionList **,** *PropertyDefinition*

PropertyDefinition **:**
IdentifierName
CoverInitialisedName
PropertyName **:** *AssignmentExpression*
MethodDefinition

PropertyName **:**
IdentifierName
StringLiteral
NumericLiteral

CoverInitialisedName **:**
IdentifierName Initialiser

Initialiser **:**
= *AssignmentExpression*

NOTE 1 *MethodDefinition* is defined in 13.3.

NOTE 2 In certain contexts, *ObjectLiteral* is used as a cover grammar for a more restricted secondary grammar. The *CoverInitialisedName* production is necessary to fully cover these secondary grammars. However, use of this production results in an early Syntax Error in normal contexts where an actual *ObjectLiteral* is expected.

Static Semantics

Static Semantics: Early Errors

In addition to describe an actual object initialiser the *ObjectLiteral* productions are used as a cover grammar for *ObjectAssignmentPattern* (11.13.1). When *ObjectLiteral* appears in a context where *ObjectAssignmentPattern* is required, the following Early Error rules are not applied.

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 *MethodDefinition* and the other occurrence was obtained from a **`set`** accessor *MethodDefinition*.

PropertyDefinition **:** *MethodDefinition*

- It is a Syntax Error if ReferencesSuper of *MethodDefinition* is **true**.

PropertyDefinition **:** *IdentifierName*

- It is a Syntax Error if *IdentifierName* is a *ReservedWord*.

PropertyDefinition **:** *CoverInitialisedName*

- Always throw a Syntax Error if this production is present

NOTE This production exists so that *ObjectLiteral* can serve as a cover grammar for *ObjectAssignmentPattern* (11.13.1). It can not occur in an actual object initialiser.

Static Semantics: Contains

With parameter *symbol*.

PropertyDefinition **:** *MethodDefinition*

1. If *symbol* is MethodDefinition, return **true**.
2. Return **false**.

NOTE Static semantic rules that depend upon substructure generally do not look into function definitions.

PropertyName **:** *IdentifierName*

1. If *symbol* is a *ReservedWord*, return **false**.
2. If *symbol* is an *Identifier* and StringValue of *symbol* is the same value as the StringValue of *IdentifierName*, return **true**;
3. Return **false**.

Static Semantics: IsValidSimpleAssignmentTarget

PrimaryExpression : *Literal*

1. Return **false**.

Static Semantics: PropName

PropertyDefinition : *IdentifierName*

1. Return StringValue of *IdentifierName*.

PropertyDefinition : *PropertyName* : *AssignmentExpression*

1. Return PropName of *PropertyName*.

PropertyName : *StringLiteral*

1. Return a String value whose characters are the SV of the *StringLiteral*.

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*.
2. Append PropName of *PropertyDefinition* to the end of *list*.
3. Return *list*.

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 abstract operation ObjectCreate.
2. Let *status* be the result of performing Property Definition Evaluation of *PropertyDefinitionList* with argument *obj*.
3. ReturnIfAbrupt(*status*).
4. Return *obj*.

Runtime Semantics: Property Definition Evaluation

With parameter *object*.

PropertyDefinitionList : *PropertyDefinitionList* , *PropertyDefinition*

1. Let *status* be the result of performing Property Definition Evaluation of *PropertyDefinitionList* with argument *object*.
2. ReturnIfAbrupt(*status*).
3. Return the result of performing Property Definition Evaluation of *PropertyDefinition* with argument *object*.

PropertyDefinition **:** *IdentifierName*

1. Let *propName* be StringValue of *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. ReturnIfAbrupt(*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* and *desc*

PropertyDefinition **:** *PropertyName* **:** *AssignmentExpression*

1. Let *propName* be PropName of *PropertyName*.
2. Let *exprValue* be the result of evaluating *AssignmentExpression*.
3. Let *propValue* be GetValue(*exprValue*).
4. ReturnIfAbrupt(*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* and *desc*.

11.1.6 Function Defining Expressions

See 13.1 for *PrimaryExpression* **:** *FunctionExpression*.

See 13.4 for *PrimaryExpression* **:** *GeneratorExpression*.

See 13.5 for *PrimaryExpression* **:** *ClassExpression*.

11.1.7 Generator Comprehensions

Syntax

GeneratorComprehension **:**
(*Expression ComprehensionForList* **)**
(*Expression ComprehensionForList* **if** *Expression* **)**

11.1.8 Regular Expression Literals

Syntax

See 7.8.5.

Static Semantics

Static Semantics: Early Errors

PrimaryExpression **:** *RegularExpressionLiteral*

- It is a Syntax Error if BodyText of *RegularExpressionLiteral* can not be recognized using the goal symbol *Pattern* of the ECMAScript RegExp grammar specified in 15.10.
- It is a Syntax Error if FlagText of *RegularExpressionLiteral* contains any character other than **"g"**, **"i"**, **"m"**, **"u"**, or **"y"**, or if it contains the same character more than once.

Runtime Semantics

Runtime Semantics: Evaluation

PrimaryExpression **:** *RegularExpressionLiteral*

1. 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*.

11.1.9 Template Literals

Syntax

TemplateLiteral **:**
NoSubstitutionTemplate
TemplateHead Expression [Lexical goal *InputElementTemplateTail*] *TemplateSpans*

*TemplateSpans***:**
TemplateTail
TemplateMiddleList [Lexical goal *InputElementTemplateTail*] *TemplateTail*

*TemplateMiddleList***:**
TemplateMiddle Expression
TemplateMiddleList [Lexical goal *InputElementTemplateTail*] *TemplateMiddle Expression*

Static Semantics

Static Semantics: TemplateStrings

With parameter *raw*.

TemplateLiteral **:** *NoSubstitutionTemplate*

1. If *raw* is **false**, then
   a. Let *string* be the TV of *NoSubstitutionTemplate*.
2. Else,
   a. Let *string* be the TRV of *NoSubstitutionTemplate*.
3. Return a List containing the single element, *string*.

TemplateLiteral **:** *TemplateHead Expression* [Lexical goal *InputElementTemplateTail*] *TemplateSpans*

1. If *raw* is **false**, then
   a. Let *head* be the TV of *TemplateHead*.
2. Else,
   a. Let *head* be the TRV of *TemplateHead*.
3. Let *tail* be TemplateStrings of *TemplateSpans* with argument *raw*.
4. Return a List containing *head* followed by the element, in order of *tail*.

TemplateSpans **:** *TemplateTail*

1. If *raw* is **false**, then
   a. Let *tail* be the TV of *TemplateTail*.
2. Else,
   a. Let *tail* be the TRV of *TemplateTail*.
3. Return a List containing the single element, *tail*.

TemplateSpans **:** *TemplateMiddleList* [Lexical goal *InputElementTemplateTail*] *TemplateTail*

1. Let *middle* be TemplateStrings of *TemplateMiddleList* with argument *raw*.
2. If *raw* is **false**, then
    a. Let *tail* be the TV of *TemplateTail*.
3. Else,
    a. Let *tail* be the TRV of *TemplateTail*.
4. Return a List containing the elements, in order, of *middle* followed by *tail*.

TemplateMiddleList **:** *TemplateMiddle Expression*

1. If *raw* is **false**, then
    a. Let *string* be the TV of *TemplateMiddle*.
2. Else,
    a. Let *string* be the TRV of *TemplateMiddle*.
3. Return a List containing the single element, *string*.

TemplateMiddleList **:** *TemplateMiddleList* [Lexical goal *InputElementTemplateTail*] *TemplateMiddle Expression*

1. Let *front* be TemplateStrings of *TemplateMiddleList* with argument *raw*.
2. If *raw* is **false**, then
    a. Let *last* be the TV of *TemplateMiddle*.
3. Else,
    a. Let *last* be the TRV of *TemplateMiddle*.
4. Append *last* as the last elemnt of the List *front*.
5. Return *front*.

Runtime Semantics

Runtime Semantics: **ArgumentListEvaluation**

TemplateLiteral **:** *NoSubstitutionTemplate*

1. Let *siteObj* be the result of the abstraction operation GetTemplateCallSite passing this *TemplateLiteral* production as the argument.
2. Return a List containing the one element which is *siteObj*.

TemplateLiteral **:** *TemplateHead Expression* [Lexical goal *InputElementTemplateTail*] *TemplateSpans*

1. Let *siteObj* be the result of the abstraction operation GetTemplateCallSite passing this *TemplateLiteral* production as the argument.
2. Let *firstSub* be the result of evaluating *Expression*.
3. ReturnIfAbrupt(*firstSub*).
4. Let *restSub* be SubstitutionEvaluation of *TemplateSpans*.
5. ReturnIfAbrupt(*restSub*).
6. Assert, *restSub* is a List.
7. Return a List whose first element is *siteObj*, whose second elements is *firstSub*, and whose subsequent elements are the elements of *restSub*, in order. *restSub* may contain no elements.

Runtime Semantics: GetTemplateCallSite Abstract Operation

The abstract operation GetTemplateCallSite is called with a grammar production, *templateLiteral*, as an argument. It performs the following steps:

1. If a call site object for the source code corresponding to *templateLiteral* has already been created by a previous call to this abstract operation, then return that call site object.
2. Let *cookedStrings* be TemplateStrings of *templateLiteral* with argument **false**.
3. Let *rawStrings* be TemplateStrings of *templateLiteral* with argument **true**.
4. Let *count* be the number of elements in the List *cookedStrings*.
5. Let *siteObj* be the result of the abstraction operation ArrayCreate with argument *count*.

6. Let *rawObj* be the result of the abstraction operation ArrayCreate with argument *count*.
7. Let *index* be 0.
8. Repeat while *index* < *count*
    a. Let *prop* be ToString(*index*).
    b. Let *cookedValue* be the string value at 0-based position *index* of the List *cookedStrings*.
    c. Call the [[DefineOwnProperty]] internal method of *siteObj* with arguments *prop* **and** Property Descriptor {[[Value]]: *cookedValue*, [[Writable]]: **false**, [[Configurable]]: **false**}.
    d. Let *rawValue* be the string value at 0-based position *index* of the List *rawStrings*.
    e. Call the [[DefineOwnProperty]] internal method of *rawObj* with arguments *prop* **and** Property Descriptor {[[Value]]: *rawValue*, [[Writable]]: **false**, [[Configurable]]: **false**}.
    f. Let *index* be *index*+1.
9. Call the [[Freeze]] internal method of *rawObj*.
10. Call the [[DefineOwnProperty]] internal method of *siteObj* with arguments **"raw"** and Property Descriptor {[[Value]]: *rawObj*, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false**}.
11. Call the [[Freeze]] internal method of *siteObj*.
12. Remember an association between the source code corresponding to *templateLiteral* and siteObj such that *siteObj* can be retrieve in subsequent calls to this abstract operation.
13. Return *siteObj*.

NOTE 1 The creation of a call site object cannot result in an abrupt completion.

NOTE 2 Each *TemplateLiteral* in the program code is associated with a unique Template call site object that is used in the evaluation of tagged Templates (11.2.6). The same call site object is used each time a specific tagged Template is evaluated. Whether call site objects are created lazily upon first evaluation of the *TemplateLiteral* or eagerly prior to first evaluation is an implementation choice that is not observable to ECMAScript code.

Runtime Semantics: SubstitutionEvaluation

TemplateSpans **:** *TemplateTail*

1. Return an empty List.

TemplateSpans **:** *TemplateMiddleList* [Lexical goal *InputElementTemplateTail*] *TemplateTail*

1. Return the result of SubstitutionEvaluation of *TemplateMiddleList*.

TemplateMiddleList **:** *TemplateMiddle Expression*

1. Let *sub* be the result of evaluating *Expression*.
2. ReturnIfAbrupt(*sub*).
3. Return a List containing only *sub*.

TemplateMiddleList **:** *TemplateMiddleList* [Lexical goal *InputElementTemplateTail*] *TemplateMiddle Expression*

1. Let *preceeding* be the result of SubstitutionEvaluation of *TemplateMiddleList* .
2. ReturnIfAbrupt(*preceeding*).
3. Let *next* be the result of evaluating *Expression*.
4. ReturnIfAbrupt(*next*).
5. Append *next* as the list element of the List *preceeding*.
6. Return *preceeding*.

Runtime Semantics: Evaluation

TemplateLiteral **:** *NoSubstitutionTemplate*

1. Return the string value whose elements are the TV of *NoSubstitutionTemplate* as defined in 7.8.6.

TemplateLiteral **:** *TemplateHead Expression* [Lexical goal *InputElementTemplateTail*] *TemplateSpans*

1. Let *head* be the TV of *TemplateHead* as defined in 7.8.6.

2. Let *sub* be the result of evaluating *Expression*.
3. Let *middle* be ToString(*sub*).
4. ReturnIfAbrupt(*middle*).
5. Let *tail* be the result of evaluating *TemplateSpans* .
6. ReturnIfAbrupt(*tail*).
7. Return the string value whose elements are the code units of *head* followed by the code units of *tail*.

TemplateSpans **:** *TemplateTail*

1. Let *tail* be the TV of *TemplateTail* as defined in 7.8.6.
2. Return the string whose elements are the code units of *tail*.

TemplateSpans **:** *TemplateMiddleList* [Lexical goal *InputElementTemplateTail*] *TemplateTail*

1. Let *head* be the result of evaluating *TemplateMiddleList*.
2. ReturnIfAbrupt(*head*).
3. Let *tail* be the TV of *TemplateTail* as defined in 7.8.6.
4. Return the string whose elements are the elements of *head* followed by the elements of *tail*.

TemplateMiddleList **:** *TemplateMiddle Expression*

1. Let *head* be the TV of *TemplateMiddle* as defined in 7.8.6.
2. Let *sub* be the result of evaluating *Expression*.
3. Let *middle* be ToString(*sub*).
4. ReturnIfAbrupt(*middle*).
5. Return the sequence of characters consisting of the code units of *head* followed by the elements of *middle*.

TemplateMiddleList **:** *TemplateMiddleList* [Lexical goal *InputElementTemplateTail*] *TemplateMiddle Expression*

1. Let *rest* be the result of evaluating *TemplateMiddleList* .
2. ReturnIfAbrupt(*rest*).
3. Let *middle* be the TV of *TemplateMiddle* as defined in 7.8.6.
4. Let *sub* be the result of evaluating *Expression*.
5. Let *last* be ToString(*sub*).
6. ReturnIfAbrupt(*last*).
7. Return the sequence of characters consisting of the elements of *rest* followed by the code units of *middle* followed by the elements of *last*.

11.1.10 The Grouping Operator

Static Semantics: Early Errors

PrimaryExpression **:** *CoverParenthesizedExpressionAndArrowParameterList*

- It is a Syntax Error if the lexical token sequence matched by *CoverParenthesizedExpressionAndArrowParameterList* cannot be parsed with no tokens left over using *ParenthesizedExpression* as the goal symbol.
- All Early Errors rules for *ParenthesizedExpression* and its derived productions also apply to the CoveredParenthesizedExpression of *CoverParenthesizedExpressionAndArrowParameterList*.

Static Semantics: IsValidSimpleAssignmentTarget

PrimaryExpression **:** *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *expr* be CoveredParenthesizedExpression of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return IsValidSimpleAssignmentTarget of *expr*.

ParenthesizedExpression **:** **(** *Expression* **)**

1. Return IsValidSimpleAssignmentTarget of *Expression*.

Runtime Semantics: Evaluation

PrimaryExpression : *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *expr* be CoveredParenthesizedExpression of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return the result of evaluating *expr*.

ParenthesizedExpression : **(** *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.

11.2 Left-Hand-Side Expressions

Syntax

MemberExpression :
[Lexical goal *InputElementRegExp*] *PrimaryExpression*
MemberExpression **[** *Expression* **]**
MemberExpression **.** *IdentifierName*
MemberExpression *TemplateLiteral*
super **[** *Expression* **]**
super **.** *IdentifierName*
new *MemberExpression* *Arguments*

NewExpression :
MemberExpression
new *NewExpression*

CallExpression :
MemberExpression *Arguments*
super *Arguments*
CallExpression *Arguments*
CallExpression **[** *Expression* **]**
CallExpression **.** *IdentifierName*
CallExpression *TemplateLiteral*

Arguments :
(**)**
(*ArgumentList* **)**

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

LeftHandSideExpression :
NewExpression
CallExpression

Static Semantics

Static Semantics: Contains

With parameter *symbol*.

MemberExpression **:** *MemberExpression* **.** *IdentifierName*

1. If *MemberExpression* Contains *symbol* is **true**, return **true**.
2. If *symbol* is a *ReservedWord*, return **false**.
3. If *symbol* is an *Identifier* and StringValue of *symbol* is the same value as the StringValue of *IdentifierName*, return **true**;
4. Return **false**.

MemberExpression **:** **`super`** **.** *IdentifierName*

1. If *symbol* is the *ReservedWord* **`super`**, return **true**.
2. If *symbol* is a *ReservedWord*, return **false**.
3. If *symbol* is an *Identifier* and StringValue of *symbol* is the same value as the StringValue of *IdentifierName*, return **true**;
4. Return **false**.

CallExpression **:** *CallExpression* **.** *IdentifierName*

1. If *CallExpression* Contains *symbol* is **true**, return **true**.
2. If *symbol* is a *ReservedWord*, return **false**.
3. If *symbol* is an *Identifier* and StringValue of *symbol* is the same value as the StringValue of *IdentifierName*, return **true**;
4. Return **false**.

Static Semantics: IsValidSimpleAssignmentTarget

CallExpression **:**
MemberExpression *Arguments*
`super` *Arguments*
CallExpression *Arguments*
CallExpression **[** *Expression* **]**
CallExpression **.** *IdentifierName*

MemberExpression **:**
MemberExpression **[** *Expression* **]**
MemberExpression **.** *IdentifierName*
`super` **[** *Expression* **]**
`super` **.** *IdentifierName*

1. Return **true**.

CallExpression **:** *CallExpression* *TemplateLiteral*
NewExpression **:** **`new`** *NewExpression*
MemberExpression **:** **`new`** *MemberExpression* *Arguments*

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. ReturnIfAbrupt(*baseValue*).
4. Let *propertyNameReference* be the result of evaluating *Expression*.
5. Let *propertyNameValue* be GetValue(*propertyNameReference*).
6. ReturnIfAbrupt(*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 reference flag is *strict*.

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*.
2. Let *constructor* be GetValue(*ref*).
3. ReturnIfAbrupt(*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* with an empty List as the argument.

MemberExpression **:** **new** *MemberExpression Arguments*

1. Let *ref* be the result of evaluating *MemberExpression*.
2. Let *constructor* be GetValue(*ref*).
3. ReturnIfAbrupt(*constructor*).

4. Let *argList* be the result of evaluating *Arguments*, producing an internal List of argument values (11.2.4).
5. ReturnIfAbrupt(*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*, passing *argList* as the argument.

11.2.3 Function Calls

Runtime Semantics: Evaluation

CallExpression **:** *MemberExpression Arguments*

1. Let *ref* be the result of evaluating *MemberExpression*.
2. If this *CallExpression* is in a tail position (13.7) then let *tailCall* be **true**, otherwise let *tailCall* be **false**.
3. Return the result of the abstract operation EvaluateCall with arguments *ref*, *Arguments*, and *tailCall*.

CallExpression **:** *CallExpression Arguments*

1. Let *ref* be the result of evaluating *CallExpression*.
2. If this *CallExpression* is in a tail position (13.7) then let *tailCall* be **true**, otherwise let *tailCall* be **false**.
3. Return the result of the abstract operation EvaluateCall with arguments *ref*, *Arguments*, and *tailCall*.

Runtime Semantics: EvaluateCall Abstract Operation

The abstract operation EvaluateCall takes as arguments a value *ref*, and a syntactic grammar production *arguments*, and a Boolean argument *tailPosition*. It performs the following steps:

1. Let *func* be GetValue(*ref*).
2. ReturnIfAbrupt(*func*).
3. Let *argList* be the result of performing ArgumentListEvaluation of *arguments*.
4. ReturnIfAbrupt(*argList*).
5. If Type(*func*) is not Object, throw a **TypeError** exception.
6. If IsCallable(*func*) is **false**, throw a **TypeError** exception.
7. 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*).
8. Else Type(*ref*) is not Reference,
    a. Let *thisValue* be **undefined**.
9. If *tailPosition* is **true**, then
    a. Let *leafContext* be the running execution context.
    b. Suspend *leafContext*.
    c. Pop *leafContext* from the execution context context stack. The execution context now on the top of the stack becomes the running execution context, however it remains in its suspended state.
    d. Assert: *leafContext* has no further use. It will never be activated as the running execution context.
10. Let *result* be the result of calling the [[Call]] internal method on *func*, passing *thisValue* as the *thisArgument* and *argList* as the *argumentsList*.
11. Assert: If *tailPosition* is **true**, the above call will not return here, but instead evaluation will continue with the resumption of *leafCallerContext* as the running execution context.
12. Return *result*.

A tail position call must either release any transient internal resources associated with the currently executing function execution context before invoking the target function or reuse those resources in support of the target function.

NOTE 1 For example, a tail position call should only grow an implementation's activication record stack by the amount that the size of the target function's activation record exceeds the size of the calling function's activation record. If the target function's activation record is smaller, then the total size of the stack should decrease.

NOTE 2 The returned result will never be of type Reference if *func* is an ordinary object. Whether calling an exotic 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

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 GetSuperBase 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 reference 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 GetSuperBase concrete method of *env*.
5. ReturnIfAbrupt(CheckObjectCoercible(*baseValue*)).
6. Let *propertyKey* be StringValue 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 is *actualThis*, and whose strict reference flag is *strict*.

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 GetSuperBase concrete method of *env*.
5. ReturnIfAbrupt(CheckObjectCoercible(*baseValue*)).
6. Let *propertyKey* be the result of calling the GetMethodName 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. If this *CallExpression* is in a tail position (13.7) then let *tailCall* be **true**, otherwise let *tailCall* be **false**.
10. Return the result of the abstract operation EvaluateCall with arguments *ref*, *Arguments*, and *tailCall*.

11.2.5 Argument Lists

The evaluation of an argument list produces a List of values (see 8.7).

Runtime Semantics

Runtime Semantics: **ArgumentListEvaluation**

Arguments **:** **( )**

1. Return an empty List.

ArgumentList **:** *AssignmentExpression*

1. Let *ref* be the result of evaluating *AssignmentExpression*.
2. Let *arg* be GetValue(*ref*).
3. ReturnIfAbrupt(*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. ReturnIfAbrupt(*spreadObj*).
6. Let *lenVal* be the result of calling Get(*spreadObj*, **`"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 Get(*spreadObj*, ToString(*n*)).
    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. ReturnIfAbrupt(*precedingArgs*).
3. Let *ref* be the result of evaluating *AssignmentExpression*.
4. Let *arg* be GetValue(*ref*).
5. ReturnIfAbrupt(*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. ReturnIfAbrupt(*spreadObj*).
6. Let *lenVal* be the result of calling Get(*spreadObj*, **`"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 Get(*spreadObj*, ToString(*n*)).
    b. ReturnIfAbrupt(*nextArg*).
    c. Append *nextArg* as the last element of *precedingArgs*.
    d. Let *n* = *n*+1.
11. Return *precedingArgs*.

11.2.6 Tagged Templates

Runtime Semantics

Runtime Semantics: Evaluation

MemberExpression **:** *MemberExpression TemplateLiteral*

1. Let *tagRef* be the result of evaluating *MemberExpression*.
2. If this *MemberExpression* is in a tail position (13.7) then let *tailCall* be **true**, otherwise let *tailCall* be **false**.
3. Return the result of the abstract operation EvaluateCall with arguments *tagRef*, *TemplateLiteral*, and *tailCall*.

CallExpression **:** *CallExpression TemplateLiteral*

1. Let *tagRef* be the result of evaluating *CallExpression*.
2. If this *CallExpression* is in a tail position (13.7) then let *tailCall* be **true**, otherwise let *tailCall* be **false**.
3. Return the result of the abstract operation EvaluateCall with arguments *tagRef*, *TemplateLiteral*, and *tailCall*.

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 an early Reference Error if IsValidSimpleAssignmentTarget of *LeftHandSideExpression* is **false**.

Static Semantics: IsValidSimpleAssignmentTarget

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

1. Return **false**.

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. ReturnIfAbrupt(*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. Let *status* be PutValue(*lhs*, *newValue*).
6. ReturnIfAbrupt(*status*).
7. 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).
4. Let *status* be PutValue(*lhs*, *newValue*).
5. ReturnIfAbrupt(*status*).
6. 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*

- 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 : *CoverParenthesizedExpressionAndArrowParameterList*
  and derives a production that if used in place of *UnaryExpression* would produce a Syntax Error according to these rules. This rule is recursively applied.

UnaryExpression **:**
++ *UnaryExpression*
-- *UnaryExpression*

- It is an early Reference Error if IsValidSimpleAssignmentTarget of *UnaryExpression* is **false**.

Static Semantics: IsValidSimpleAssignmentTarget

UnaryExpression **:**
`delete` *UnaryExpression*
`void` *UnaryExpression*
`typeof` *UnaryExpression*
++ *UnaryExpression*
-- *UnaryExpression*
+ *UnaryExpression*
- *UnaryExpression*
~ *UnaryExpression*
! *UnaryExpression*

1. Return **false**.

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.

Runtime Semantics: Evaluation

UnaryExpression **:** **`delete`** *UnaryExpression*

1. Let *ref* be the result of evaluating *UnaryExpression*.
2. ReturnIfAbrupt(*ref*).
3. If Type(*ref*) is not Reference, return **true**.
4. If IsUnresolvableReference(*ref*) is **true**, then,
   a. If IsStrictReference(*ref*) is **true**, then throw a **SyntaxError** exception.
   b. Return **true**.
5. If IsPropertyReference(*ref*) is **true**, then
   a. If IsSuperReference(*ref*), then throw a **ReferenceError** exception.
   b. Let *deleteStatus* be the result of calling the [[Delete]] internal method on ToObject(GetBase(*ref*)), providing GetReferencedName(*ref*) as the argument.
   c. ReturnIfAbrupt(*deleteStatus*).
   d. If *deleteStatus* is **false** and IsStrictReference(*ref*) is **true**, then throw a **typeError** exception.
   e. Return **true**.
6. Else *ref* is a Reference to an Environment Record binding,
   a. Let *bindings* be GetBase(*ref*).
   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. Let *status* be Call GetValue(*expr*).
3. ReturnIfAbrupt(*status*).
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. ReturnIfAbrupt(*val*).
4. Return a String determined by Type(*val*) according to **Table 28** .

Table 28 — typeof Operator Results

Type of *val*	***Result***
Undefined	**`"undefined"`**
Null	**`"object"`**
Boolean	**`"boolean"`**
Number	**`"number"`**
String	**`"string"`**
Object (ordinary and does not implement [[Call]])	**`"object"`**
Object (implements [[Call]])	**`"function"`**
Object (exotic and does not implement [[Call]])	Implementation-defined unless explicitly specified. May not be **`"undefined"`**, **`"boolean"`**, **`"number"`**, or **`"string"`**.

11.4.4 Prefix Increment Operator

Runtime Semantics: Evaluation

UnaryExpression **`:`** **`++`** *UnaryExpression*

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* be ToNumber(GetValue(*expr*)).
3. ReturnIfAbrupt(*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. Let *status* be PutValue(*expr*, *newValue*).
6. ReturnIfAbrupt(*status*).
7. 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. ReturnIfAbrupt(*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. Let *status* be PutValue(*expr*, *newValue*).
6. ReturnIfAbrupt(*status*).
7. 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**.

Runtime Semantics: Evaluation

UnaryExpression **: –** *UnaryExpression*

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* be ToNumber(GetValue(*expr*)).
3. ReturnIfAbrupt(*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* be ToInt32(GetValue(*expr*)).
3. ReturnIfAbrupt(*oldValue*).
4. Return the result of applying bitwise complement to *oldValue*. The result is a signed 32-bit integer.

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. ReturnIfAbrupt(*oldValue*).
4. If *oldValue* is **true**, return **false**.
5. Return **true**.

11.5 Multiplicative Operators

Syntax

MultiplicativeExpression **:**
UnaryExpression
MultiplicativeExpression ***** *UnaryExpression*
MultiplicativeExpression **/** *UnaryExpression*
MultiplicativeExpression **%** *UnaryExpression*

Static Semantics: IsValidSimpleAssignmentTarget

MultiplicativeExpression **:**
MultiplicativeExpression ***** *UnaryExpression*
MultiplicativeExpression **/** *UnaryExpression*
MultiplicativeExpression **%** *UnaryExpression*

1. Return **false**.

Runtime Semantics: Evaluation

The production *MultiplicativeExpression* **:** *MultiplicativeExpression* @ *UnaryExpression*, where @ stands for one of the operators in the above definitions, is evaluated as follows:

1. Let *left* be the result of evaluating *MultiplicativeExpression*.
2. Let *leftValue* be GetValue(*left*).
3. ReturnIfAbrupt(*leftValue*).
4. Let *right* be the result of evaluating *UnaryExpression*.
5. Let *rightValue* be GetValue(*right*).
6. Let *lnum* be ToNumber(*leftValue*).
7. ReturnIfAbrupt(*lnum*).
8. Let *rnum* be ToNumber(*rightValue*).
9. ReturnIfAbrupt(*rnum*).
10. Return the result of applying the specified operation (*, /, or %) to *lnum* and *rnum*. 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 × 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*

Static Semantics: IsValidSimpleAssignmentTarget

AdditiveExpression **:**
AdditiveExpression **+** *MultiplicativeExpression*
AdditiveExpression **–** *MultiplicativeExpression*

1. Return **false**.

11.6.1 The Addition operator (+)

NOTE The addition operator either performs string concatenation or numeric addition.

Runtime Semantics: Evaluation

AdditiveExpression **:** *AdditiveExpression* **+** *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *MultiplicativeExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lprim* be ToPrimitive(*lval*).
8. ReturnIfAbrupt(*lprim*).
9. Let *rprim* be ToPrimitive(*rval*).
10. ReturnIfAbrupt(*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 standard 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. Exotic 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.

11.6.2 The Subtraction Operator (–)

Runtime Semantics: Evaluation

AdditiveExpression **:** *AdditiveExpression* **–** *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *MultiplicativeExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lnum* be ToNumber(*lval*).
8. ReturnIfAbrupt(*lnum*).
9. Let *rnum* be ToNumber(*rval*).
10. ReturnIfAbrupt(*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*

Static Semantics: IsValidSimpleAssignmentTarget

ShiftExpression **:**
ShiftExpression **<<** *AdditiveExpression*
ShiftExpression **>>** *AdditiveExpression*
ShiftExpression **>>>** *AdditiveExpression*

1. Return **false**.

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.

Runtime Semantics: Evaluation

ShiftExpression **:** *ShiftExpression* **<<** *AdditiveExpression*

1. Let *lref* be the result of evaluating *ShiftExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *AdditiveExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lnum* be ToInt32(*lval*).
8. ReturnIfAbrupt(*lnum*).
9. Let *rnum* be ToUint32(*rval*).
10. ReturnIfAbrupt(*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.

Runtime Semantics: Evaluation

ShiftExpression **:** *ShiftExpression* **>>** *AdditiveExpression*

1. Let *lref* be the result of evaluating *ShiftExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *AdditiveExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lnum* be ToInt32(*lval*).
8. ReturnIfAbrupt(*lnum*).
9. Let *rnum* be ToUint32(*rval*).
10. ReturnIfAbrupt(*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.

Runtime Semantics: Evaluation

ShiftExpression **:** *ShiftExpression* **>>>** *AdditiveExpression*

1. Let *lref* be the result of evaluating *ShiftExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *AdditiveExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lnum* be ToUint32(*lval*).
8. ReturnIfAbrupt(*lnum*).
9. Let *rnum* be ToUint32(*rval*).
10. ReturnIfAbrupt(*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.

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.

Static Semantics: IsValidSimpleAssignmentTarget

RelationalExpression **:**
RelationalExpression **<** *ShiftExpression*
RelationalExpression **>** *ShiftExpression*
RelationalExpression **<=** *ShiftExpression*
RelationalExpression **>=** *ShiftExpression*
RelationalExpression **instanceof** *ShiftExpression*
RelationalExpression **in** *ShiftExpression*

1. Return **false**.

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. ReturnIfAbrupt(*x*).
2. ReturnIfAbrupt(*y*).
3. If the *LeftFirst* flag is **true**, then
    a. Let *px* be the result of calling ToPrimitive(*x*, hint Number).
    b. ReturnIfAbrupt(*px*).
    c. Let *py* be the result of calling ToPrimitive(*y*, hint Number).
    d. ReturnIfAbrupt(*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. ReturnIfAbrupt(*py*).
    c. Let *px* be the result of calling ToPrimitive(*x*, hint Number).
    d. ReturnIfAbrupt(*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 **+∞**, return **false**.
    i. If *ny* is **+∞**, return **true**.
    j. If *ny* is **−∞**, return **false**.
    k. If *nx* is **−∞**, 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.

Runtime Semantics: Evaluation

RelationalExpression **:** *RelationalExpression* **<** *ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*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. ReturnIfAbrupt(*r*).
8. If *r* is **undefined**, return **false**. Otherwise, return *r*.

RelationalExpression **:** *RelationalExpression* **>** *ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*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. ReturnIfAbrupt(*r*).
8. If *r* is **undefined**, return **false**. Otherwise, return *r*.

RelationalExpression **:** *RelationalExpression* **<=** *ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*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. ReturnIfAbrupt(*r*).
8. If *r* is **true** or **undefined**, return **false**. Otherwise, return **true**.

RelationalExpression **:** *RelationalExpression* **>=** *ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*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. ReturnIfAbrupt(*r*).
8. If *r* is **true** or **undefined**, return **false**. Otherwise, return **true**.

*RelationalExpression***:** *RelationalExpression* **`instanceof`** *ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *ShiftExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Return the result of calling the instanceOfOperator abstract ooperator with arguments *rval* and *lval*.

The abstract operation HasInstanceOperator implements the generic algorithm for determining if an object *O* inherits from the inheritance path defined by constructor *C*. This abstract operation performs, the following steps:

1. If Type(*C*) is not Object, throw a **TypeError** exception.
2. Let *instOfHandler* be the result of GetMethod(*C*,@@hasInstance).
3. ReturnIfAbrupt(*instOfHandler*).
4. If *instOfHandler* is not **undefined**, then
    a. Return the result of calling the [[Call]] internal method of *instOfHandler* passing *C* as *thisArgument* and a new List containing *O* as *argumentsList*.
5. Return the result of OrdinaryHasInstance(*C*, *O*).

RelationalExpression **:** *RelationalExpression* **`in`** *ShiftExpression*

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *ShiftExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. If Type(*rval*) is not Object, throw a **TypeError** exception.
8. Return the result of HasProperty(*rval*, ToPropertyKey(*lval*)).

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.

Static Semantics: IsValidSimpleAssignmentTarget

EqualityExpression **:**
EqualityExpression **==** *RelationalExpression*
EqualityExpression **!=** *RelationalExpression*
EqualityExpression **===** *RelationalExpression*
EqualityExpression **!==** *RelationalExpression*
EqualityExpression [no *LineTerminator* here] **is** *RelationalExpression*
EqualityExpression [no *LineTerminator* here] **isnt** *RelationalExpression*

1. Return **false**.

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 Type(*x*) is the same as Type(*y*), then
   a. Return the result of performing strict equality comparison algorithm *x* === *y*.
2. If *x* is **null** and *y* is **undefined**, return **true**.
3. If *x* is **undefined** and *y* is **null**, return **true**.
4. If Type(*x*) is Number and Type(*y*) is String,
   return the result of the comparison *x* == ToNumber(*y*).
5. If Type(*x*) is String and Type(*y*) is Number,
   return the result of the comparison ToNumber(*x*) == *y*.
6. If Type(*x*) is Boolean, return the result of the comparison ToNumber(*x*) == *y*.
7. If Type(*y*) is Boolean, return the result of the comparison *x* == ToNumber(*y*).
8. If Type(*x*) is either String or Number and Type(*y*) is Object,
   return the result of the comparison *x* == ToPrimitive(*y*).
9. If Type(*x*) is Object and Type(*y*) is either String or Number,
   return the result of the comparison ToPrimitive(*x*) == *y*.
10. 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 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 Type(*x*) is different from Type(*y*), return **false**.
2. If Type(*x*) is Undefined, return **true**.
3. If Type(*x*) is Null, return **true**.
4. If 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**.
5. If Type(*x*) is String, then
    a. If *x* and *y* are exactly the same sequence of characters (same length and same characters in corresponding positions), return **true**.
    b. Else, return **false**.
6. If Type(*x*) is Boolean, then
    a. If *x* and *y* are both **true** or both **false**, return **true**.
    b. Else, return **false**.
7. If *x* and *y* are the same Object value, return **true**.
8. Return **false**.

NOTE This algorithm differs from the SameValue Algorithm (9.12) in its treatment of signed zeroes and NaNs.

Runtime Semantics: Evaluation

EqualityExpression : *EqualityExpression* **==** *RelationalExpression*

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *RelationalExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Return the result of performing abstract equality comparison algorithm *rval* == *lval*.

EqualityExpression **:** *EqualityExpression* **!=** *RelationalExpression*

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *RelationalExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*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**.

EqualityExpression **:** *EqualityExpression* **===** *RelationalExpression*

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*)
4. Let *rref* be the result of evaluating *RelationalExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Return the result of performing the strict equality comparison algorithm *rval* === *lval*.

EqualityExpression **:** *EqualityExpression* **!==** *RelationalExpression*

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).

4. Let *rref* be the result of evaluating *RelationalExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *r* be the result of performing strict equality comparison algorithm *rval* === *lval*.
8. If *r* is **true**, return **false**. Otherwise, return **true**.

EqualityExpression **:** *EqualityExpression* [no *LineTerminator* here] **`is`** *RelationalExpression*

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *RelationalExpression*.
5. Let *rval* be GetValue(*rref*).
6. Return the result of performing SameValue(*rval*, *lval*).

EqualityExpression **:** *EqualityExpression* [no *LineTerminator* here] **`isnt`** *RelationalExpression*

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*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. ReturnIfAbrupt(*r*).
8. If *r* is **true**, return **false**. Otherwise, return **true**.

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*

Static Semantics: IsValidSimpleAssignmentTarget

BitwiseANDExpression **:** *BitwiseANDExpression* **&** *EqualityExpression*
BitwiseXORExpression **:** *BitwiseXORExpression* **^** *BitwiseANDExpression*
BitwiseORExpression **:** *BitwiseORExpression* **|** *BitwiseXORExpression*

1. Return **false**.

Runtime Semantics: Evaluation

The production *A* **:** *A* @ *B*, where @ is one of the bitwise operators in the productions above, is evaluated as follows:

1. Let *lref* be the result of evaluating *A*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *B*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lnum* be ToInt32(*lval*).
8. ReturnIfAbrupt(*lnum*).
9. Let *rnum* be ToInt32(*rval*).
10. ReturnIfAbrupt(*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.

Static Semantics: IsValidSimpleAssignmentTarget

LogicalANDExpression **:** *LogicalANDExpression* **&&** *BitwiseORExpression*
LogicalORExpression **:** *LogicalORExpression* **||** *LogicalANDExpression*

1. Return **false**.

Runtime Semantics: Evaluation

LogicalANDExpression **:** *LogicalANDExpression* **&&** *BitwiseORExpression*

1. Let *lref* be the result of evaluating *LogicalANDExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *lbool* be ToBoolean(*lval*).
4. ReturnIfAbrupt(*lbool*).
5. If *lbool* is **false**, return *lval*.
6. Let *rref* be the result of evaluating *BitwiseORExpression*.
7. Return GetValue(*rref*).

LogicalORExpression **:** *LogicalORExpression* **||** *LogicalANDExpression*

1. Let *lref* be the result of evaluating *LogicalORExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *lbool* be ToBoolean(*lval*).
4. ReturnIfAbrupt(*lbool*).
5. If *lbool* is **true**, return *lval*.
6. Let *rref* be the result of evaluating *LogicalANDExpression*.
7. Return GetValue(*rref*).

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.

Static Semantics: IsValidSimpleAssignmentTarget

ConditionalExpression **:** *LogicalORExpression* **?** *AssignmentExpression* **:** *AssignmentExpression*

1. Return **false**.

Runtime Semantics: Evaluation

ConditionalExpression **:** *LogicalORExpression* **?** *AssignmentExpression* **:** *AssignmentExpression*

1. Let *lref* be the result of evaluating *LogicalORExpression*.
2. Let *lval* be ToBoolean(GetValue(*lref*)).
3. ReturnIfAbrupt(*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.

Static Semantics

Static Semantics: Early Errors

AssignmentExpression **:** *LeftHandSideExpression* **=** *AssignmentExpression*

- It is a Syntax Error if *LeftHandSideExpression* is either an *ObjectLiteral* or an *ArrayLiteral* and if the lexical token sequence matched by *LeftHandSideExpression* cannot be parsed with no tokens left over using *AssignmentPattern* as the goal symbol.
- If *LeftHandSideExpression* is either an *ObjectLiteral* or an *ArrayLiteral* and if the lexical token sequence matched by *LeftHandSideExpression* can be parsed with no tokens left over using *AssignmentPattern* as the goal symbol then the following rules are not applied. Instead, the Early Error rules for *AssignmentPattern* are used.
- It is a Syntax Error if *LeftHandSideExpression* is an *Identifier* that can be statically determined to always resolve to a declarative environment record binding and the resolved binding is an immutable binding.
- It is an early Reference Error if *LeftHandSideExpression* is neither an *ObjectLiteral* nor an *ArrayLiteral* and IsValidSimpleAssignmentTarget of *LeftHandSideExpression* is **false**.

AssignmentExpression **:** *LeftHandSideExpression AssignmentOperator AssignmentExpression*

- It is a Syntax Error if the *LeftHandSideExpression* is an *Identifier* that can be statically determined to always resolve to a declarative environment record binding and the resolved binding is an immutable binding.
- It is an early Reference Error if IsValidSimpleAssignmentTarget of *LeftHandSideExpression* is **false**.

Static Semantics: IsValidSimpleAssignmentTarget

AssignmentExpression **:**
YieldExpression
ArrowFunction
LeftHandSideExpression **=** *AssignmentExpression*
LeftHandSideExpression AssignmentOperator AssignmentExpression

1. Return **false**.

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. ReturnIfAbrupt(*lref*).
   c. Let *rref* be the result of evaluating *AssignmentExpression*.
   d. Let *rval* be GetValue(*rref*).
   e. Let *status* be PutValue(*lref*, *rval*).
   f. ReturnIfAbrupt(*status*).
   g. 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. ReturnIfAbrupt(*rval*).
6. Let *status* be the result of performing Destructuring Assignment Evaluation of *AssignmentPattern* using *rval* as the argument.
7. ReturnIfAbrupt(*status*).
8. Return *rval*.

AssignmentExpression **:** *LeftHandSideExpression AssignmentOperator AssignmentExpression*

1. Let *lref* be the result of evaluating *LeftHandSideExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *AssignmentExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *operator* be the `@` where *AssignmentOperator* is `@=`
8. Let *r* be the result of applying operator @ to *lval* and *rval*.
9. Let *status* be PutValue(*lref*, *r*).
10. ReturnIfAbrupt(*status*).
11. 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 it is an unresolvable 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 property with the attribute value {[[Set]]:**undefined**}, nor to a non-existent property of an object where calling its [[GetExtensible]] internal method returns the value **false**. In these cases a **TypeError** exception is thrown.

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*$_{opt}$ *AssignmentRestElement*$_{opt}$ **]**
[*AssignmentElementList* **]**
[*AssignmentElementList* **,** *Elision*$_{opt}$ *AssignmentRestElement*$_{opt}$ **]**

AssignmentPropertyList **:**
AssignmentProperty
AssignmentPropertyList **,** *AssignmentProperty*

AssignmentElementList **:**
Elision$_{opt}$ *AssignmentElement*
AssignmentElementList **,** *Elision*$_{opt}$ *AssignmentElement*

AssignmentProperty **:**
Identifier Initialiser$_{opt}$
PropertyName **:** *AssignmentElement*

AssignmentElement **:**
DestructuringAssignmentTarget Initialiser$_{opt}$

AssignmentRestElement **:**
... *DestructuringAssignmentTarget*

DestructuringAssignmentTarget **:**
LeftHandSideExpression

Static Semantics

Static Semantics: Early Errors

AssignmentProperty **:** *Identifier Initialiser*$_{opt}$

- It is a Syntax Error if *Identifier* is the *Identifier* **`eval`** or the *Identifier* **`arguments`**.
- It is a Syntax Error if *Identifier* does not statically resolve to a declarative environment record binding or if the resolved binding is an immutable binding.

DestructuringAssignmentTarget **:** *LeftHandSideExpression*

- It is a Syntax Error *LeftHandSideExpression* is either an *ObjectLiteral* or an *ArrayLiteral* and if the lexical token sequence matched by *LeftHandSideExpression* cannot be parsed with no tokens left over using *AssignmentPattern* as the goal symbol.
- It is a Syntax Error if *LeftHandSideExpression* is neither an *ObjectLiteral* nor an *ArrayLiteral* and IsValidSimpleAssignmentTarget of *LeftHandSideExpression* is **false**.
- It is a Syntax Error if the *LeftHandSideExpression* is an *Identifier* that can be statically determined to always resolve to a declarative environment record binding and the resolved binding is an immutable binding.
- It is a Syntax Error if *LeftHandSideExpression* is the *Identifier* **`eval`** or the *Identifier* **`arguments`**.
- It is a Syntax Error if IsInvalidAssignmentPattern of *LeftHandSideExpression* is **true**.

- It is a Syntax Error if the *LeftHandSideExpression* is *CoverParenthesizedExpressionAndArrowParameterList* : **(** *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*

ObjectAssignmentPattern **: { }**
and
ArrayAssignmentPattern **:**
[ ]
[*Elision* **]**

1. Return NormalCompletion(empty).

AssignmentPropertyList **:** *AssignmentPropertyList* **,** *AssignmentProperty*

1. Let *status* be the result of performing Destructuring Assignment Evaluation for *AssignmentPropertyList* using *obj* as the argument.
2. ReturnIfAbrupt(*status*).
3. Return the result of performing Destructuring Assignment Evaluation for *AssignmentProperty* using *obj* as the argument.

AssignmentProperty **:** *Identifier Initialiser*$_{opt}$

1. Let *P* be StringValue of *Identifier*.
2. Let *v* be the result of calling Get(*obj*, *P*).
3. ReturnIfAbrupt(*v*).
4. If *Initialiser*$_{opt}$ is present and *v* is **undefined**, then
    a. Let *defaultValue* be the result of evaluating *Initialiser*.
    b. Let *v* be ToObject(*defaultValue*).
5. ReturnIfAbrupt(*v*).
6. Let *lref* be the result of performing Identifier Resolution(10.3.1) with the *IdentifierName* corresponding to *Identifier*.
7. Return PutValue(*lref*,*v*).

AssignmentProperty **:** *PropertyName* **:** *AssignmentElement*

1. Let *name* be PropName of *PropertyName*.
2. Return the result of performing Keyed Destructuring Assignment Evaluation of *AssignmentElement* with *obj* and *name* as the arguments.

ArrayAssignmentPattern **: [** *Elision*$_{opt}$ *AssignmentRestElement* **]**

1. Let *skip* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
2. Return the result of performing Indexed Destructuring Assignment Evaluation of *AssignmentRestElement* with *obj* and *skip* as the arguments.

ArrayAssignmentPattern **: [** *AssignmentElementList* **]**

1. Return the result of performing Indexed Destructuring Assignment Evaluation of *AssignmentElementList* using *obj* and 0 as the arguments.

ArrayAssignmentPattern **: [** *AssignmentElementList* **,** *Elision*$_{opt}$ *AssignmentRestElement*$_{opt}$ **]**

1. Let *lastIndex* be the result of performing Indexed Destructuring Assignment Evaluation of *AssignmentElementList* using *obj* and 0 as the arguments.
2. ReturnIfAbrupt(*lastIndex*).
3. Let *skip* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
4. If *AssignmentRestElement* is present, then return the result of performing Indexed Destructuring Assignment Evaluation of *AssignmentRestElement* with *obj* and *lastIndex*+*skip* as the arguments.
5. Return *lastIndex*.

Runtime Semantics: Indexed Destructuring Assignment Evaluation

with parameters *obj* and *index*

AssignmentElementList **:** *Elision*$_{opt}$ *AssignmentElement*

1. Let *skip* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
2. Let *name* be ToString(*index*+*skip*).
3. Let *status* be the result of performing Keyed Destructuring Assignment Evaluation of *AssignmentElement* with *obj* and *name* as the arguments.
4. ReturnIfAbrupt(*status*).
5. Return *index*+*skip*+1.

AssignmentElementList **:** *AssignmentElementList* **,** *Elision*$_{opt}$ *AssignmentElement*

1. Let *listNext* be the result of performing Indexed Destructuring Assignment Evaluation of *AssignmentElementList* using *obj* as the *obj* parameter and *index* as the *index* parameter
2. Let *skip* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
3. ReturnIfAbrupt(*listNext*).
4. Let *name* be ToString(*listNext*+*skip*).
5. Let *status* be the result of performing Keyed Destructuring Assignment Evaluation of *AssignmentElement* with *obj* and *name* as the arguments.
6. ReturnIfAbrupt(*status*).
7. Return *listNext*+*skip*+1.

AssignmentRestElement **: ...** *DestructuringAssignmentTarget*

1. Let *lref* be the result of evaluating *DestructuringAssignmentTarget*.
2. ReturnIfAbrupt(*lref*).
3. Let *lenVal* be the result of Get(*obj*, **`"length"`**).
4. Let *len* be ToUint32(*lenVal*).
5. ReturnIfAbrupt(*len*).
6. Let *A* be the result of the abstract operation ArrayCreate 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 HasProperty(*obj*, *P*).
    c. ReturnIfAbrupt(*exists*).
    d. If *exists* is **true**, then
        i. Let *v* be the result of Get(*obj*, ToString(*index*)).
        ii. ReturnIfAbrupt(*len*).
        iii. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*n*) and Property Descriptor {[[Value]]: *v*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
    e. Let *n* = *n*+1.
    f. Let *index* = *index*+1.
9. Return PutValue(*lref*,*A*).

Runtime Semantics: Keyed Destructuring Assignment Evaluation

with parameters *obj* and *propertyName*

AssignmentElement **:** *DestructuringAssignmentTarget* *Initialiser*$_{opt}$

1. Let *v* be the result of Get(*obj*, *propertyName*).
2. ReturnIfAbrupt(*v*).
3. If *Initialiser*$_{opt}$ is present and *v* is **undefined**, then
    a. Let *v* be the result of evaluating *Initialiser*.
4. If *DestructuringAssignmentTarget* is an *ObjectLiteral* or an *ArrayLiteral* then
    a. Let *AssignmentPattern* be the parse of the source code corresponding to *DestructuringAssignmentTarget* using *AssignmentPattern* as the goal symbol
    b. Let *vObj* be ToObject(*v*).
    c. ReturnIfAbrupt(*vObj*).
    d. Return the result of performing Destructuring Assignment Evaluation of *AssignmentPattern* with *vObj* as the argument.
5. ReturnIfAbrupt(*v*).
6. Let *lref* be the result of evaluating *DestructuringAssignmentTarget*.
7. Return PutValue(*lref*,*v*).

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.

Static Semantics: IsValidSimpleAssignmentTarget

Expression **:** *Expression* **,** *AssignmentExpression*

1. Return **false**.

Runtime Semantics: Evaluation

Expression **:** *Expression* **,** *AssignmentExpression*

1. Let *lref* be the result of evaluating *Expression*.
2. ReturnIfAbrupt(GetValue(*lref*))
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.

12 Statements and Declarations

Syntax

Statement **:**
BlockStatement
VariableStatement
EmptyStatement
ExpressionStatement
IfStatement
BreakableStatement
ContinueStatement
BreakStatement
ReturnStatement
WithStatement
LabelledStatement
ThrowStatement
TryStatement
DebuggerStatement

Declaration **:**
FunctionDeclaration
GeneratorDeclaration
ClassDeclaration
LexicalDeclaration

BreakableStatement **:**
IterationStatement
SwitchStatement

Static Semantics

Static Semantics: VarDeclaredNames

Statement **:**
EmptyStatement
ExpressionStatement
ContinueStatement
BreakStatement
ReturnStatement
ThrowStatement
DebuggerStatement

1. Return a new empty List.

Runtime Semantics

Runtime Semantics: Labelled Evaluation

With argument *labelSet*.

BreakableStatement **:** *IterationStatement*

1. Let *stmtResult* be the result of performing Labelled Evaluation of *IterationStatement* with 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 exited via an unlabelled *BreakStatement*.

Runtime Semantics: Evaluation

BreakableStatement **:**
IterationStatement
SwitchStatement

1. Let *newLabelSet* be a new empty List.
2. Return the result of performing Labelled Evaluation of this *BreakableStatement* with argument *newLabelSet*.

12.1 Block

Syntax

BlockStatement **:**
Block

Block **:**
{ *StatementList*$_{opt}$ **}**

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 **:** *StatementList StatementListItem*

1. Let *declarations* be LexicalDeclarations of *StatementList*.
2. Append to *declarations* the elements of the LexicalDeclarations of *StatementListItem*.
3. Return *declarations*.

StatementListItem **:** *Statement*

1. Return a new empty List.

StatementListItem **:** *Declaration*

1. Return a new List containing *Declaration*.

Static Semantics: LexicallyDeclaredNames

Block **:** **{ }**

1. Return a new empty List.

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: TopLevelLexicallyDeclaredNames

StatementList **:** *StatementList StatementListItem*

1. Let *names* be TopLevelLexicallyDeclaredNames of *StatementList*.
2. Append to *names* the elements of the TopLevelLexicallyDeclaredNames of *StatementListItem*.
3. Return *names*.

StatementListItem **:** *Statement*

1. Return a new empty List.

StatementListItem **:** *Declaration*

1. If *Declaration* is *Declaration* **:** *FunctionDeclaration*, then return a new empty List.
2. Return the BoundNames of *Declaration*.

NOTE At the top level of a function, or script, function declarations are treated like var declarations rather than like lexical declarations.

Static Semantics: TopLevelLexicallyScopedDeclarations

StatementList **:** *StatementList StatementListItem*

1. Let *declarations* be TopLevelLexicallyScopedDeclarations of *StatementList*.
2. Append to *declarations* the elements of the TopLevelLexicallyScopedDeclarations of *StatementListItem*.
3. Return *declarations*.

StatementListItem **:** *Statement*

1. Return a new empty List.

StatementListItem **:** *Declaration*

1. If *Declaration* is *Declaration* **:** *FunctionDeclaration*, then return a new empty List.
2. Return a new List containing *Declaration*.

Static Semantics: TopLevelVarDeclaredNames

StatementList **:** *StatementList StatementListItem*

1. Let *names* be TopLevelVarDeclaredNames of *StatementList*.
2. Append to *names* the elements of the TopLevelVarDeclaredNames of *StatementListItem*.
3. Return *names*.

StatementListItem **:** *Declaration*

1. If *Declaration* is *Declaration* **:** *FunctionDeclaration*, then return the LexicallyDeclaredNames of *Declaration*.
2. Return a new empty List.

StatementListItem **:** *Statement*

1. Return *VarDeclaredNames* of *Statement*.

NOTE At the top level of a function or script, inner function declarations are treated like var declarations.

Static Semantics: TopLevelVarScopedDeclarations

StatementList **:** *StatementList StatementListItem*

1. Let *declarations* be TopLevelVarScopedDeclarations of *StatementList*.
2. Append to *declarations* the elements of the TopLevelVarScopedDeclarations of *StatementListItem*.
3. Return *declarations*.

StatementListItem **:** *Statement*

1. If *Statement* is *Statement* **:** *VariableStatement*, then return a new List containing *VariableStatement*.
2. Return a new empty List.

StatementListItem **:** *Declaration*

1. If *Declaration* is *Declaration* **:** *FunctionDeclaration*, then return a new List containing *Declaration*.
2. Return a new empty List.

Static Semantics: VarDeclaredNames

Block **: { }**

1. Return a new empty List.

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 **:** *Declaration*

2. Return a new empty List.

Runtime Semantics

Runtime Semantics: Evaluation

Block **: { }**

1. Return NormalCompletion(empty).

Block **: {** *StatementList* **}**

1. Let *oldEnv* be the running 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 running execution context's LexicalEnvironment to *blockEnv*.
5. Let *blockValue* be the result of evaluating *StatementList*.
6. Set the running 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 **:** *StatementList StatementListItem*

1. Let *sl* be the result of evaluating *StatementList*.
2. ReturnIfAbrupt(*sl*).
3. Let *s* be the result of evaluating *StatementListItem*.
4. If *s*.[[type]] is throw, return *s*.
5. If *s*.[[value]] is empty, let *V* = *sl*.[[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 algoritm 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;")
```

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 running 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 *LexicalBinding* is evaluated, not when the variable is created. If a *LexicalBinding* in a **`let`** declaration does not have an an *Initialiser* the variable is assigned the value **undefined** when the *LexicalBinding* is evaluated.

Syntax

LexicalDeclaration **:**
LetOrConst BindingList **`;`**

LexicalDeclarationNoIn **:**
LetOrConst BindingListNoIn

LetOrConst **:**
`let`
`const`

BindingList **:**
LexicalBinding
BindingList **`,`** *LexicalBinding*

BindingListNoIn **:**
LexicalBindingNoIn
BindingListNoIn **`,`** *LexicalBindingNoIn*

LexicalBinding **:**
BindingIdentifier Initialiser$_{opt}$
BindingPattern Initialiser

LexicalBindingNoIn **:**
BindingIdentifier InitialiserNoIn$_{opt}$
BindingPattern InitialiserNoIn

BindingIdentifier **:**
Identifier

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: BoundNames

LexicalDeclaration **:** *LetOrConst BindingList* **;**

1. Return the BoundNames of *BindingList*.

BindingList **:** *BindingList* **,** *LexicalBinding*

1. Let *names* be the BoundNames of *BindingList*.
2. Append to *names* the elements of the BoundNames of *LexicalBinding*.
3. Return *names*.

LexicalBinding **:** *BindingIdentifier Initialiser*$_{opt}$

1. Return the BoundNames of *BindingIdentifier*.

*LexicalBinding***:** *BindingPattern Initialiser*

1. Return the BoundNames of *BindingPattern*.

BindingIdentifier **:** *Identifier*

1. Return a new List containing the StringValue of *Identifier*.

Static Semantics: IsConstantDeclaration

LexicalDeclaration **:** *LetOrConst BindingList* **;**

1. Return IsConstantDeclaration of *LetOrConst*.

LetOrConst **:** **`let`**

1. Return **false**.

LetOrConst **:** **`const`**

1. Return **true**.

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. If *environment* is not **undefined**, then
    a. Let *name* be StringValue of *Identifier*.
    b. Let *env* be the environment record component of *environment*.
    c. Call the InitializeBinding concrete method of *env* passing *name* and *value* as the arguments.
    d. Return NormalCompletion(undefined).
2. Else
    a. Let *lhs* be the result of evaluating *Identifier* as described in 11.1.2.
    b. Return PutValue(*lhs*, *value*).

Runtime Semantics: Evaluation

LexicalDeclaration **:** *LetOrConst BindingList* **`;`**

1. Let *next* be the result of evaluating *BindingList*.
2. ReturnIfAbrupt(*next*).
3. Return NormalCompletion(empty).

BindingList **:** *BindingList* **`,`** *LexicalBinding*

1. Let *next* be the result of evaluating *BindingList*.
2. ReturnIfAbrupt(*next*).
3. Return the result of evaluating *LexicalBinding*.

LexicalBinding **:** *BindingIdentifier*

1. Let *env* be the running execution context's LexicalEnvironment.
2. Return the result of performing Binding Initialisation for *BindingIdentifier* passing **undefined** and *env* as the arguments.

NOTE A static semantics rule ensures that this form of *LexicalBinding* never occurs in a **`const`** declaration.

LexicalBinding **:** *BindingIdentifier Initialiser*

1. Let *rhs* be the result of evaluating *Initialiser*.
2. Let *value* be GetValue(*rhs*).
3. ReturnIfAbrupt(*value*).
4. Let *env* be the running execution context's LexicalEnvironment.
5. Return the result of performing Binding Initialisation for *BindingIdentifier* passing *value* and *env* as the arguments.

*LexicalBinding***:** *BindingPattern Initialiser*

1. Let *rhs* be the result of evaluating *Initialiser*.
2. Let *value* be ToObject(GetValue(*rhs*)).
3. ReturnIfAbrupt(*value*).
4. Let *env* be the running execution context's LexicalEnvironment.
5. Return the result of performing Binding Initialisation for *BindingPattern* using *value* and *env* as the *arguments*.

12.2.2 Variable Statement

NOTE A **`var`** statement declares variables that are scoped to the running 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 VariableEnvironemnt 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.

Syntax

VariableStatement **:**
`var` *VariableDeclarationList* `;`

VariableDeclarationList **:**
VariableDeclaration
VariableDeclarationList `,` *VariableDeclaration*

VariableDeclarationListNoIn **:**
VariableDeclarationNoIn
VariableDeclarationListNoIn `,` *VariableDeclarationNoIn*

VariableDeclaration **:**
BindingIdentifier Initialiser$_{opt}$
BindingPattern Initialiser

VariableDeclarationNoIn **:**
BindingIdentifier InitialiserNoIn$_{opt}$
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

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*$_{opt}$

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*.

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.

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. ReturnIfAbrupt(*next*).
3. Return NormalCompletion(empty).

VariableDeclarationList **:** *VariableDeclarationList* **`,`** *VariableDeclaration*

1. Let *next* be the result of evaluating *VariableDeclarationList*.
2. ReturnIfAbrupt(*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. ReturnIfAbrupt(*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 ToObject(GetValue(*rhs*)).
4. ReturnIfAbrupt(*rval*).

5. Return the result of performing Binding Initialisation for *BindingPattern* passing *rval* and **undefined** as arguments.

12.2.4 Destructuring Binding Patterns

Syntax

BindingPattern **:**
ObjectBindingPattern
ArrayBindingPattern

ObjectBindingPattern **:**
{ }
{ *BindingPropertyList* **}**
{ *BindingPropertyList* **, }**

ArrayBindingPattern **:**
[$Elision_{opt}$ $BindingRestElement_{opt}$ **]**
[*BindingElementList* **]**
[*BindingElementList* **,** $Elision_{opt}$ $BindingRestElement_{opt}$ **]**

BindingPropertyList **:**
BindingProperty
BindingPropertyList **,** *BindingProperty*

BindingElementList **:**
$Elision_{opt}$ *BindingElement*
BindingElementList **,** $Elision_{opt}$ *BindingElement*

BindingProperty **:**
SingleNameBinding
PropertyName **:** *BindingElement*

BindingElement **:**
SingleNameBinding
BindingPattern Initialiser$_{opt}$

SingleNameBinding **:**
BindingIdentifier Initialiser$_{opt}$

BindingRestElement **:**
... *BindingIdentifier*

Static Semantics

Static Semantics: Early Errors

BindingPattern **:** *ObjectBindingPattern*

- It is a Syntax Error if the BoundNames of *ObjectBindingPattern* contains the string "**`eval`**" or the string "**`arguments`**".

BindingPattern **:** *ArrayBindingPattern*

- It is a Syntax Error if the BoundNames of *ArrayBindingPattern* contains the string "**`eval`**" or the string "**`arguments`**".

Static Semantics: BoundNames

ObjectBindingPattern: **{ }**

1. Return an empty List.

ArrayBindingPattern : **[** *Elision*$_{opt}$ **]**

1. Return an empty List.

ArrayBindingPattern : **[** *Elision*$_{opt}$ *BindingRestElement* **]**

1. Return the BoundNames of *BindingRestElement*.

ArrayBindingPattern : **[** *BindingElementList* **,** *Elision*$_{opt}$ **]**

1. Return the BoundNames of *BindingElementList*.

ArrayBindingPattern : **[** *BindingElementList* **,** *Elision*$_{opt}$ *BindingRestElement* **]**

1. Let *names* be BoundNames of *BindingElementList*.
2. Append to *names* the elements of BoundNames of *BindingRestElement*.
3. Return *names*.

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*$_{opt}$ *BindingElement*

1. Return BoundNames of *BindingElement*.

BindingElementList : *BindingElementList* **,** *Elision*$_{opt}$ *BindingElement*

1. Let *names* be BoundNames of *BindingElementList*.
2. Append to *names* the elements of BoundNames of *BindingElement*.
3. Return *names*.

BindingProperty : *PropertyName* **:** *BindingElement*

1. Return the BoundNames of *BindingElement*.

SingleNameBinding : *BindingIdentifier Initialiser*$_{opt}$

1. Return the BoundNames of *BindingIdentifier*.

BindingElement : *BindingPattern Initialiser*$_{opt}$

1. Return the BoundNames of *BindingPattern*.

Static Semantics: HasInitialiser

BindingElement : *BindingPattern*

1. Return **false**.

BindingElement : *BindingPattern Initialiser*

1. Return **true**.

SingleNameBinding **:** *BindingIdentifier*

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. Assert: Type(*value*) is Object
2. Return the result of performing Binding Initialisation for *ObjectBindingPattern* using *value* and *environment* as arguments.

BindingPattern **:** *ArrayBindingPattern*

1. Assert: Type(*value*) is Object
2. Return the result of performing Indexed Binding Initialisation for *ArrayBindingPattern* using *value*, 0, and *environment* as arguments.

*ObjectBindingPattern***: { }**

1. Return NormalCompletion(empty).

BindingPropertyList **:** *BindingPropertyList* **,** *BindingProperty*

1. Let *status* be the result of performing Binding Initialisation for *BindingPropertyList* using *value* and *environment* as arguments.
2. ReturnIfAbrupt(*status*).
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*$_{opt}$ **]**

1. Return NormalCompletion(empty).

ArrayBindingPattern: **[** *Elision*$_{opt}$ *BindingRestElement* **]**

1. Let *nextIndex* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
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*$_{opt}$**]**

1. Return the result of performing Indexed Binding Initialisation for *BindingElementList* using *array*, *nextIndex*, and *environment* as arguments.

ArrayBindingPattern: **[** *BindingElementList* **,** *Elision*$_{opt}$ *BindingRestElement* **]**

1. Let *next* be the result of performing Indexed Binding Initialisation for *BindingElementList* using *array* , *nextIndex*, and *environment* as arguments.
2. ReturnIfAbrupt(*next*).
3. Let *skip* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
4. Return the result of performing Indexed Binding Initialisation for *BindingRestElement* using *array*, *next*+*skip* , and *environment* as arguments.

BindingElementList **:** *Elision*$_{opt}$ *BindingElement*

1. Let *skip* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
2. Let *status* be the result of performing Indexed Binding Initialisation for *BindingElement* using *array*, *nextIndex*+*skip* , and *environment* as arguments.
3. ReturnIfAbrupt(*status*).
4. Return *nextIndex* +*skip*+1.

BindingElementList **:** *BindingElementList* **,** *Elision*$_{opt}$ *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. Let *skip* be the Elision Width of *Elision*; if *Elision* is not present, use the numeric value zero.
4. Let *status* be the result of performing Indexed Binding Initialisation for *BindingElement* using *array*, *listNext*+*skip* , and *environment* as arguments.
5. ReturnIfAbrupt(*status*).
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*$_{opt}$

1. Let *P* be ToString(*nextIndex*).
2. Let *v* be the result of Get(*array*, *P*).
3. ReturnIfAbrupt(*v*).
4. If *Initialiser*$_{opt}$ is present and *v* is **undefined**, then
   a. Let *defaultValue* be the result of evaluating *Initialiser*.
   b. Let *v* be ToObject(*defaultValue*).
5. ReturnIfAbrupt(*v*).
6. Return the result of performing Binding Initialisation for *BindingPattern* passing *v* and *environment* as arguments.

BindingRestElement : **...** *BindingIdentifier*

2. Let *A* be the result of the abstract operation ArrayCreate with argument 0.
3. Let *lenVal* be the result of Get(*array*, **`"length"`**).
4. Let *arrayLength* be ToUint32(*lenVal*).
5. ReturnIfAbrupt(*arrayLength*).
6. Let *n*=0.
7. Let *index* = *nextIndex*.
8. Repeat, while *index* < *arrayLength*
   a. Let *P* be ToString(*index*).
   b. Let *exists* be the result of HasProperty(*array*, *P*).
   c. ReturnIfAbrupt(exists).
   d. If *exists* is **true**, then
      i. Let *v* be the result of Get(*array*, *P*).
      ii. ReturnIfAbrupt(*v*).
      iii. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*n*) and Property Descriptor {[[Value]]: *v*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
   e. Let *n* = *n*+1.
   f. Let *index* = *index*+1.
9. Return the result of performing Binding Initialisation for *BindingIdentifier* using *A* and *environment* as arguments.

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: *BindingPattern Initialiser*$_{opt}$

1. Let *v* be the result of Get(*obj*, *propertyName*).
2. ReturnIfAbrupt(*v*).
3. If *Initialiser*$_{opt}$ is present and *v* is **undefined**, then
   a. Let *defaultValue* be the result of evaluating *Initialiser*.
   b. Let *v* be ToObject(*defaultValue*).
4. ReturnIfAbrupt(*v*).
5. Return the result of performing Binding Initialisation for *BindingPattern* passing *v* and *environment* as arguments.

SingleNameBinding : *BindingIdentifier Initialiser*$_{opt}$

1. Let *v* be the result of Get(*obj*, *propertyName*).
2. ReturnIfAbrupt(*v*).
3. If *Initialiser*$_{opt}$ is present and *v* is **undefined**, then
   a. Let *v* be the result of evaluating *Initialiser*.
4. ReturnIfAbrupt(*v*).

5. Return the result of performing Binding Initialisation 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 ∉ {**{**, **function**, **class** }] *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** or **class** keywords because that would make it ambiguous with a *FunctionDeclaration*, a *GeneratorDeclaration*, or a *ClassDeclaration*.

Runtime Semantics

Runtime Semantics: Evaluation

ExpressionStatement **:** [lookahead ∉ {**{**, **function**, **class** }] *Expression* **;**

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *value* be GetValue(*exprRef*).
3. ReturnIfAbrupt(*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*.
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. ReturnIfAbrupt(*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. ReturnIfAbrupt(*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 ($ExpressionNoIn_{opt}$ **;** $Expression_{opt}$ **;** $Expression_{opt}$ **)** *Statement*
for (**var** *VariableDeclarationListNoIn* **;** $Expression_{opt}$ **;** $Expression_{opt}$ **)** *Statement*
for (*LexicalDeclarationNoIn* **;** $Expression_{opt}$ **;** $Expression_{opt}$ **)** *Statement*
for (*LeftHandSideExpression* **in** *Expression* **)** *Statement*
for (**var** *ForBinding* **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.

Runtime Semantics

Runtime Semantics: LoopContinues Abstract Operation

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**.

NOTE Within the *Statement* part of an *IterationStatement* a *ContinueStatement* may be used to begin a new iteration.

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

Static Semantics: VarDeclaredNames

IterationStatement : **for (***ExpressionNoIn*$_{opt}$ **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$**)** *Statement*

1. Return the VarDeclaredNames of *Statement*.

IterationStatement : **for (var** *VariableDeclarationListNoIn* **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*

1. Let *names* be BoundNames of *VariableDeclarationListNoIn*.
2. Append to *names* the elements of the VarDeclaredNames of *Statement*.
3. Return *names*.

IterationStatement : **for (***LexicalDeclarationNoIn* **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$**)** *Statement*

1. Return the VarDeclaredNames of *Statement*.

Runtime Semantics

Runtime Semantics: Labelled Evaluation

With argument *labelSet*.

IterationStatement : **for (***ExpressionNoIn*$_{opt}$ **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$**)** *Statement*

1. If *ExpressionNoIn* is present, then
    a. Let *exprRef* be the result of evaluating *ExpressionNoIn*.
    b. Let *exprValue* be GetValue(*exprRef*).
    c. If LoopContinues(*exprValue*,*labelSet*) is **false**, return *exprValue*.
2. Return the result of performing For Body Evaluation with the first *Expression* as the *testExpr* argument, the second *Expression* as the *incrementExpr* argument, *Statement* as the *stmt* argument, and with *labelSet*.

IterationStatement : **for (var** *VariableDeclarationListNoIn* **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *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 *testExpr* argument, the second *Expression* as the *incrementExpr* argument, *Statement* as the *stmt* argument, and with *labelSet*.

IterationStatement : **for (** *LexicalDeclarationNoIn* **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*

1. Let *oldEnv* be the running 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 *LexicalDeclarationNoIn*.
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 running execution context's LexicalEnvironment to *loopEnv*.
6. Let *forDcl* be the result of evaluating *LexicalDeclarationNoIn*.
7. If LoopContinues(*forDcl*,*labelSet*) is **false**, then
    a. Set the running execution context's LexicalEnvironment to *oldEnv*.
    b. Return *forDcl*.

8. Let *bodyResult* be the result of performing For Body Evaluation with the first *Expression* as the *testExpr* argument, the second *Expression* as the *incrementExpr* argument, *Statement* as the *stmt* argument, and with *labelSet*.
9. Set the running execution context's LexicalEnvironment to *oldEnv*.
10. Return *bodyResult*.

Runtime Semantics: For Body Evaluation Abstract Operation

The abstract operation For Body Evaluation with arguments *testExpr*, *incrementExpr*, *stmt*, 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 if LoopContinues (*testExprValue*,*labelSet*) is **false**, return *testExprValue*.
   b. Let *result* be the result of evaluating *stmt*.
   c. If *result*.[[value]] is not empty, let *V* = *result*.[[value]].
   d. If LoopContinues (*result*,*labelSet*) is **false**, return *result*.
   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*.

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 *LeftHandSideExpression* is either an *ObjectLiteral* or an *ArrayLiteral* and if the lexical token sequence matched by *LeftHandSideExpression* cannot be parsed with no tokens left over using *AssignmentPattern* as the goal symbol.
- If *LeftHandSideExpression* is either an *ObjectLiteral* or an *ArrayLiteral* and if the lexical token sequence matched by *LeftHandSideExpression* can be parsed with no tokens left over using *AssignmentPattern* as the goal symbol then the following rules are not applied. Instead, the Early Error rules for *AssignmentPattern* are used.
- It is a Syntax Error if the *LeftHandSideExpression* is an *Identifier* that can be statically determined to always resolve to a declarative environment record binding and the resolved binding is an immutable binding.
- It is a Syntax Error if *LeftHandSideExpression* is neither an *ObjectLiteral* nor an *ArrayLiteral* and IsValidSimpleAssignmentTarget of *LeftHandSideExpression* is **false**.
- It is a Syntax Error if the *LeftHandSideExpression* is *CoverParenthesizedExpressionAndArrowParameterList* : **`(`** *Expression* **`)`**
  and *Expression* derived a production that would produce a Syntax Error according to these rules. This rule is recursively applied.

IterationStatement **:**
`for (`*ForDeclaration* **`in`** *Expression* **`)`** *Statement*
`for (`*ForDeclaration* **`of`** *Expression* **`)`** *Statement*

- It is a Syntax Error if any element of the BoundNames of *ForDeclaration* also occurs in the VarDeclaredNames of *Statement*.

Static Semantics: BoundNames

ForDeclaration **:** *LetOrConst ForBinding*

1. Return the BoundNames of *ForBinding*.

Static Semantics: VarDeclaredNames

IterationStatement **: `for (`** *LeftHandSideExpression* **`in`** *Expression* **`)`** *Statement*

1. Return the VarDeclaredNames of *Statement*.

IterationStatement **: `for (` `var`** *ForBinding* **`in`** *Expression* **`)`** *Statement*

1. Let *names* be the BoundNames of *ForBinding*.
2. Append to *names* the elements of the VarDeclaredNames of *Statement*.
3. Return *names*

IterationStatement **: `for (`** *ForDeclaration* **`in`** *Expression* **`)`** *Statement*

1. Return the VarDeclaredNames of *Statement*.

IterationStatement **: `for (`** *LeftHandSideExpression* **`of`** *Expression* **`)`** *Statement*

1. Return the VarDeclaredNames of *Statement*.

IterationStatement **: `for (` `var`** *ForBinding* **`of`** *Expression* **`)`** *Statement*

1. Let *names* be the BoundNames of *ForBinding*.
2. Append to *names* the elements of the VarDeclaredNames of *Statement*.
3. Return *names*

IterationStatement **: `for (`** *ForDeclaration* **`of`** *Expression* **`)`** *Statement*

1. Return the VarDeclaredNames of *Statement*.

Runtime Semantics

Runtime Semantics: Binding Instantiation

With arguments *value* and *environment*.

ForDeclaration **:** *LetOrConst ForBinding*

1. For each element *name* of the BoundNames 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. Return the result of performing 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 Evaluation with *Statement*, enumerate, and *labelSet*.
2. ReturnIfAbrupt(*keyResult*).
3. Return the result of performing For In/Of Body Evaluation 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 Evaluation with *Statement*, enumerate, and *labelSet*.
2. ReturnIfAbrupt(*keyResult*).
3. Return the result of performing For In/Of Body Evaluation 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 Evaluation with *Statement*, enumerate, and *labelSet*.
2. ReturnIfAbrupt(*keyResult*).
3. Return the result of performing For In/Of Body Evaluation with *ForDeclaration*, *Statement*, *keyResult*, lexicalBinding, and *labelSet*.

IterationStatement **: for (** *LeftHandSideExpression* **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 *LeftHandSideExpression*, *Statement*, *keyResult*, assignment, and *labelSet*.

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*.

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*.

Runtime Semantics: For In/Of Expression Evaluation Abstract Operation

The abstract operation For In/Of Expression Evaluation is called with arguments *expr*, *iterationKind*, and *labelSet*. The value of *iterationKind* is either **enumerate** or **iterate**.

1. Let *exprRef* be the result of evaluating the production that is *expr*.
2. Let *experValue* be GetValue(*exprRef*).
3. If *experValue* is an abrupt completion,
    a. If LoopContinues(*experValue*,*labelSet*) is **false**, then return *experValue*.
    b. Else, return Completion {[[type]]: break, [[value]]: empty, [[target]]: empty}.
4. If *experValue*.[[value]] is **null** or **undefined**, return Completion {[[type]]: break, [[value]]: empty, [[target]]: empty}.
5. Let *obj* be ToObject(*experValue*).

6. If *iterationKind* is enumerate, then
    a. Let *keys* be the result of calling the [[Enumerate]] internal method of *obj* with no arguments.
7. Else,
    a. Assert *iterationKind* is iterate.
    b. Let *keys* be the result of performing Invoke with arguments *obj*, %iterator%, and an empty List.
8. If *keys* is an abrupt completion, then
    a. If LoopContinues(*experValue*,*labelSet*) is **false**, then return *experValue*.
    b. Assert: *keys*.[[type]] is continue
    c. Return Completion {[[type]]: break, [[value]]: empty, [[target]]: empty}.
9. Return *keys*.

Runtime Semantics: For In/Of Body Evaluation Abstract Operation

The abstract operation For In/Of Body Evaluation is called with arguments *lhs*, *stmt*, *keys*, *lhsKind*, and *labelSet.* The value of *lhsKind* is either **assignment**, **varBinding** or **lexicalBinding**.

1. Let *oldEnv* be the running execution context's LexicalEnvironment.
2. Let *noArgs* be an empty List.
3. Let *V* = **undefined** .
4. Repeat
    a. Let *next* be the result of Invoke(*keys*, "`next`").
    b. If IteratorComplete(*next*) is **true**, then return NormalCompletion(*V*).
    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. Assert *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 running execution context's LexicalEnvironment to *iterationEnv*.
        vi. If *status*.[[type]] is normal, then
            1. Let *status* be the result of evaluating *stmt*.
            2. If *status*.[[type]] is normal and *status*.[[value]] is not empty, then
                a Let *V* = *status*.[[value]].
        vii. Set the running 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*.

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

- It is a Syntax Error if a **`return`** statement 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. ReturnIfAbrupt(*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 running 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. ReturnIfAbrupt(*obj*).
4. Let *oldEnv* be the running 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 running execution context's LexicalEnvironment to *newEnv*.
8. Let *C* be the result of evaluating *Statement*.
9. Set the running execution context's Lexical Environment to *oldEnv*.
10. Return *C*.

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 :
`{` $CaseClauses_{opt}$ **`}`**
`{` $CaseClauses_{opt}$ *DefaultClause* $CaseClauses_{opt}$ **`}`**

CaseClauses :
CaseClause
CaseClauses CaseClause

CaseClause :
`case` *Expression* **`:`** $StatementList_{opt}$

DefaultClause :
`default :` $StatementList_{opt}$

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*.

Static Semantics: LexicalDeclarations

CaseBlock : **`{ }`**

1. Return a new empty List.

CaseBlock : **{** *CaseClauses*$_{opt}$ *DefaultClause CaseClauses*$_{opt}$ **}**

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 LexicalDeclarations of the *DefaultClause*.
4. If the second *CaseClauses* is not present, return *declarations*.
5. Else return the result of appending to *declarations* the elements of the LexicalDeclarations of the second *CaseClauses*.

CaseClauses : *CaseClauses CaseClause*

1. Let *declarations* be LexicalDeclarations of *CaseClauses*.
2. Append to *declarations* the elements of the LexicalDeclarations of *CaseClause*.
3. Return *declarations*.

CaseClause : **case** *Expression* : *StatementList*$_{opt}$

1. If the *StatementList* is present, return the LexicalDeclarations of *StatementList*.
2. Else return a new empty List.

DefaultClause : **default** : *StatementList*$_{opt}$

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*$_{opt}$ *DefaultClause CaseClauses*$_{opt}$ **}**

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*.

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*$_{opt}$

1. If the *StatementList* is present, return the LexicallyDeclaredNames of *StatementList*.
2. Else return a new empty List.

DefaultClause : **default** : *StatementList*$_{opt}$

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*

1. Return the VarDeclaredNames of *CaseBlock*.

CaseBlock : **{ }**

1. Return a new empty List.

CaseBlock : **{** *CaseClauses*$_{opt}$ *DefaultClause CaseClauses*$_{opt}$ **}**

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*.

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*$_{opt}$

1. If the *StatementList* is present, return the VarDeclaredNames of *StatementList*.
2. Else return a new empty List.

DefaultClause : **default :** *StatementList*$_{opt}$

1. If the *StatementList* is present, return the VarDeclaredNames of *StatementList*.
2. Else return a new empty List.

Runtime Semantics

Runtime Semantics: Case Block Evaluation

With argument *input*.

CaseBlock : **{** *CaseClauses*$_{opt}$ **}**

1. Let *V* = **undefined**.
2. Let *A* be the list of *CaseClause* items in source text order.
3. Let *searching* be **true**.
4. Repeat, while *searching* is **true**
    a. Let *C* be the next *CaseClause* in *A*. If there is no such *CaseClause*, return NormalCompletion(*V*).
    b. Let *clauseSelector* be the result of evaluating *C*.
    c. ReturnIfAbrupt(*clauseSelector*).
    d. If *input* is equal to *clauseSelector* as defined by the Strict Equality Comparison Algorithm (11.9.1), then
        i. Set *searching* to **false**.
        ii. If *C* has a *StatementList*, then
            1. Evaluate *C*'s *StatementList* and let *R* be the result.
            2. ReturnIfAbrupt(*R*).
            3. Let *V* = *R*.[[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
        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*$_{opt}$ *DefaultClause* *CaseClauses*$_{opt}$ }

1. Let *V* = **undefined**.
2. Let *A* be the list of *CaseClause* items in the first *CaseClauses*, in source text order.
3. Let *found* be **false**.
4. Repeat letting *C* be in order each *CaseClause* in *A*
   a. If *found* is **false**, then
      i. Let *clauseSelector* be the result of Case Selector Evaluation of *C*.
      ii. If *clauseSelector* is an abrupt completion, then
         1. If *clauseSelector*.[[value]] is empty, then return Completion {[[type]]: *clauseSelector*.[[type]], [[value]]: **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. Evaluate *CaseClause C* 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]]}.
5. Let *foundInB* be **false**.
6. If *found* is **false**, then
   a. Let *B* be a new list of the *CaseClause* items in the second *CaseClauses*, in source text order.
   b. Repeat, letting *C* be in order each *CaseClause* in *B*
      i. If *foundInB* is **false**, then
         1. Let *clauseSelector* be the result of Case Selector Evaluation of *C*.
         2. If *clauseSelector* is an abrupt completion, then
            a If *clauseSelector*.[[value]] is empty, then return Completion {[[type]]: *clauseSelector*.[[type]], [[value]]: **undefined**, [[target]]: *clauseSelector*.[[target]]}.
            b Else, return *clauseSelector*.
         3. If *input* is equal to *clauseSelector* as defined by the Strict Equality Comparison Algorithm (11.9.1), then set *foundInB* to **true**.
      ii. If *foundInB* is **true**, then
         1. Evaluate *CaseClause C* 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]]}.
7. If *foundInB* is **true**, then return NormalCompletion(*V*).
8. Evaluate *DefaultClause* and let *R* be the result.
9. If *R*.[[value]] is not empty, then let *V* = *R*.[[value]].
10. If *R* is an abrupt completion, then return Completion {[[type]]: *R*.[[type]], [[value]]: *V*, [[target]]: *R*.[[target]]}.
11. Let *B* be a new list of the *CaseClause* items in the second *CaseClauses*, in source text order.
12. Repeat, letting *C* be in order each *CaseClause* in *B* (NOTE this is another complete iteration of the second *CaseClauses*)
    b. Evaluate *CaseClause C* and let *R* be the result.
    c. If *R*.[[value]] is not empty, then let *V* = *R*.[[value]].
    d. If *R* is an abrupt completion, then return Completion {[[type]]: *R*.[[type]], [[value]]: *V*, [[target]]: *R*.[[target]]}.
13. Return NormalCompletion(*V*).

Runtime Semantics: Case Selector Evaluation

CaseClause : **case** *Expression* : *StatementList*$_{opt}$

1. Let *exprRef* be the result of evaluating *Expression*.
2. Return GetValue(*exprRef*).

NOTE Case Selector Evaluation 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.

Runtime Semantics: Evaluation

SwitchStatement **: switch (** *Expression* **)** *CaseBlock*

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *switchValue* be GetValue(*exprRef*).
3. ReturnIfAbrupt(*switchValue*).
4. Let *oldEnv* be the running 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 Block Evaluation of *CaseBlock* with argument *switchValue*.
8. Set the running 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* **:** [empty]

1. Return NormalCompletion(empty).

CaseClause **: case** *Expression* **:** *StatementList*

1. Return the result of evaluating *StatementList*.

DefaultClause **: default:** [empty]

1. Return NormalCompletion(empty).

DefaultClause **: default:** *StatementList*

1. Return the result of evaluating *StatementList*.

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 the enclosed *LabelledStatement*. This does not apply to a *LabelledStatement* appearing within the body of a *FunctionDeclaration* and a *LabelledStatement* that encloses, directly or indirectly the *FunctionDeclaration* .

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*.
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,
    a. 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* **`;`**

Runtime Semantics: Evaluation

The production *ThrowStatement* **:** **`throw`** [no *LineTerminator* here] *Expression* **`;`** is evaluated as follows:

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be GetValue(*exprRef*).
3. ReturnIfAbrupt(*exprValue*).
4. Return Completion {[[type]]: throw, [[value]]: GetValue(*exprRef*), [[target]]: 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.

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*.

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*.

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*.

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.

CatchParameter : *BindingPattern*

4. Let *exceptionObj* be ToObject(*value*).
5. ReturnIfAbrupt(*exceptionObj*).
6. Return the result of performing Binding Initialisation 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 running 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 Binding Initialisation for *CatchParameter* passing *thrownValue* and *catchEnv* as arguments.
5. ReturnIfAbrupt(*status*).
6. Set the running execution context's LexicalEnvironment to *catchEnv*.
7. Let *B* be the result of evaluating *Block*.
8. Set the running 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*.

12.15 The `debugger` statement

Syntax

DebuggerStatement :
`debugger ;`

Runtime Semantics: Evaluation

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.

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* **}**

FunctionExpression :
function *BindingIdentifier*$_{opt}$ **(** *FormalParameterList* **)** **{** *FunctionBody* **}**

FormalParameterList :
[empty]
FunctionRestParameter
FormalsList
FormalsList , *FunctionRestParameter*

FormalsList :
FormalParameter
FormalsList , *FormalParameter*

FunctionRestParameter :
... *BindingIdentifier*

FormalParameter :
BindingElement

FunctionBody :
StatementList$_{opt}$

Static Semantics

Static Semantics: Early Errors

FunctionDeclaration : **function** *BindingIdentifier* **(** *FormalParameterList* **)** **{** *FunctionBody* **}**
and
FunctionExpression : **function** *BindingIdentifier*$_{opt}$ **(** *FormalParameterList* **)** **{** *FunctionBody* **}**

- It is a Syntax Error if IsSimpleParameterList of *FormalParameterList* is **true** and any element of the BoundNames of *FormalParameterList* also occurs in the VarDeclaredNames of *FunctionBody*.
- It is a Syntax Error if IsSimpleParameterList of *FormalParameterList* is **false** and BoundNames of *FormalParameterList* contains any duplicate elements.
- It is a Syntax Error if IsSimpleParameterList of *FormalParameterList* is **false** and BoundNames of *FormalParameterList* contains either **"eval"** or **"arguments"**.

- It is a Syntax Error if the source code matching this production is strict code and BoundNames of *FormalParameterList* contains any duplicate elements.
- 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*.

NOTE The LexicallyDeclaredNames of a *FunctionBody* does not include identifiers bound using var or function declarations. Simple parameter lists bind identifiers as VarDeclaredNames. Parameter lists that contain destructuring patterns, default value initialisers, or a rest parameter bind identifiers as LexicallyDeclaredNames. Multiple occurrences of the same *Identifier* in a *FormalParamterList* is only allowed for non-strict functions with simple parameter lists.

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*.

FormalParameter **:** *BindingElement*

- It is a Syntax Error if *BindingElement* Contains *YieldExpression.*

Static Semantics: BoundNames

FunctionDeclaration **:** **`function`** *BindingIdentifier* **`(`** *FormalParameterList* **`)`** **`{`** *FunctionBody* **`}`**

1. Return the BoundNames of *BindingIdentifier*.

FormalParameterList **:** [empty]

1. Return an empty List.

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*.

Static Semantics: Contains

With parameter *symbol*.

FunctionDeclaration **:** **`function`** *BindingIdentifier* **`(`** *FormalParameterList* **`)`** **`{`** *FunctionBody* **`}`**

1. Return **false**.

FunctionExpression **:** **`function`** $BindingIdentifier_{opt}$ **`(`** *FormalParameterList* **`)`** **`{`** *FunctionBody* **`}`**

3. 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* **,** *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. A *FormalParameter* without an initializer is allowed after the first parameter with an initializer but such parameters are considered to be optional with **undefined** as their default value.

FormalsList **:** *FormalParameter*

1. If HasInitialiser of *FormalParameter* is **false** return 0
2. Return 1.

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 **:** *FormalsList* **,** *FormalParameter*

1. If HasInitialiser of *FormalsList* is **true**, then return **true**.
2. Return HasInitialiser of *FormalParameter*.

Static Semantics: IsConstantDeclaration

FunctionDeclaration **:** **`function`** *BindingIdentifier* **`(`** *FormalParameterList* **`)`** **`{`** *FunctionBody* **`}`**

1. Return **false**.

Static Semantics: IsSimpleParameterList

FormalParameterList **:** [empty]

1. Return **true**.

FormalParameterList **:** *FunctionRestParameter*

1. Return **false**.

FormalParameterList **:** *FormalsList* **,** *FunctionRestParameter*

1. Return **false**.

FormalsList **:** *FormalsList* **,** *FormalParameter*

1. If IsSimpleParameterList of *FormalsList* is **false**, return **false**.
2. Return IsSimpleParameterList of *FormalParameter*.

FormalParameter **:** *BindingElement*

1. If HasInitialiser of *BindingElement* is **true**, return **false**.

2. If *FormalParameter* Contains *BindingPattern* is **true**, return **false**.
3. Return **true**.

Static Semantics: IsStrict

FunctionBody **:** *StatementList*$_{opt}$

1. If this *FunctionBody* is contained in strict code or if *StatementList* is strict code, then return **true**. Otherwise, return **false**.

Static Semantics: LexicallyDeclaredNames

FunctionDeclaration **:** **`function`** *BindingIdentifier* **`(`** *FormalParameterList* **`)`** **`{`** *FunctionBody* **`}`**

1. Return the BoundNames of *BindingIdentifier*.

FunctionBody **:** [empty]

1. Return an empty List.

FunctionBody **:** *StatementList*

1. Return TopLevelLexicallyDeclaredNames 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 TopLevelVarDeclaredNames of *StatementList*.

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.

FormalParameterList **:** [empty]

1. Return NormalCompletion(empty).

FormalParameterList **:** *FunctionRestParameter*

1. Return the result of performing Indexed Binding Initialisation for *FunctionRestParameter* using *value*, 0, and *environment* as the arguments.

FormalParameterList : *FormalsList*

1. Return the result of performing Indexed Binding Initialisation for *FormalsList* using *value*, 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*, *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*, *nextIndex*, and *environment* 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. Assert: *array* is a well formed arguments object and hence it has a valid integer valued **`"length"`** property.
2. Let *status* be the result of Get(*array*, **`"length"`**).
3. Let *argumentsLength* be *status*.[[value]].
4. Let *A* be the result of the abstract operation ArrayCreate 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 Get(*array*, *P*).
   d. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*n*) and Property Descriptor {[[Value]]: *v*.[[value]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
   e. Let *n* = *n*+1.
   f. Let *nextIndex* = *nextIndex* +1.
7. Return the result of performing Binding Initialisation for *BindingIdentifier* using *A* and *environment* as arguments.

Runtime Semantics: InstantiateFunctionObject

With parameter *scope*.

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 *F* be the result of performing the FunctionCreate abstract operation with arguments Normal, *FormalParameterList*, *FunctionBody*, *scope*, and *strict*.
3. Perform the abstract operation MakeConstructor with argument *F*.
4. Return *F*.

Runtime Semantics: Evaluation

FunctionDeclaration **:** **`function`** *BindingIdentifier* **`(`** *FormalParameterList* **`)`** **`{`** *FunctionBody* **`}`**

1. Return NormalCompletion(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 running execution context.
3. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Normal, *FormalParameterList*, *FunctionBody*, *scope*, and *strict*.
4. Perform 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 running execution context's Lexical Environment as the argument
3. Let *envRec* be *funcEnv's* environment record.
4. Let *name* be StringValue of *BindingIdentifier*.
5. Call the CreateImmutableBinding concrete method of *envRec* passing *name* as the argument.
6. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Normal, *FormalParameterList*, *FunctionBody*, *funcEnv*, and *strict*.
7. Perform the abstract operation MakeConstructor with argument *closure*.
8. Call the InitializeBinding concrete method of *envRec* passing *name* and *closure* as the arguments.
9. Return NormalCompletion(*closure*).

NOTE 1 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 2 A **`prototype`** property is automatically created for every function defined using a *FunctionDeclaration* or *FunctionExpression*, to allow for the possibility that the function will be used as a constructor.

FunctionBody **:** $StatementList_{opt}$

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
CoverParenthesizedExpressionAndArrowParameterList

ConciseBody **:**
[lookahead ∉ { **{** }] *AssignmentExpression*
{ *FunctionBody* **}**

Supplemental Syntax

When processing the production *ArrowParameters* **:** *CoverParenthesizedExpressionAndArrowParameterList* the following grammar is used to refine the interpretation of *CoverParenthesizedExpressionAndArrowParameterList*.

ArrowFormalParameterList **:**
(*FormalParameterList* **)**

Static Semantics

Static Semantics: Early Errors

ArrowFunction **:** *ArrowParameters* **=>** *ConciseBody*

- It is a Syntax Error if IsSimpleParameterList of *ArrowParameters* is **true** and any element of the BoundNames of *ArrowParameters* also occurs in the VarDeclaredNames of *ConciseBody*.
- It is a Syntax Error if IsSimpleParameterList of *ArrowParameters* is **false** and BoundNames of *ArrowParameters* contains any duplicate elements.
- It is a Syntax Error if IsSimpleParameterList of *ArrowParameters* is **false** and BoundNames of *ArrowParameters* contains either **"eval"** or **"arguments"**.
- It is a Syntax Error if the source code matching this production is strict code and BoundNames of *ArrowParameters* contains any duplicate elements.
- 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*.

ArrowParameters **:** *BindingIdentifier*

- It is a Syntax Error if the StringValue of the sole element of the BoundNames of *BindingIdentifier* is **`eval`** or **`arguments`**.

ArrowParameters **:** *CoverParenthesizedExpressionAndArrowParameterList*

- It is a Syntax Error if the lexical token sequence matched by *CoverParenthesizedExpressionAndArrowParameterList* cannot be parsed with no tokens left over using *ArrowFormalParameterList* as the goal symbol.

Static Semantics: BoundNames

ArrowParameters **:** *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *formals* be CoveredFormalsList of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return the BoundNames of *formals*.

Static Semantics: Contains

With parameter *symbol*.

ArrowFunction **:** *ArrowParameters* **=>** *ConciseBody*

1. If *ArrowParameters* Contains *symbol* is **true**, return **true**;
2. Return *ConciseBody* Contains *symbol* .

ArrowParameters **:** *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *formals* be CoveredFormalsList of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return *formals* Contains *symbol*.

NOTE Contains is used to detect **`yield`** and **`super`** usage within an *ArrowFunction*.

Static Semantics: CoveredFormalsList

*CoverParenthesizedExpressionAndArrowParameterList***:**
(*Expression* **)**
( )
(... *Identifier* **)**
(*Expression* **, ...** *Identifier***)**

1. Return the result of parsing the lexical token stream matched by *CoverParenthesizedExpressionAndArrowParameterList* using *ArrowFormalParameterList* as the goal symbol.

Static Semantics: ExpectedArgumentCount

ArrowParameters **:** *BindingIdentifier*

1. Return 1.

ArrowParameters **:** *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *formals* be CoveredFormalsList of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return the ExpectedArgumentCount of *formals*.

Static Semantics: IsSimpleParameterList

ArrowParameters **:** *BindingIdentifier*

1. Return **true**.

ArrowParameters **:** *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *formals* be CoveredFormalsList of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return the IsSimpleParameterList of *formals*.

Static Semantics: LexicallyDeclaredNames

ConciseBody **:** [lookahead ∉ { **{** }] *AssignmentExpression*

1. Return an empty List.

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.

ArrowParameters : *BindingIdentifier*

1. Return the result of performing Binding Initialisation for *BindingIdentifier* using *value* and *environment* as the arguments.

ArrowParameters : *CoverParenthesizedExpressionAndArrowParameterList*

1. Let *formals* be CoveredFormalsList of *CoverParenthesizedExpressionAndArrowParameterList*.
2. Return the result of performing Binding initialisation of *formals* with arguments *value* and *environment*.

Runtime Semantics: Evaluation

ArrowFunction : *ArrowParameters* **=>** *ConciseBody*

1. Let *strict* be **true**.
2. Let *scope* be the LexicalEnvironment of the running execution context.
3. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Arrow, *ArrowParameters*, *ConciseBody*, *scope*, and *strict*.
4. Return *closure*.

ConciseBody : [lookahead ∉ { **{** }] *AssignmentExpression*

1. The code of this *ConciseBody* 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 *ConciseBody* 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. ReturnIfAbrupt(*exprValue*).
5. Return Completion {[[type]]: return, [[value]]: *exprValue*, [[target]]: empty}.

13.3 Method Definitions

Syntax

MethodDefinition :
PropertyName **(** *FormalParameterList* **)** **{** *FunctionBody* **}**
***** *PropertyName* **(** *FormalParameterList* **)** **{** *FunctionBody* **}**
get *PropertyName* **(** **)** **{** *FunctionBody* **}**
set *PropertyName* **(** *PropertySetParameterList* **)** **{** *FunctionBody* **}**

PropertySetParameterList :
BindingIdentifier
BindingPattern

NOTE The single element of a *PropertySetParameterList* may not have a default value *Initialiser* because set accessor are always called with an implicitly provided argument.

Static Semantics

Static Semantics: Early Errors

MethodDefinition : *PropertyName* **(** *FormalParameterList* **)** **{** *FunctionBody* **}**

and

MethodDefinition : ***** *PropertyName* **(** *FormalParameterList* **)** **{** *FunctionBody* **}**

- It is a Syntax Error if IsSimpleParameterList of *FormalParameterList* is **true** and any element of the BoundNames of *FormalParameterList* also occurs in the VarDeclaredNames of *FunctionBody*.

- It is a Syntax Error if IsSimpleParameterList of *FormalParameterList* is **false** and BoundNames of *FormalParameterList* contains any duplicate elements.
- It is a Syntax Error if IsSimpleParameterList of *FormalParameterList* is **false** and BoundNames of *FormalParameterList* contains either **"eval"** or **"arguments"**.
- It is a Syntax Error if the source code matching this production is strict code and BoundNames of *FormalParameterList* contains any duplicate elements.
- It is a Syntax Error if any element of the BoundNames of *FormalParameterList* also occurs in the LexicallyDeclaredNames of *FunctionBody*.

MethodDefinition **:** *PropertyName* **(** *FormalParameterList* **)** **{** *FunctionBody* **}**

- It is a Syntax Error if *FunctionBody* Contains *YieldExpression*.

MethodDefinition **:** ***** *PropertyName* **(** *FormalParameterList* **)** **{** *FunctionBody* **}**

- It is a Syntax Error if *FunctionBody* Contains *YieldExpression* is **false**.

MethodDefinition **:** **get** *PropertyName* **( )** **{** *FunctionBody* **}**

- It is a Syntax Error if *FunctionBody* Contains *YieldExpression*.

MethodDefinition **:** **set** *PropertyName* **(** *PropertySetParameterList* **)** **{** *FunctionBody* **}**

- It is a Syntax Error if IsSimpleParameterList of *PropertySetParameterList* is **true** and any element of the BoundNames of *PropertySetParameterList* also occurs in the VarDeclaredNames of *FunctionBody*.
- It is a Syntax Error if IsSimpleParameterList of *PropertySetParameterList* is **false** and BoundNames of *PropertySetParameterList* contains any duplicate elements.
- It is a Syntax Error if IsSimpleParameterList of *PropertySetParameterList* is **false** and BoundNames of *PropertySetParameterList* contains either **"eval"** or **"arguments"**.
- It is a Syntax Error if BoundNames of *PropertySetParameterList* contains any duplicate elements.
- It is a Syntax Error if any element of the BoundNames of *PropertySetParameterList* also occurs in the LexicallyDeclaredNames of *FunctionBody*.
- It is a Syntax Error if *PropertySetParameterList* Contains *YieldExpression*.
- It is a Syntax Error if *FunctionBody* Contains *YieldExpression*.

Static Semantics: ExpectedArgumentCount

PropertySetParameterList **:** *BindingIdentifier*

1. Return 1.

PropertySetParameterList **:** *BindingPattern*

1. Return 1.

Static Semantics: IsSimpleParameterList

PropertySetParameterList **:** *BindingIdentifier*

1. Return **true**.

PropertySetParameterList **:** *BindingPattern*

1. Return **false**.

Static Semantics: PropName

MethodDefinition **:**
PropertyName **(** *FormalParameterList* **)** **{** *FunctionBody* **}**
***** *PropertyName* **(** *FormalParameterList* **)** **{** *FunctionBody* **}**
get *PropertyName* **(** **)** **{** *FunctionBody* **}**
set *PropertyName* **(** *PropertySetParameterList* **)** **{** *FunctionBody* **}**

1. Return PropName of *PropertyName*.

Static Semantics: ReferencesSuper

MethodDefinition **:**
PropertyName **(** *FormalParameterList* **)** **{** *FunctionBody* **}**
***** *PropertyName* **(** *FormalParameterList* **)** **{** *FunctionBody* **}**
get *PropertyName* **(** **)** **{** *FunctionBody* **}**
set *PropertyName* **(** *PropertySetParameterList* **)** **{** *FunctionBody* **}**

1. Return *FunctionBody* Contains **super**.

Static Semantics: SpecialMethod

MethodDefinition **:** *PropertyName* **(** *FormalParameterList* **)** **{** *FunctionBody* **}**

1. Return **false**.

MethodDefinition **:**
***** *PropertyName* **(** *FormalParameterList* **)** **{** *FunctionBody* **}**
get *PropertyName* **(** **)** **{** *FunctionBody* **}**
set *PropertyName* **(** *PropertySetParameterList* **)** **{** *FunctionBody* **}**

1. Return **true**.

Runtime Semantics

Runtime Semantics: Property Definition Evaluation

With parameter *object*.

MethodDefinition **:** *PropertyName* **(** *FormalParameterList* **)** **{** *FunctionBody* **}**

1. Let *propName* be PropName of *PropertyName*.
2. Let *strict* be IsStrict of *FunctionBody*.
3. Let *scope* be the running execution context's LexicalEnvironment.
4. Let *needsSuperBinding* be the result of *FunctionBody* Contains **super**.
5. If *needsSuperBinding* is **false**, then let *needsSuperBinding* be the result of *FormalParameterList* Contains **super**.
6. If *needsSuperBinding*, then
    a. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Method, *FormalParameterList*, *FunctionBody*, *scope*, and *strict* and with *object* as the *homeObject* optional argument and *propName* as the *methodName* optional argument.
7. Else
    a. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Method, *FormalParameterList*, *FunctionBody*, *scope*, and *strict*.
8. Let *desc* be the Property Descriptor{[[Value]]: *closure*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
9. Let *status* be the result of calling the [[DefineOwnProperty]] internal method of *object* with arguments *propName* and *desc*.
10. ReturnIfAbrupt(*status*).
11. NormalCompletion(closure).

MethodDefinition **:** ***** *PropertyName* **(** *FormalParameterList* **)** **{** *FunctionBody* **}**

1. Let *propName* be PropName of *PropertyName*.
2. Let *strict* be IsStrict of *FunctionBody*.
3. Let *scope* be the running execution context's LexicalEnvironment.
4. Let *needsSuperBinding* be the result of *FunctionBody* Contains **`super`**.
5. If *needsSuperBinding* is **false**, then let *needsSuperBinding* be the result of *FormalParameterList* Contains **`super`**.
6. If *needsSuperBinding*, then
    a. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Method, *FormalParameterList*, *FunctionBody*, *scope*, and *strict* and with *object* as the *homeObject* optional argument and *propName* as the *methodName* optional argument.
7. Else
    a. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Method, *FormalParameterList*, *FunctionBody*, *scope*, and *strict*.
8. Let *desc* be the Property Descriptor{[[Value]]: *closure*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
9. Let *status* be the result of calling the [[DefineOwnProperty]] internal method of *object* with arguments *propName* and *desc*.
10. ReturnIfAbrupt(*status*).
11. Return NormalCompletion(*closure*).

MethodDefinition **:** **`get`** *PropertyName* **`( ) {`** *FunctionBody* **`}`**

1. Let *propName* be PropName of *PropertyName*.
2. Let *strict* be IsStrict of *FunctionBody*.
3. Let *scope* be the running execution context's LexicalEnvironment.
4. Let *formalParameterList* be the production *FormalParameterList* **:** [empty]
5. Let *needsSuperBinding* be the result of *FunctionBody* Contains **`super`**.
6. If *needsSuperBinding*, then
    a. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Method, *formalParameterList*, *FunctionBody*, *scope*, and *strict* and with *object* as the *homeObject* optional argument and *propName* as the *methodName* optional argument.
7. Else
    a. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Method, *formalParameterList*, *FunctionBody*, *scope*, and *strict*.
8. Let *desc* be the Property Descriptor {[[Get]]: *closure*, [[Enumerable]]: **true**, [[Configurable]]: **true**}
9. Let *status* be the result of calling the [[DefineOwnProperty]] internal method of *object* with arguments *propName* and *desc*.
10. ReturnIfAbrupt(*status*).
11. Return NormalCompletion(*closure*).

MethodDefinition **:** **`set`** *PropertyName* **`(`** *PropertySetParameterList* **`) {`** *FunctionBody* **`}`**

1. Let *propName* be PropName of *PropertyName*.
2. Let *strict* be IsStrict of *FunctionBody*.
3. Let *scope* be the running execution context's LexicalEnvironment.
4. Let *needsSuperBinding* be the result of *FunctionBody* Contains **`super`**.
5. If *needsSuperBinding* is **false**, then let *needsSuperBinding* be the result of *PropertySetParameterList* Contains **`super`**.
6. If *needsSuperBinding*, then
    a. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Method, *PropertySetParameterList*, *FunctionBody*, *scope*, and *strict* and with *object* as the *homeObject* optional argument and *propName* as the *methodName* optional argument.
7. Else
    a. Let *closure* be the result of performing the FunctionCreate abstract operation with arguments Method, *PropertySetParameterList*, *FunctionBody*, *scope*, and *strict*.
8. Let *desc* be the Property Descriptor {[[Set]]: *closure*, [[Enumerable]]: **true**, [[Configurable]]: **true**}
9. Let *status* be the result of calling the [[DefineOwnProperty]] internal method of *object* with arguments *propName* and *desc*.
10. ReturnIfAbrupt(*status*).
11. Return NormalCompletion(*closure*).

13.4 Generator Definitions

Syntax

GeneratorDeclaration **:**
function * *BindingIdentifier* **(** *FormalParameterList* **) {** *FunctionBody* **}**

GeneratorExpression **:**
function * *BindingIdentifier*$_{opt}$ **(** *FormalParameterList* **) {** *FunctionBody* **}**

YieldExpression **:**
yield *YieldDelegator*$_{opt}$ [Lexical goal *InputElementRegExp*] *AssignmentExpression*

YieldDelegator **:**
*****

Static Semantics

Static Semantics: Early Errors

GeneratorDeclaration **: function *** *BindingIdentifier* **(** *FormalParameterList* **) {** *FunctionBody* **}**
and
GeneratorExpression **: function *** *BindingIdentifier*$_{opt}$ **(** *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* is **false**.

YieldExpression **: yield** *YieldDelegator*$_{opt}$ *AssignmentExpression*

- It is a Syntax Error if *AssignmentExpression* Contains *YieldExpression*.

Static Semantics: BoundNames

GeneratorDeclaration **: function *** *BindingIdentifier* **(** *FormalParameterList* **) {** *FunctionBody* **}**

1. Return the BoundNames of *BindingIdentifier*.

Static Semantics: Contains

With parameter *symbol*.

GeneratorDeclaration **: function *** *BindingIdentifier* **(** *FormalParameterList* **) {** *FunctionBody* **}**

1. Return **false**.

GeneratorExpression **: function *** *BindingIdentifier*$_{opt}$ **(** *FormalParameterList* **) {** *FunctionBody* **}**

1. Return **false**.

NOTE Static semantic rules that depend upon substructure generally do not look into function definitions.

Static Semantics: IsConstantDeclaration

GeneratorDeclaration **: function *** *BindingIdentifier* **(** *FormalParameterList* **) {** *FunctionBody* **}**

1. Return **false**.

Static Semantics: LexicallyDeclaredNames

GeneratorDeclaration : **function** * *BindingIdentifier* **(** *FormalParameterList* **)** **{** *FunctionBody* **}**

1. Return the BoundNames of *BindingIdentifier*.

Static Semantics: VarDeclaredNames

GeneratorDeclaration : **function** * *BindingIdentifier* **(** *FormalParameterList* **)** **{** *FunctionBody* **}**

1. Return an empty List.

Runtime Semantics

13.5 Class Definitions

Syntax

ClassDeclaration:
class *BindingIdentifier* *ClassTail*

ClassExpression :
class *BindingIdentifier*$_{opt}$ *ClassTail*

ClassTail :
ClassHeritage$_{opt}$ **{** *ClassBody*$_{opt}$ **}**

ClassHeritage:
extends *AssignmentExpression*

ClassBody :
ClassElementList

ClassElementList :
ClassElement
ClassElementList *ClassElement*

ClassElement :
MethodDefinition
;

Static Semantics

Static Semantics: Early Errors

ClassDeclaration : **class** *BindingIdentifier* *ClassTail*
and
ClassExpression : **class** *BindingIdentifier* *ClassTail*

- It is a Syntax Error if BoundNames of *BindingIdentifier* contains either **"eval"** or **"arguments"**

ClassBody : *ClassElementList*

- It is a Syntax Error if PropertyNameList of *ClassElementList* contains any duplicate entries, unless the following condition is true for each duplicate entry: The duplicated entry occurs exactly twice in the list and one occurrence was obtained from a **get** accessor *MethodDefinition* and the other occurrence was obtained from a **set** accessor *MethodDefinition*.

ClassElement : *MethodDefinition*

- It is a Syntax Error if PropName of *MethodDefinition* is **"constructor"** and SpecialMethod of *MethodDefinition* is **true**.

Static Semantics: BoundNames

ClassDeclaration: **class** *BindingIdentifier ClassTail*

1. Return the BoundNames of *BindingIdentifier*.

Static Semantics: ConstructorMethod

ClassBody : *ClassElementList*

1. Let *list* be MethodDefinitions of *ClassElementList*.
2. For each *MethodDefinition m* in *list*, do
   a. If PropName of *m* is **"constructor"**, return *m*.
3. Return empty.

NOTE Early Error rules ensure that there is only one method definition named **"constructor"** and that it isn't an accessor property or generator definition.

Static Semantics: Contains

With parameter *symbol*.

ClassTail : *ClassHeritage*$_{opt}$ { *ClassBody* }

1. If *symbol* is *ClassBody*, return **true**.
2. If *ClassHeritage* is not present, return **false**.
3. If *symbol* is *ClassHeritage*, return **true**.
4. Return the result of Contains for *ClassHeritage* with argument *symbol*.

NOTE Static semantic rules that depend upon substructure generally do not look into class bodies.

Static Semantics: IsConstantDeclaration

ClassDeclaration: **class** *BindingIdentifier ClassTail*

1. Return **false**.

Static Semantics: LexicallyDeclaredNames

ClassDeclaration: **class** *BindingIdentifier ClassTail*

1. Return the BoundNames of *BindingIdentifier*.

Static Semantics: MethodDefinitions

ClassElementList : *ClassElement*

1. If PropName of *ClassElement* is empty, return a new empty List.
2. Return a List containing *ClassElement*.

ClassElementList : *ClassElementList ClassElement*

5. Let *list* be MethodDefinitions of *ClassElementList*.
6. If PropName of *ClassElement* is empty, return *list*.
7. Append *ClassElement* to the end of *list*.
8. Return *list*.

Static Semantics: PropName

ClassElement : ;

1. Return empty.

Static Semantics: PropertyNameList

ClassElementList : *ClassElement*

1. If PropName of *ClassElement* is empty, return a new empty List.
2. Return a List containing PropName of *ClassElement*.

ClassElementList : *ClassElementList ClassElement*

1. Let *list* be PropertyNameList of *ClassElementList*.
2. If PropName of *ClassElement* is empty, return *list*.
3. Append PropName of *ClassElement* to the end of *list*.
4. Return *list*.

Static Semantics: VarDeclaredNames

ClassDeclaration: **`class`** *BindingIdentifier ClassTail*

1. Return an empty List.

Runtime Semantics

Runtime Semantics: ClassDefinitionEvaluation

With parameter *className*.

ClassTail : $ClassHeritage_{opt}$ **{** *ClassBody* **}**

1. If $ClassHeritage_{opt}$ is not present, then
    a. let *protoParent* be the intrinsic object %ObjectPrototype%.
    b. Let *constructorParent* be the intrinsic object %FunctionPrototype%.
2. Else
    a. Let *superclass* be the result of evaluating *ClassHeritage*.
    b. ReturnIfAbrupt(*superclass*).
    c. If *superclass* is **null**, then
        i. Let *protoParent* be **null**.
        ii. Let *constructorParent* be the intrinsic object %FunctionPrototype%.
    d. Else if Type(*superclass*) is not Object, throw a TypeError exception.
    e. Else if *superclass* does not have a [[Construct]] internal method, then
        i. Let *protoParent* be *superclass*.
        ii. Let *constructorParent* be the intrinsic object %FunctionPrototype%.
    f. Else
        i. Let *protoParent* be the result of Get(*superclass*, **`"prototype"`**).
        ii. ReturnIfAbrupt(*protoParent*).
        iii. If Type(*protoParent*) is neither Object or Null, throw a TypeError exception.
        iv. Let *constructorParent* be *superclass*.
3. Let *proto* be the result of the abstract operation ObjectCreate with argument *protoParent*.
4. Let *lex* be the LexicalEnvironment of the running execution context.
5. If *className* is not **undefined**, then
    a. Let *scope* be the result of calling NewDeclarativeEnvironment passing *lex* as the argument
    b. Let *envRec* be *scope's* environment record.
    c. Call the CreateImmutableBinding concrete method of *envRec* passing *className* as the argument.
    d. Set the running execution context's LexicalEnvironment to *scope*.

6. Let *constructor* be ConstructorMethod of *ClassBody*.
7. If *constructor* is empty, then
    a. Let *constructor* be the result of parsing the String **`"constructor(...args){super.constructor(...args);}"`** using the syntactic grammar with the goal symbol *MethodDefinition*.
8. If the *ClassTail* is contained in strict code or if *constructor* is strict code, then let *strict* be **true**. Otherwise let *strict* be **false**.
9. Let *F* be the result of performing Property Definition Evaluation for *constructor* with argument *proto*.
10. Perform the abstract operation MakeConstructor with argument *F* and **false** as the optional *writablePrototype* argument and *proto* as the optional *prototype* argument.
11. Let *desc* be the Property Descriptor{[[Enumerable]]: **false**, [[Writable]]: **true**, [[Configurable]]: **true**}.
12. Call the [[DefineOwnProperty]] internal method of *proto* with arguments **`"constructor"`** and *desc*
13. Let *methods* be MethodDefinitions of *ClassBody*.
14. For each *MethodDefinition m* in order from *methods*
    a. Perform Property Definition Evaluation for *m* with argument *proto*.
15. Set the running execution context's LexicalEnvironment to *lex*.
16. Return *F*.

Runtime Semantics: Evaluation

ClassDeclaration: **`class`** *BindingIdentifier ClassTail*

1. Let *value* be the result of ClassDefinitionEvaluation of *ClassTail* with argument **undefined**.
2. ReturnIfAbrupt(*value*).
3. Let *env* be the running execution context's LexicalEnvironment.
4. Let *status* be the result of performing Binding Initialisation for *BindingIdentifier* passing *value* and *env* as the arguments.
5. ReturnIfAbrupt(*status*).
6. Return NormalCompletion(empty).

NOTE The argument to ClassDefinitionEvaluation controls whether or not the class that is defined with a BindingIdentifier has a local binding to the identifier. Only a *ClassExpression* gets a local name binding of its name. A *ClassDeclaration* never has such a binding. This maintains the parallel with *FunctionExpression* and *FunctionDeclaration*.

ClassExpression: **`class`** $BindingIdentifier_{opt}$ *ClassTail*

1. If $BindingIdentifier_{opt}$ is not present, then let *className* be **undefined**.
2. Else, let *className* be StringValue of *BindingIdentifier*.
3. Let *value* be the result of ClassDefinitionEvaluation of *ClassTail* with argument *className*.
4. ReturnIfAbrupt(*value*).
5. Return NormalCompletion(*value*).

13.6 Creating Function Objects and Constructors

Runtime Semantics: FunctionCreate Abstract Operation

The abstract operation FunctionCreate requires the arguments: *kind* which is one of (Normal, Method, Arrow), an parameter list specified by *ParameterList*, a body specified by *Body*, 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 ECMAScript object and let *F* be that object.
2. Set *F*'s essential internal methods except for [[GetP]] and [[GetOwnProperty]] to the default ordinary object definitions specified in 8.3.
3. Set *F*'s essential internal methods for [[Call]], [[GetP]] and [[GetOwnProperty]] to the default ordinary object definitions specified in 8.3.19.
4. Add the [[BuiltinBrand]] internal data property with value BuiltinFunction to *F*.
5. If the *functionPrototype* argument was not provided,then
    a. Let *functionPrototype* be the intrinsic object %FunctionPrototype%.

6. Set the [[Prototype]] internal data property of *F* to *functionPrototype*.
7. Set the [[Scope]] internal data property of *F* to the value of *Scope*.
8. Set the [[FormalParameters]] internal property of *F* to *ParameterList*. .
9. Set the [[Code]] internal data property of *F* to *Body*.
10. Set the [[Extensible]] internal data property of *F* to **true**.
11. Set the [[Realm]] internaldata property of *F* to the running execution context's Realm.
12. If the *homeObject* argument was provided, set the [[HomeObject]] internal data property of *F* to *homeObject*.
13. If the *methodName* argument was provided, set the [[MethodName]] internal data property of *F* to *methodName*.
14. Set the [[Strict]] internal data property of *F* to *Strict*.
15. If *kind* is Arrow, then set the [[ThisMode]] internal data property of *F* to lexical.
16. Else if *Strict* is **true**, then set the [[ThisMode]] internal data property of *F* to strict.
17. Else set the [[ThisMode]] internal data property of *F* to global.
18. Let *len* be the ExpectedArgumentCount of *ParameterList*.
19. Call the [[DefineOwnProperty]] internal method of *F* with arguments **"length"** and Property Descriptor {[[Value]]: *len*, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false**}
20. If *kind* is Normal and *Strict* is **true**, then
    a. Perform the AddRestrictedFunctionProperties abstract operationr with argument *F*.
21. Return *F*.

Runtime Semantics: MakeConstructor Abstract Operation

The abstract operation MakeConstructor requires a Function argument *F* and optionally, a Boolean *writablePrototype* and an object *prototype*. If *prototype* is provided it is assume to already contain a **"constructor"** whose value is *F*. It converts *F* into a constructor by performs the following steps:

1. Let *installNeeded* be **false**.
2. If the *prototype* argument was not provided,then
    a. Let *installNeeded* be **true**.
    b. Let *prototype* be the result of the abstract operation ObjectCreate.
3. If the *writablePrototype* argument was not provided,then
    a. Let *writablePrototype* be **true**.
4. Set *F*'s essential internal method [[Construct[[to the definitions specified in 8.3.19.2.
5. If *installNeeded*, then
    a. Call the [[DefineOwnProperty]] internal method of *prototype* with arguments **"constructor"** and Property Descriptor {[[Value]]: *F*, [[Writable]]: *writablePrototype*, [[Enumerable]]: **false**, [[Configurable]]: *writablePrototype* }
7. Call the [[DefineOwnProperty]] internal method of *F* with arguments **"prototype"** and Property Descriptor {[[Value]]: *prototype* , [[Writable]]: *writablePrototype* , [[Enumerable]]: **false**, [[Configurable]]: **false**}.
8. Return.

13.6.3 The [[ThrowTypeError]] Function Object

The [[ThrowTypeError]] object is a unique function object that is defined once as follows:

1. Create a new ECMAScript object and let *F* be that object.
2. Set all the internal methods of *F* as described in 8.12.
3. Add the [[BuiltinBrand]] internal data property with value BuiltinFunction to *F*.
4. Set the [[Prototype]] internal data property of *F* to the standard built-in Function prototype object as specified in 15.3.3.1.
5. Set the [[Call]] internal method of *F* as described in 13.6.1.
6. Set the [[Scope]] internal data property of *F* to the Global Environment.
7. Set the [[FormalParameters]] internal data property of *F* to the *FormalParameterList* **:** [empty] production.
8. Set the [[Code]] internal data 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"** and Property Descriptor {[[Value]]: 0, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false**}.
10. Call the [[PreventExtensons]] internal method of *F*.
11. Let [[ThrowTypeError]] be *F*.

Runtime Semantics: AddRestrictedFunctionProperties Abstract Operation

The abstract operation is called with a function object *F* as its argument. It performs the following steps:

1. Let *thrower* be the [[ThrowTypeError]] function Object defined above.
2. Call the [[DefineOwnProperty]] internal method of *F* with arguments **"caller"** and PropertyDescriptor {[[Get]]: *thrower*, [[Set]]: *thrower*, [[Enumerable]]: **false**, [[Configurable]]: **false**}.
3. Call the [[DefineOwnProperty]] internal method of *F* with arguments **"arguments"** and PropertyDescriptor {[[Get]]: *thrower*, [[Set]]: *thrower*, [[Enumerable]]: **false**, [[Configurable]]: **false**}.

13.7 Tail Position Calls

The wiki proposal has a preliminary attempt at defining tail position. See http://wiki.ecmascript.org/doku.php?id=harmony:proper_tail_calls.

This material still needs to be reviewed and updated for incorporation here.

14 Scripts and Modules

14.1 Script

Syntax

Script **:**
ScriptBody$_{opt}$

ScriptBody **:**
OuterStatementList

OuterStatementList **:**
OuterItem
OuterStatementList OuterItem

OuterItem **:**
ModuleDeclaration
ImportDeclaration
StatementListItem

Static Semantics

Static Semantics: Early Errors

ScriptBody **:** *OuterStatementList*

- It is a Syntax Error if the LexicallyDeclaredNames of *OuterStatementList* contains any duplicate entries.
- It is a Syntax Error if any element of the LexicallyDeclaredNames of *OuterStatementList* also occurs in the VarDeclaredNames of *OuterStatementList*.
- It is a Syntax Error if *OuterStatementList* Contains *ReturnStatement*.
- It is a Syntax Error if *OuterStatementList* Contains **super**.
- It is a Syntax Error if *OuterStatementList* Contains *YieldExpression*.

NOTE Additional error conditions relating to conflicting or duplicate declarations are checked during module linking prior to evaluation of a *Script*. If any such errors are detected the *Script* is not evaluated.

Static Semantics: IsStrict

ScriptBody **:** *OuterStatementList*

1. If this *ScriptBody* is contained in strict code or if *OuterStatementList* is strict code, then return **true**. Otherwise, return **false**.

Static Semantics: LexicallyDeclaredNames

OuterStatementList **:** *OuterStatementList OuterItem*

1. Let *names* be LexicallyDeclaredNames of *OuterStatementList*.
2. Append to *names* the elements of the LexicallyDeclaredNames of *OuterItem*.
3. Return *names*.

OuterItem **:** *ModuleDeclaration*

1. Return the BoundNames of *ModuleDeclaration*.

OuterItem **:** *ImportDeclaration*

1. Return the BoundNames of *ImportDeclaration*.

OuterItem **:** *StatementListItem*

1. Return TopLevelLexicallyDeclaredNames of *StatementListItem*.

NOTE At the top level of a *Script*, function declarations are treated like var declarations rather than like lexical declarations.

Static Semantics: LexicallyScopedDeclarations

OuterStatementList **:** *OuterStatementList OuterItem*

1. Let *declarations* be LexicallyScopedDeclarations of *OuterStatementList*.
2. Append to *declarations* the elements of the LexicallyScopedDeclarations of *OuterItem*.
3. Return *declarations*.

OuterItem **:** *ModuleDeclaration*

1. Return a new List containing *ModuleDeclaration*.

OuterItem **:** *ImportDeclaration*

1. Return a new List containing *ImportDeclaration*.

OuterItem **:** *StatementListItem*

1. Return TopLevelLexicallyScopedDeclarations of *StatementListItem*.

Static Semantics: VarDeclaredNames

OuterStatementList **:** *OuterStatementList OuterItem*

1. Let *names* be VarDeclaredNames of *OuterStatementList*.
2. Append to *names* the elements of the VarDeclaredNames of *OuterItem*.
3. Return *names*.

OuterItem **:** *ModuleDeclaration*

2. Return an empty List.

OuterItem **:** *ImportDeclaration*

2. Return an empty List.

OuterItem **:** *StatementListItem*

2. Return TopLevelVarDeclaredNames of *StatementListItem*.

Static Semantics: VarScopedDeclarations

OuterStatementList **:** *OuterStatementList OuterItem*

1. Let *declarations* be VarScopedDeclarations of *OuterStatementList*.
2. Append to *declarations* the elements of the VarScopedDeclarations of *OuterItem*.
3. Return *declarations*.

OuterItem **:** *ModuleDeclaration*

1. Return a new empty List.

OuterItem **:** *ImportDeclaration*

1. Return a new empty List.

OuterItem **:** *StatementListItem*

1. Return the TopLevelVarScopedDeclarations of *StatementListItem*.

Runtime Semantics

Runtime Semantics: Script Evaluation

With argument *realm* and *deletableBindings*.

Script **:** *ScriptBody*$_{opt}$

1. The code of this *Script* is strict mode code if the Directive Prologue (14.1) of its *ScriptBody* contains a Use Strict Directive or if any of the conditions of 10.1.1 apply. If the code of this *Script* is strict mode code, *ScriptBody* is evaluated in the following steps as strict mode code. Otherwise *ScriptBody* is evaluated in the following steps as non-strict mode code.
2. If *ScriptBody* is not present, return NormalCompletion(empty).
3. Let *globalEnv* be *realm*.[[globalEnv]].
4. Let *status* be the result of performing Global Declaration Instantiation as described in 10.5.1 using *ScriptBody*, *globalEnv*, and *deletableBindings* as arguments.
5. ReturnIfAbrupt(*status*).
6. Let *progCxt* be a new ECMAScript code execution context.
7. Set the *progCxt's* Realm to *realm*.
8. Set the *progCxt's* VariableEnvironment to *globalEnv*.
9. Set the *progCxt's* LexicalEnvironment to *globalEnv*.
10. If there is a currently running execution context, suspend it.
11. Push *progCxt* on to the execution context stack; *progCxt* is now the running execution context.
12. Let *result* be the result of evaluating *ScriptBody*.
13. Suspend *progCxt* and remove it from the execution context stack.
14. If the execution context stack is not empty, resume the context that is now on the top of the execution context stack as the running execution context. Otherwise, the execution context stack is now empty and there is no running execution context.
15. Return *result*.

NOTE The processes for initiating the evaluation of a *Script* and for dealing with the result of such an evaluation are defined by an ECMAScript implementation and not by this specification.

Runtime Semantics: Evaluation

OuterStatementList **:** *OuterStatementList OuterItem*

1. Let *sl* be the result of evaluating *OuterStatementList*.
2. ReturnIfAbrupt(*sl*).
3. Let *s* be the result of evaluating *OuterItem*.
4. If *s*.[[type]] is throw, return *s*.
5. If *s*.[[value]] is empty, let *V* = *sl*.[[value]], otherwise let *V* = *s*.[[value]].
6. Return Completion {[[type]]: *s*.[[type]], [[value]]: *V*, [[target]]: *s*.[[target]]}.

NOTE See the 12.1 NOTE regarding evaluation of *StatementList* **:** *StatementList StatementListItem*.

14.1.1 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 *ScriptBody* or *FunctionBody* and where each *ExpressionStatement* in the sequence consists entirely of a *StringLiteral* token followed by 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*.

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.

14.2 Modules

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 [[BuiltinBrand]] internal data property with value BuiltinFunction if that built-in object has a [[Call]] internal method. Unless specified otherwise, the [[Extensible]] internal data 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 data 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 data 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.

Every built-in Function object, *F*, described in this clause—whether as a constructor, an ordinary function, or both—has the properties that are defined by performing the following step when the function object is created:

1. Perform the AddRestrictedFunctionProperties (13.6.3) abstract operation with argument *F*.

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 data property described in this clause has the attributes { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true** } unless otherwise specified.

Every accessor property described in this clause has the attributes {[[Enumerable]]: **false**, [[Configurable]]: **true** } unless otherwise specified. If only a get accessor function is described, the set accessor function is the default value, **undefined**. If only a set accessor is function is described the get accessor is the default value, **undefined**.

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 method; 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 method; it is not possible to invoke the global object as a function.

The value of the [[Prototype]] internal data 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.

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:

4. If Type(*x*) is not String, return *x*.
5. Let *script* be the ECMAScript code that is the result of parsing *x*, interpreted as UTF-16 encoded Unicode text as described in 8.4, for the goal symbol *Script*. If the parse fails or any early errors are detected, throw a **SyntaxError** exception (but see also clause 16).
6. If *script* Contains *ScriptBody* is **false**, return **undefined**.
7. Let *strictScript* be IsStrict of *script*.
8. If this is a direct call to eval (15.1.2.1.1), let *direct* be **true**, otherwise let *direct* be **false**.
9. If *direct* is **true** and the code that made the direct call to eval is strict code, then let *strictCaller* be **true**. Otherwise, let *strictCaller* be **false**.
10. Let *ctx* be the running execution context. If *direct* is **true** *ctx* will be the execution context that performed the direct eval. If *direct* is **false** *ctx* will be the execution context for the invocation of the eval function.
11. Let *evalRealm* be *ctx's* Realm.
12. If *direct* is **false** and *strictScript* is **false**, then
    a. Return the result of Script Evaluation for *script* with arguments *evalRealm* and **true**.
13. If *direct* is **true**, *strictScript* is **false**, *strictCaller* is **false**, and ctx's LexicalEnvironment is the same as *evalRealm*.[[globalEnv]], then
    a. Return the result of Script Evaluation for *script* with arguments *evalRealm* and **true**.
14. If *direct* is **true**, then
    a. If the code that made the direct call to eval is function code and ValidInFuction of *script* is **false**, then throw a SyntaxError exception.

   b. If the code that made the direct call to eval is module code and ValidInModule of *script* is **false**, then throw a SyntaxError exception.
15. If *direct* is **true**, then
   a. Let *lexEnv* be *ctx's* LexicalEnvironment.
   b. Let *varEnv* be *ctx's* VariableEnvironment.
16. Else,
   a. Let *lexEnv* be *evalRealm*.[[globalEnv]].
   b. Let *varEnv* be *evalRealm*.[[globalEnv]].
17. If *strictScript* is **true** or if *direct* is **true** and *strictCaller* is **true** , then
   a. Let *strictVarEnv* be the result of calling NewDeclarativeEnvironment passing the *lexEnv* as the argument.
   b. Let *lexEnv* be *strictVarEnv*.
   c. let *varEnv* be *strictVarEnv*.
18. Let *status* be the result of performing Eval Declaration Instantiation as described in 10.5.5 with *script*, *varEnv*, and *lexEnv*.
19. ReturnIfAbrupt(*status*).
20. Let *evalCxt* be a new ECMAScript code execution context.
21. Set the *evalCxt's* Realm to *evalRealm*.
22. Set the *evalCxt's* VariableEnvironment to *varEnv*.
23. Set the *evalCxt's* LexicalEnvironment to *lexEnv*.
24. If there is a currently running execution context, suspend it.
25. Push *evalCxt* on to the execution context stack; *evalCxt* is now the running execution context.
26. Let *result* be the result of evaluating *script*.
27. Suspend *evalCxt* and remove it from the execution context stack.
28. Resume the context that is now on the top of the execution context stack as the running execution context.
29. Return *result*.

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.

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* = ToInt32(*radix*).
8. ReturnIfAbrupt(*R*).
9. Let *stripPrefix* be **true**.
10. If *R* ≠ 0, then
    a. If *R* < 2 or *R* > 36, then return **NaN**.
    b. If *R* ≠ 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* × *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**.

NOTE A reliable way for ECMAScript code to test if a value `X` is a **NaN** is an expression of the form `X !== 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**.

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 **:::**
uriCharacters$_{opt}$

uriCharacters **:::**
uriCharacter uriCharacters$_{opt}$

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.

Runtime Semantics

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`".

Runtime Semantics: Encode Abstract Operation

The encoding and escaping process is described by the abstract operation Encode taking two String arguments *string* and *unescapedSet*.

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 *unescapedSet*, 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 *unescapedSet*,
        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.

Runtime Semantics: Decode Abstract Operation

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 × (*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* within *string* 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* < 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* ≥ 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>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 29.

Table 29 — UTF-8 Encodings

Code Unit Value	*Representation*	*1st Octet*	*2nd Octet*	*3rd Octet*	*4th Octet*
0x0000 – 0x007F	00000000 0*zzzzzzz*	0*zzzzzzz*			
0x0080 – 0x07FF	00000*yyy yyzzzzzz*	110*yyyyy*	10*zzzzzz*		
0x0800 – 0xD7FF	*xxxxyyyy yyzzzzzz*	1110*xxxx*	10*yyyyyy*	10*zzzzzz*	
0xD800 – 0xDBFF followed by 0xDC00 – 0xDFFF	110110*vv vvwwwwxx* followed by 110111*yy yyzzzzzz*	11110*uuu*	10*uuwwww*	10*xxyyyy*	10*zzzzzz*
0xD800 – 0xDBFF not followed by 0xDC00 – 0xDFFF	causes URIError				
0xDC00 – 0xDFFF	causes URIError				
0xE000 – 0xFFFF	*xxxxyyyy yyzzzzzz*	1110*xxxx*	10*yyyyyy*	10*zzzzzz*	

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 0xD800-0xDFFF 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 *unescapedURISet* be a String containing one instance of each character valid in *uriReserved* and *uriUnescaped* plus "**#**".
4. Return the result of calling Encode(*uriString*, *unescapedURISet*)

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 *unescapedURIComponentSet* be a String containing one instance of each character valid in *uriUnescaped*.
4. Return the result of calling Encode(*componentString*, *unescapedURIComponentSet*)

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.

15.1.4.16 Map (. . .)

See 15.14.3.

15.1.4.17 WeakMap (. . .)

See 15.15.3.

15.1.4.18 Set (. . .)

See 15.16.3.

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

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 abstract 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 return *value*.
    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 abstract operation ObjectCreate.

15.2.3 Properties of the Object Constructor

The value of the [[Prototype]] internal data property of the Object constructor is the standard built-in Function prototype object.

Besides 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 result of calling the [[GetInheritence]] internal method 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.
2. Let *key* be ToPropertyKey(*P*).
3. ReturnIfAbrupt(*name*).
4. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument *key*.
5. Return the result of calling FromPropertyDescriptor(*desc*) (8.2.5.4).

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 *keys* be the result of calling the [[OwnPropertyKEYS]] internl method of *O*.
3. ReturnIfAbrupt(*keys*).
4. Return CreateArrayFromList(*keys*);.

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 abstract 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*.

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 ToPropertyKey(*P*).
3. ReturnIfAbrupt(*name*).
4. Let *desc* be the result of calling ToPropertyDescriptor with *Attributes* as the argument.
5. ReturnIfAbrupt(*desc*).
6. Let *success* be the result of DefinePropertyOrThrow(*O*,*name*, *desc*).
7. ReturnIfAbrupt(*success*).
8. 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*.

Runtime Semantics: ObjectDefineProperties Abstract Operation

The abstract operation 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 keys 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 Get(*props*, *P*).
    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. Let *pendingException* be **undefined**.
7. 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 DefinePropertyOrThrow(*O*,*P*, *desc*).
    d. If *status* is an Abrupt Completion then,
        i. If pendingException is undefined, then set *pendingException* to *status*.
8. ReturnIfAbrupt(*pendingException*).
9. 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.

NOTE An exception in defining an individual property in step 7 does not terminate the process of defining other properties. All valid property definitions are processed.

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. Let *status* be the result of MakeSecuure(*O*, **false**).
3. ReturnIfAbrupt(*status*).
4. Return *O*.

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. Let *status* be the result of MakeSecuure(*O*, **true**).
3. ReturnIfAbrupt(*status*).
4. Return *O*.

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. Let *status* be the result of calling the [[PreventExtensions]] internal method of *O*.
3. ReturnIfAbrupt(*status*).
4. 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. Return TestIfSecureObject(*O*, **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. Return TestIfSecureObject(*O*, **true**).

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 result of calling the [[GetExtensible]] internal method 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 Type(*O*) is not Object, throw a **TypeError** exception.
2. Let *keys* be the result of calling the [[Keys]] internal method of *O*.
3. ReturnIfAbrupt(*keys*).
4. Return CreateArrayFromList(*keys*).

15.2.3.15 Object.assign (target, source)

TODO :

- Only enumerable own properties of source
- Invoke [[Get]] on property list derived from source, for each property in list [[Put]] on target
- private names are not copied
- unique names are copied
- super mechanism (rebind super)
- Returns modified "target"

15.2.4 Properties of the Object Prototype Object

The value of the [[Prototype]] internal data property of the Object prototype object is **null** and the initial value of the [[Extensible]] internal data property is **true**.

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* is an exotic Symbol object, then let *tag* be `"Symbol"`.
5. Else if *O* has a [[BuiltinBrand]] internal data property, let *tag* be the corresponding value from Table 30.
6. Else
    a. Let *hasTag* be the result of HasProperty(*O*, @@toStringTag).

b. ReturnIfAbrupt(*hasTag*).
c. If *hasTag* is **false**, then let *tag* be **"Object"**.
d. Else,
   i. Let *tag* be the result of Get(*O*, @@toStringTag).
   ii. If *tag* is an abrupt completion, let *tag* be NormalCompletion(**"???"**).
   iii. Let *tag* be *tag*.[[value]].
   iv. If Type(*tag*) is not String, let *tag* be **"???"**.
   v. If *tag* is any of **"Arguments"**, **"Array"**, **"Boolean"**, **"Date"**, **"Error"**, **"Function"**, **"JSON"**, **"Math"**, **"Number"**, **"Object"**, **"RegExp"**, or **"String"** then let *tag* be the string value **"~"** concatenated with the current value of *tag*.

7. Return the String value that is the result of concatenating the three Strings "[object ", tag, and "]".

Table 30 — [[BuiltinBrand]] Tag Values

[[BuiltinBrand]] Value	*tag* Value
BuiltinFunction	**"Function"**
BuiltinArray	**"Array"**
BuiltinStringWrapper	**"String"**
BuiltinBooleanWrapper	**"Boolean"**
BuiltinNumberWrapper	**"Number"**
BuiltinMath	**"Math"**
BuiltinDate	**"Date"**
BuiltinRegExp	**"RegExp"**
BuiltinError	**"Error"**
BuiltinJSON	**"JSON"**
BuiltinArguments	**"Arguments"**

NOTE Historically, this function was occasionally used to access the string value of the [[Class]] internal data property that was used in previous editions of this specification as a nominal type tag for various built-in objects. This definition of **toString** preserves the ability to use it as a reliable test for those specific kinds of built-in objects but it does not provide a reliable type testing mechanism for other kinds of built-in or program defined objects.

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 Invoke(*O*, **"toString"**).

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. Return *O*.

15.2.4.5 Object.prototype.hasOwnProperty (V)

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

1. Let *P* be ToPropertyKey(*V*).

2. ReturnIfAbrupt(*P*).
3. Let *O* be the result of calling ToObject passing the **this** value as the argument.
4. ReturnIfAbrupt(*O*).
5. Return the result of calling the [[HasOwnProperty]] internal method of *O* passing *P* as the argument.

NOTE The ordering of steps 1 and 3 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 result of calling the [[GetInheritence]] internal method of *V* with no arguments.
    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 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 (p1, p2, … , pn, body)

When the **`Function`** function is called with some arguments *p1*, *p2*, … , *pn*, *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 (p1, p2, … , pn, 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 *p1*, *p2*, … , *pn*, *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*, interpreted as UTF-16 encoded Unicode text as described in 8.4, using *FormalParameterList* as the goal symbol. Throw a **SyntaxError** exception if the parse fails.
9. Let *body* be the result of parsing *bodyText*, interpreted as UTF-16 encoded Unicode text as described in 8.4, 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 13.6 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")
```

15.3.3 Properties of the Function Constructor

The Function constructor is itself a Function object and has a [[BuiltinBrand]] internal data property whose value is BuiltinFunction. The value of the [[Prototype]] internal data property of the Function constructor is the standard built-in Function prototype object (15.3.4).

The value of the [[Extensible]] internal data 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 [[BuiltinBrand]] internal data property whose value is BuiltinFunction . When invoked, it accepts any arguments and returns **undefined**.

The value of the [[Prototype]] internal data property of the Function prototype object is the standard built-in Object prototype object (15.2.4). The initial value of the [[Extensible]] internal data property of the Function prototype object is **true**.

The function prototype object does not have a **`prototype`** property.

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 *thisArgument* and an empty List of arguments as *argumentsList*.
3. If Type(*argArray*) is not Object, then throw a **TypeError** exception.
4. Let *len* be the result of Get(*argArray*, **`"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 Get(*argArray*, *indexName*).
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 *thisArgument* and *argList* as *argumentsList*.

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 *thisArgument* and *argList* as *argumentsList*.

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 consisting of all of the argument values provided after *thisArg* (*arg1*, *arg2* etc), in order.
4. Let *F* be the result of the abstract operation BoundFuctionCreate with arguments *Target*, *thisArg*, and *A*.
5. If *Target* has the [[BuiltinBrand]] internal property with value BuiltinFunction, then
   a. Let *targetLen* be the result of Get(*Target*, **`"length"`**).
   b. ReturnIfAbrupt(*targetLen*).
   c. Let *L* be the larger of 0 and the result of *targetLen* minus the number of elements of *A*.
6. Else let *L* be 0.
7. Call the [[DefineOwnProperty]] internal method of *F* with arguments **`"length"`** and PropertyDescriptor {[[value]]: *L*, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false**}.
8. Perform the AddRestrictedFunctionProperties abstract operation with argument *F*.
9. Return *F*.

The **`length`** property of the **`bind`** method is **1**.

NOTE Function objects created using **`Function.prototype.bind`** are exotic objects. They also do not have a **`prototype`** property.

15.3.4.6 @@hasInstance (V)

When the @@hasInstance method of an object *F* is called with value *V*, the following steps are taken:

1. Let *F* be the **this** value.
2. Return the result of OrdinaryHasInstance(*F*, *V*);

15.3.5 Properties of Function Instances

In addition to the required internal properties, every function instance has a [[Call]] internal method and in most cases uses a different version of the [[Get]] internal method. Depending on how they are created (see 8.6.2, 13.6, 15, and 15.3.4.5), function instances may have a [[Scope]] internal data property, a [[Construct]] internal method, a [[FormalParameters]] internal data property, a [[Code]] internal data property, a [[BoundTargetFunction]] internal data property, a [[BoundThis]] internal data property, and a [[BoundArguments]] internal data property.

Every function instance has a [[BuiltinBrand]] internal data property whose value is BuiltinFunction.

Function instances that correspond to strict mode functions (13.6) 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.

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 data property of a newly created ordinay 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.

15.4 Array Objects

Array objects are exotic objects that give special treatment to a certain class of property names. See 8.4.2 for a definition of this special treatment.

An Array object, *O*, is said to be *sparse* if the following algorithm returns **true**:

1. Let *len* be the result of Get(*O*, **"length"**).
2. For each integer *i* in the range 0≤*i*<ToUint32(*len*)
    a. Let *elem* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument ToString(*i*).
    b. If *elem* is **undefined**, return **true**.
3. Return **false**.

Runtime Semantics: ArrayCreate Abstract Operation

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:

10. Let *A* be a newly created ECMAScript object.
11. Set *A*'s common internal methods except for [[DefineOwnProperty]] to the default definitions for ordinary objects specified in 8.3.
12. Set the [[Prototype]] internal data property of *A* to the intrinsic object %ArrayPrototype%.
13. Set *A*'s [[DefineOwnProperty]] internal method to the definition given in 15.4.5.1.

14. Set the [[BuiltinBrand]] internal data property of *A* to the value BuiltinArray.
15. Set the [[Extensible]] internal data property of *A* to **true**.
16. Call the ordinary object [[DefineOwnProperty]] internal method (8.3.10) on *A* with arguments **"length"** and Property Descriptor {[[Value]]: *length*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false**}.
17. 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.

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.

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.

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 abstract operation ArrayCreate with argument *len*.
3. ReturnIfAbrupt(*array*).
4. Let *k* be 0.
5. Let *items* be a zero-origined List contain the argument items in order.
6. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *itemK* be k^{th} element of *items*.
    c. Call the [[DefineOwnProperty]] internal method of *array* with arguments *Pk* and Property Descriptor {[[Value]]: *itemK*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
    d. Increase *k* by 1.
7. Let *putStatus* be the result of Put(*array*, **"length"**, *len*, **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 abstract operation ArrayCreate with argument 1.
    b. ReturnIfAbrupt(*array*).
    c. Let *defineStatus* be the result of DefinePropertyOrThrow(*array*, **"0"**, Property Descriptor {[[Value]]: *len*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}).
    d. ReturnIfAbrupt(*defineStatus*).
    e. Return *array*.
2. Let *intLen* be ToUint32(*len*).
3. If *intLen* ≠ *len*, then throw a RangeError exception.
4. Let *array* be the result of the abstract operation ArrayCreate with argument *intLen*.
5. Return *array*.

15.4.3 Properties of the Array Constructor

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

Besides 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 [[BuiltinBrand]] internal property is BuildinArray; otherwise it returns **false**. The following steps are taken:

1. If Type(*arg*) is not Object, return **false**.
2. If *arg* has the [[BuiltinBrand]] internal property with value BuiltinArray, 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:

1. Let *lenValue* be the result of Get(*items*, **`"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 abstract operation ArrayCreate 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 Get(*items*, *Pk*).
    c. Let *defineStatus* be the result DefinePropertyOrThrow(*A*,*Pk*, Property Descriptor {[[Value]]: *kValue*.[[value]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}).
    d. ReturnIfAbrupt(*defineStatus*).
    e. Increase *k* by 1.
9. Let *putStatus* be the result of Put(*A*, **`"length"`**, *len*, **true**).
10. ReturnIfAbrupt(*putStatus*).
11. Return *A*.

The **`length`** property of the **`of`** method is **0**.

NOTE 1 The *items* argument is assume 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.

15.4.3.4 Array.from (arrayLike)

When the **`from`** method is called with argument *arrayLike*, the following steps are taken:

1. Let *items* be ToObject(*arrayLike*).
2. ReturnIfAbrupt(*items*).
3. Let *lenValue* be the result of Get(*items*, **"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 abstract operation ArrayCreate 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 HasProperty(*items*, *Pk*).
    c. ReturnIfAbrupt(*kPresent*).
    d. If *kPresent* is **true**, then
        i. Let *kValue* be the result of Get(*items*, *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*).
    e. Increase *k* by 1.
12. Let *putStatus* be the result of Put(*A*, **"length"**, *len*, **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 inherited 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 data 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 [[BuiltinBrand]] internal data property with value BuiltinArray, 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 [[BuiltinBrand]] internal data property with value BuiltinArray.

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 Get(*array*, **"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 *thisArgument* and an empty List as *argumentsList*.

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 an exotic object that is not an Array 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 Get(*array*, **"length"**).
4. Let *len* be ToUint32(*arrayLen*).
5. ReturnIfAbrupt(*len*).
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 Get(*array*, **"0"**).
9. Let *noArgs* be an empty List.
10. ReturnIfAbrupt(*firstElement*).
11. If *firstElement* is **undefined** or **null**, then
    a. Let *R* be the empty String.
12. Else
    a. Let *R* be the result of Invoke(*firstElement*, **"toLocaleString"**).
    b. Let *R* be ToString(*R*).
    c. ReturnIfAbrupt(*R*).
13. Let *k* be **1**.
14. Repeat, while *k* < *len*
    a. Let *S* be a String value produced by concatenating *R* and *separator*.
    b. Let *nextElement* be the result of Get(*array*,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 Invoke(*nextElement*, **"toLocaleString"**).
        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.
15. 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 an exotic object that is not an Array 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 abstract operation ArrayCreate 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 [[BuiltinBrand]] internal data property with value BuiltinArray, then
        i. Let *k* be 0.
        ii. Let *len* be the result of Get(*E*, **`"length"`**).
        iii. ReturnIfAbrupt(*len*).
        iv. Repeat, while *k* < *len*
            1. Let *P* be ToString(*k*).
            2. Let *exists* be the result of HasProperty(*E*, *P*).
            3. ReturnIfAbrupt(*exists*).
            4. If *exists* is **true**, then
                a Let *subElement* be the result of Get(*E*, *P*).
                b Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*n*) and Property Descriptor {[[Value]]: *subElement*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
            5. Increase *n* by 1.
            6. Increase *k* by 1.
    c. Else *E* is not an Array,
        i. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*n*) and Property Descriptor {[[Value]]: *E*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
        ii. Increase *n* by 1.
7. Let *putStatus* be the result of Put(*A*, **`"length"`**, *n*, **true**).
8. ReturnIfAbrupt(*putStatus*).
9. Return *A*.

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

NOTE 1 The explicit setting of the **`length`** property in step 7 is necessary to ensure that its value is correct in situations where the trailing elements of the result Array are not present.

NOTE 2 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 an exotic object that is not an Array 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 Get(*O*, **`"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 Get(*O*, `"0"`).
10. If *element0* is **undefined** or **null**, let *R* be the empty String; otherwise, let *R* be ToString(*element0*).
11. ReturnIfAbrupt(*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 Get(*O*, ToString(*k*)).
    c. If *element* is **undefined** or **null**, then 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 an exotic object that is not an Array is implementation-dependent.

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 Get(*O*, `"length"`).
4. Let *len* be ToUint32(*lenVal*).
5. ReturnIfAbrupt(*len*).
6. If *len* is zero,
    a. Let *putStatus* be the result of Put(*O*, `"length"`, 0, **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 Get(*O*, *indx*).
    d. ReturnIfAbrupt(*element*).
    e. Let *deleteStatus* be the result of DeletePropertyOrThrow(*O*, *indx*).
    f. ReturnIfAbrupt(*deleteStatus*).
    g. Let *putStatus* be the result of Put(*O*, `"length"`, *newLen*, **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 an exotic object that is not an Array 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 Get(*O*, **"length"**).
4. Let *n* be ToUint32(*lenVal*).
5. ReturnIfAbrupt(*n*).
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 Put(*O*, ToString(*n*), *E*, **true**).
    c. ReturnIfAbrupt(*putStatus*).
    d. Increase *n* by 1.
8. Let *putStatus* be the result of Put(*O*, **"length"**, *n*, **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 an exotic object that is not an Array 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 Get(*O*, **"length"**).
4. Let *len* be ToUint32(*lenVal*).
5. ReturnIfAbrupt(*len*).
6. Let *middle* be floor(*len*/2).
7. Let *lower* be **0**.
8. Repeat, while *lower* ≠ *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 Get(*O*, *lowerP*).
    e. ReturnIfAbrupt(*lowerValue*).
    f. Let *upperValue* be the result of Get(*O*, *upper*).
    g. ReturnIfAbrupt(*upperValue*).
    h. Let *lowerExists* be the result of HasProperty(*O*, *lowerP*).
    i. ReturnIfAbrupt(*lowerExists*).
    j. Let *upperExists* be the result of HasProperty(*O*, *upperP*).
    k. ReturnIfAbrupt(*upperExists*).
    l. If *lowerExists* is **true** and *upperExists* is **true**, then
        i. Let *putStatus* be the result of Put(*O*, *lowerP*, *upperValue*, **true**).
        ii. ReturnIfAbrupt(*putStatus*).
        iii. Let *putStatus* be the result of Put(*O*, *upperP*, *lowerValue*, **true**).
        iv. ReturnIfAbrupt(*putStatus*).
    m. Else if *lowerExists* is **false** and *upperExists* is **true**, then
        i. Let *putStatus* be the result of Put(*O*, *lowerP*, *upperValue*, **true**).
        ii. ReturnIfAbrupt(*putStatus*).
        iii. Let *deleteStatus* be the result of DeletePropertyOrThrow (*O*, *upperP*).
        iv. ReturnIfAbrupt(*deleteStatus*).
    n. Else if *lowerExists* is **true** and *upperExists* is **false**, then
        i. Let *deleteStatus* be the result of DeletePropertyOrThrow (*O*, *lowerP*).
        ii. ReturnIfAbrupt(*deleteStatus*).

iii. Let *putStatus* be the result of Put(*O*, *upperP*, *lowerValue*, **true**).
iv. ReturnIfAbrupt(*putStatus*).
o. Else both *lowerExists* and *upperExists* are **false**,
i. No action is required.
p. 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 an exotic object that is not an Array is implementation-dependent.

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 Get(*O*, **`"length"`**).
4. Let *len* be ToUint32(*lenVal*).
5. ReturnIfAbrupt(*len*).
6. If *len* is zero, then
   a. Let *putStatus* be the result of Put(*O*, **`"length"`**, 0, **true**).
   b. ReturnIfAbrupt(*putStatus*).
   c. Return **undefined**.
7. Let *first* be the result of Get(*O*, **`"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 HasProperty(*O*, *from*).
    d. ReturnIfAbrupt(*fromPresent*).
    e. If *fromPresent* is **true**, then
       i. Let *fromVal* be the result of Get(*O*, *from*).
       ii. ReturnIfAbrupt(*fromVal*).
       iii. Let *putStatus* be the result of Put(*O*, *to*, *fromVal*, **true**).
       iv. ReturnIfAbrupt(*putStatus*).
    f. Else *fromPresent* is **false**,
       i. Let *deleteStatus* be the result of DeletePropertyOrThrow(*O*, *to*).
       ii. ReturnIfAbrupt(*deleteStatus*).
    g. Increase *k* by 1.
11. Let *deleteStatus* be the result of DeletePropertyOrThrow(*O*, ToString(*len*–1)).
12. ReturnIfAbrupt(*deleteStatus*).
13. Let *putStatus* be the result of Put(*O*, **`"length"`**, *len*–1, **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 an exotic object that is not an Array 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 abstract operation ArrayCreate with argument 0.
4. Let *lenVal* be the result of Get(*O*, **"length"**).
5. Let *len* be ToUint32(*lenVal*).
6. ReturnIfAbrupt(*len*).
7. Let *relativeStart* be ToInteger(*start*).
8. ReturnIfAbrupt(*relativeStart*).
9. If *relativeStart* is negative, let *k* be max((*len* + *relativeStart*),0); else let *k* be min(*relativeStart*, *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((*len* + *relativeEnd*),0); else let *final* be min(*relativeEnd*, *len*).
13. Let *n* be 0.
14. Repeat, while *k* < *final*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of HasProperty(*O*, *Pk*).
    c. ReturnIfAbrupt(*kPresent*).
    d. If *kPresent* is **true**, then
        i. Let *kValue* be the result of Get(*O*, *Pk*).
        ii. ReturnIfAbrupt(*kValue*).
        iii. Let *status* be the result of CreateOwnDataProperty(*A*, ToString(*n*), *kValue*)..
        iv. ReturnIfAbrupt(*status*).
        v. If *status* is **false**, throw a **TypeError** exception.
    e. Increase *k* by 1.
    f. Increase *n* by 1.
15. Let *putStatus* be the result of Put(*A*, **"length"**, *final*, **true**).
16. ReturnIfAbrupt(*putStatus*).
17. Return *A*.

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

NOTE 1 The explicit setting of the **length** property of the result Array in step 15 is necessary to ensure that its value is correct in situations where the trailing elements of the result Array are not present.

NOTE 2 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 an exotic object that is not an Array 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 Get(*O*, **"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 result of calling the [[GetInheritence]] internal method 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)
- $0 \leq j < len$
- The result of HasProperty(*proto*, 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 result of calling the [[IsExtensible]] internal method 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.

1. Perform an implementation-dependent sequence of calls to the [[GetP]] and [[SetP]] internal methods of *obj*, to the DeletePropertyOrThrow abstract operation with *obj* as the first argument, and to SortCompare (described below), where the property key argument for each call to [[GetP]], [[SetP]], or DeletePropertyOrThrow is the string representation of a nonnegative integer less than *len* and where the arguments for calls to SortCompare are results of previous calls to the [[GetP]] internal method. If *obj* is not sparse then DeletePropertyOrThrow must not be called. If any [[SetP]] call returns **false** a **TypeError** exception is thrown. 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 π of the nonnegative integers less than *len*, such that for every nonnegative integer *j* less than *len*, if property old[*j*] existed, then new[π(*j*)] is exactly the same value as old[*j*]**,**. But if property old[*j*] did not exist, then new[π(*j*)] does not exist.
- Then for all nonnegative integers *j* and *k*, each less than *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 [[GetP]] 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 [[GetP]] 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 $a <_{CF} b$ means *comparefn*(*a*,*b*) < 0; $a =_{CF} b$ means *comparefn*(*a*,*b*) = 0 (of either sign); and $a >_{CF} b$ means *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 $a <_{CF} b$, $a =_{CF} b$, and $a >_{CF} b$ will be true for a given pair of *a* and *b*.
- Calling *comparefn*(*a*,*b*) does not modify the **this** object.
- $a =_{CF} a$ (reflexivity)
- If $a =_{CF} b$, then $b =_{CF} a$ (symmetry)
- If $a =_{CF} b$ and $b =_{CF} c$, then $a =_{CF} c$ (transitivity of $=_{CF}$)
- If $a <_{CF} b$ and $b <_{CF} c$, then $a <_{CF} c$ (transitivity of $<_{CF}$)
- If $a >_{CF} b$ and $b >_{CF} c$, then $a >_{CF} c$ (transitivity of $>_{CF}$)

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.

Runtime Semantics: SortCompare Abstract Operation

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 HasProperty(*obj*, *jString*).
4. ReturnIfAbrupt(*hasj*).
5. Let *hask* be the result of HasProperty(*obj*, *kString*).
6. ReturnIfAbrupt(*hask*).

7. If *hasj* and *hask* are both **false**, then return **+0**.
8. If *hasj* is **false**, then return 1.
9. If *hask* is **false**, then return –1.
10. Let *x* be the result of Get(*obj*,*jString*).
11. ReturnIfAbrupt(*x*).
12. Let *y* be the result of Get(*obj*, *kString*).
13. ReturnIfAbrupt(*y*).
14. If *x* and *y* are both **undefined**, return **+0**.
15. If *x* is **undefined**, return 1.
16. If *y* is **undefined**, return –1.
17. 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 *thisArgument* and with a List contain the values of *x* and *y* as the *argumentsList*.
18. Let *xString* be ToString(*x*).
19. ReturnIfAbrupt(*xString*).
20. Let *yString* be ToString(*y*).
21. ReturnIfAbrupt(*yString*).
22. If *xString* < *yString*, return –1.
23. If *xString* > *yString*, return 1.
24. 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 an exotic object that is not an Array 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 abstract operation ArrayCreate with argument 0.
4. Let *lenVal* be the result of Get(*O*, **`"length"`**)
5. Let *len* be ToUint32(*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 HasProperty(*O*, *from*).
    c. ReturnIfAbrupt(*fromPresent*).
    d. If *fromPresent* is **true**, then
       i. Let *fromValue* be the result of Get(*O*, *from*).
       ii. ReturnIfAbrupt(*fromValue*).
       iii. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*k*) and Property Descriptor {[[Value]]: *fromValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
    e. Increment *k* by 1.
13. Let *putStatus* be the result of Put(*A*, **`"length"`**, *actualDeleteCount*, **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 HasProperty(*O*, *from*).
        iv. ReturnIfAbrupt(*fromPresent*).
        v. If *fromPresent* is **true**, then
            1. Let *fromValue* be the result of Get(*O*, *from*).
            2. ReturnIfAbrupt(*fromValue*).
            3. Let *putStatus* be the result of Put(*O*, *to*, *fromValue*, **true**).
            4. ReturnIfAbrupt(*putStatus*).
        vi. Else *fromPresent* is **false**,
            1. Let *deleteStatus* be the result of DeletePropertyOrThrow(*O*, *to*).
            2. ReturnIfAbrupt(*deleteStatus*).
        vii. Increase k by 1.
    c. Let *k* be *len*.
    d. Repeat, while *k* > (*len* – *actualDeleteCount* + *itemCount*)
        i. Let *deleteStatus* be the result of DeletePropertyOrThrow(*O*, ToString(*k*–1)).
        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 HasProperty(*O*, *from*).
        iv. ReturnIfAbrupt(*fromPresent*).
        v. If *fromPresent* is **true**, then
            1. Let *fromValue* be the result of Get(*O*, *from*).
            2. ReturnIfAbrupt(*fromValue*).
            3. Let *putStatus* be the result of Put(*O*, *to*, *fromValue*, **true**).
            4. ReturnIfAbrupt(*putStatus*).
        vi. Else *fromPresent* is **false**,
            1. Let *deleteStatus* be the result of DeletePropertyOrThrow(*O*, *to*).
            2. ReturnIfAbrupt(*deleteStatus*).
        vii. 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 ofPut(*O*, ToString(*k*), *E*, **true**).
    c. ReturnIfAbrupt(*putStatus*).
    d. Increase *k* by 1.
21. Let *putStatus* be the result of Put(*O*, **"length"**, *len* – *actualDeleteCount* + *itemCount*, **true**).
22. ReturnIfAbrupt(*putStatus*).
23. Return *A*.

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

NOTE 1 The explicit setting of the **length** property of the result Array in step 13 is necessary to ensure that its value is correct in situations where its trailing elements are not present.

NOTE 2 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 an exotic object that is not an Array 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 Get(*O*, **`"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 HasProperty(*O*, *from*).
    d. ReturnIfAbrupt(*fromPresent*).
    e. If *fromPresent* is **true**, then
        i. Let *fromValue* be the result of Get(*O*, *from*).
        ii. ReturnIfAbrupt(*fromValue*).
        iii. Let *putStatus* be the result of Put(*O*, *to*, *fromValue*, **true**).
        iv. ReturnIfAbrupt(*putStatus*).
    f. Else *fromPresent* is **false**,
        i. Let *deleteStatus* be the result of DeletePropertyOrThrow(*O*, *to*).
        ii. ReturnIfAbrupt(*deleteStatus*).
    g. 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 Put(*O*, ToString(*j*), *E*, **true**).
    c. ReturnIfAbrupt(*putStatus*).
    d. Increase *j* by 1.
12. Let *putStatus* be the result of Put(*O*, **`"length"`**, *len*+*argCount*, **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 an exotic object that is not an Array 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 Get(*O*, **"length"**)
4. Let *len* be ToUint32(*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 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 $k < 0$, then let *k* be 0.
12. Repeat, while $k<len$
    a. Let *kPresent* be the result of HasProperty(*O*, ToString(*k*)).
    b. ReturnIfAbrupt(*kPresent*).
    c. If *kPresent* is **true**, then
        i. Let *elementK* be the result of Get(*O*, ToString(*k*)).
        ii. ReturnIfAbrupt(*elementK*).
        iii. Let *same* be the result of performing the Strict Equality Comparison Algorithm *searchElement* === *elementK*.
        iv. If *same* is **true,** return *k*.
    d. 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 an exotic object that is not an Array 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.

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 Get(*O*, **"length"**)
4. Let *len* be ToUint32(*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 HasProperty(*O*, ToString(*k*)).
    b. ReturnIfAbrupt(*kPresent*).
    c. If *kPresent* is **true**, then

i. Let *elementK* be the result of Get(*O*, ToString(*k*)).
ii. ReturnIfAbrupt(*elementK*).
iii. Let *same* be the result of performing Strict Equality Comparison Algorithm *searchElement* === *elementK*.
iv. If *same* is **true,** return *k*.
d. 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 an exotic object that is not an Array 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 Get(*O*, **`"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 HasProperty(*O*, *Pk*).
   c. If *kPresent* is **true**, then
      i. Let *kValue* be the result of Get(*O*, *Pk*).
      ii. ReturnIfAbrupt(*kValue*).
      iii. Let *testResult* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as *thisArgument* and a List containing *kValue*, *k*, and *O* as *argumentsList*.
      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**.

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 an exotic object that is not an Array 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 Get(*O*, **"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 HasProperty(*O*, *Pk*).
    c. ReturnIfAbrupt(*kPresent*).
    d. If *kPresent* is **true**, then
        i. Let *kValue* be the result of Get(*O*, *Pk*).
        ii. ReturnIfAbrupt(*kValue*).
        iii. Let *testResult* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as *thisArgument* and a List containing *kValue*, *k*, and *O* as *argumentsList*.
        iv. ReturnIfAbrupt(*testResult*).
        v. If ToBoolean(*testResult)* is **true**, return **true**.
    e. Increase *k* by 1.
10. Return **false**.

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 an exotic object that is not an Array is implementation-dependent.

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:

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 Get(*O*, **`"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 HasProperty(*O*, *Pk*).
    c. ReturnIfAbrupt(*kPresent*).
    d. If *kPresent* is **true**, then
        i. Let *kValue* be the result of Get(*O*, *Pk*).
        ii. ReturnIfAbrupt(*kValue*).
        iii. Let *funcResult* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as *thisArgument* and a List containing *kValue*, *k*, and *O* as *argumentsList*.
        iv. ReturnIfAbrupt(*funcResult*).
    e. Increase *k* by 1.
10. 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 an exotic object that is not an Array 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.

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 Get(*O*, **"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 abstract operation ArrayCreate 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 HasProperty(*O*, *Pk*).
    c. ReturnIfAbrupt(*kPresent*).
    d. If *kPresent* is **true**, then
        i. Let *kValue* be the result of Get(*O*, *Pk*).
        ii. ReturnIfAbrupt(*kValue*).
        iii. Let *mappedValue* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as *thisArgument* and a List containing *kValue*, *k*, and *O* as *argumentsList*.
        iv. ReturnIfAbrupt(*mappedValue*).
        v. Call the [[DefineOwnProperty]] internal method of *A* with arguments *Pk* and Property Descriptor {[[Value]]: *mappedValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
    e. 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 an exotic object that is not an Array 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

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 Get(*O*, **`"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 abstract operation ArrayCreate 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 HasProperty(*O*, *Pk*).
    c. ReturnIfAbrupt(*kPresent*).
    d. If *kPresent* is **true**, then
        i. Let *kValue* be the result of Get(*O*, *Pk*).
        ii. ReturnIfAbrupt(*kValue*).
        iii. Let *selected* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as *thisArgument* and a List containing *kValue*, *k*, and *O* as *argumentsList*.
        iv. ReturnIfAbrupt(*selected*).
        v. If ToBoolean(*selected*) is **true**, then
            1. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*to*) and Property Descriptor {[[Value]]: *kValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
            2. Increase *to* by 1.
    e. 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 an exotic object that is not an Array 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.

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 Get(*O*, **`"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 HasProperty(*O*, *Pk*).
        iii. ReturnIfAbrupt(*kPresent*).
        iv. If *kPresent* is **true**, then
            1. Let *accumulator* be the result of Get(*O*, *Pk*).
            2. ReturnIfAbrupt(*accumulator*).
        v. 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 HasProperty(*O*, *Pk*).
    c. ReturnIfAbrupt(*kPresent*).
    d. If *kPresent* is **true**, then
        i. Let *kValue* be the result of Get(*O*, *Pk*).
        ii. ReturnIfAbrupt(*kValue*).
        iii. Let *accumulator* be the result of calling the [[Call]] internal method of *callbackfn* with **undefined** as *thisArgument* and a List containing *accumulator*, *kValue*, *k*, and *O* as *argumentsList*.
        iv. ReturnIfAbrupt(*accumulator*).
    e. 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 an exotic object that is not an Array 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 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 Get(*O*, **`"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 *Pk* be ToString(*k*).
        ii. Let *kPresent* be the result of HasProperty(*O*, *Pk*).
        iii. ReturnIfAbrupt(*kPresent*).
        iv. If *kPresent* is **true**, then
            1. Let *accumulator* be the result Get(*O*, *Pk*).
            2. ReturnIfAbrupt(*accumulator*).
        v. Decrease *k* by 1.
    c. If *kPresent* is **false**, throw a **TypeError** exception.
11. Repeat, while $k \geq 0$
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of HasProperty(*O*, *Pk*).
    c. ReturnIfAbrupt(*kPresent*).
    d. If *kPresent* is **true**, then
        i. Let *kValue* be the result of Get(*O*, *Pk*).
        ii. ReturnIfAbrupt(*kValue*).
        iii. Let *accumulator* be the result of calling the [[Call]] internal method of *callbackfn* with **undefined** as *thisArgument* and a List containing *accumulator*, *kValue*, *k*, and *O* as *argumentsList*.
        iv. ReturnIfAbrupt(*accumulator*).
    e. 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 an exotic object that is not an Array is implementation-dependent.

15.4.4.23 Array.prototype.items ( )

The following steps are taken:

1. Let *O* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*O*).
3. Return the result of calling the CreateArrayIterator abstract operation with arguments *O* and **`"key+value"`**.

15.4.4.24 Array.prototype.keys ( )

The following steps are taken:

1. Let *O* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*O*).
3. Return the result of calling the CreateArrayIterator abstract operation with arguments *O* and "**`key`**".

15.4.4.25 Array.prototype.values ( )

The following steps are taken:

1. Let *O* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*O*).
3. Return the result of calling the CreateArrayIterator abstract operation with arguments *O* and "**`value`**".

15.4.4.26 Array.prototype.@@iterator ( )

The initial value of the @@iterator property is the same function object as the initial value of the **Array.prototype.items** property.

15.4.5 Properties of Array Instances

Array instances are exot Array object that inherit properties from the Array prototype object and have the [[BuiltinBrand]] internal data property with value BuiltinArray. Array instances also have the following properties.

15.4.5.1 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.

15.4.6 Array Iterator Object Structure

An Array Iterator is an object, with the structure defined below, that represent a specific iteration over some specific Array instance object. There is not a named constructor for Array Iterator objects. Instead, Array iterator objects are created by calling certain methods of Array instance objects.

15.4.6.1 CreateArrayIterator Abstract Operation

Several methods of Array objects return interator objects. The abstract operation CreateArrayIterator with arguments *array* and *kind* is used to create and such iterator objects. It performs the following steps:

1. Let *O* be the result of calling ToObject(*array*).
2. ReturnIfAbrupt(*O*).
3. Let *itr* be the result of the abstract operation ObjectCreate with the intrinsic object %ArrayIteratorPrototype% as its argument.
4. Add a [[IteratedObject]] internal data property to *itr* with value *O*.
5. Add a [[ArrayIteratorNextIndex]] internaldata property to *itr* with value 0.
6. Add a [[ArrayIterationKind]] internal data property of *itr* with value *kind*.
7. Return *itr*.

15.4.6.2 The Array Iterator Prototype

All Array Iterator Objects inherit properties from a common Array Iterator Prototype object. The [[Prototype]] internal data property of the Array Iterator Prototype is the %ObjectPrototype% intrinsic object. In addition, the Array Iterator Prototype as the following properties:

15.4.6.2.1 *ArrayIterator*.prototype.constructor

15.4.6.2.2 *ArrayIterator*.prototype.next( )

1. Let *O* be the **this** value.
2. If Type(*O*) is not Object, throw a **TypeError** exception.
3. If *O* does not have all of the internal properties of a Array Iterator Instance (15.4.6.1.2), throw a **TypeError** exception.
4. Let *a* be the value of the [[IteratedObject]] internaldata property of *O*.
5. Let *index* be the value of the [[ArrayIteratorNextIndex]] internal data property of *O*.
6. Let *itemKind* be the value of the [[ArrayIterationKind]] internal data property of *O*.
7. Let *lenValue* be the result of Get(*a*, **"length"**).
8. Let *len* be ToUint32(*lenValue*).
9. ReturnIfAbrupt(*len*).
10. If *itemKind* contains the substring **"sparse"**, then
    a. Let *found* be **false**.
    b. Repeat, while *found* is **false** and *index* < *len*
        i. Let *elementKey* be ToString(*index*).
        ii. Let *found* be the result of HasProperty(*a*, *elementKey*).
        iii. ReturnIfAbrupt(*found*).
        iv. If *found* is **false**, then
            1. Increase *index* by 1.
11. If *index* ≥ *len*, then
    a. Set the value of the [[ArrayIteratorNextIndex]] internal data property of *O* to +Infinity.
    b. Return Completion {[[type]]: throw, [[value]]: %StopIteration%, [[target]]: empty}.
12. Let *elementKey* be ToString(*index*).
13. Set the value of the [[ArrayIteratorNextIndex]] internal data property of *O* to *index*+1.
14. If *itemKind* contains the substring **"value"**, then
    a. Let *elementValue* be the result of Get(*a*, *elementKey*).
    b. ReturnIfAbrupt(*elementValue*).
15. If *itemKind* contains the substring **"key+value"**, then
    a. Let *result* be the result of the abstract operation ArrayCreate with argument 2.
    b. Assert: *result* is a new, well-formed Array object so the following operations will never fail.
    c. Call the [[DefineOwnProperty]] internal method of *result* with arguments **"0"** and Property Descriptor {[[Value]]: *elementKey*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
    d. Call the [[DefineOwnProperty]] internal method of *result* with arguments **"1"** and Property Descriptor {[[Value]]: *elementValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
    e. Return *result*.
16. Else If *itemKind* contains the substring **"key"** then, return *elementKey*,.
17. Else *itemKind* is **"value"**,
    a. Return *elementValue*.

15.4.6.2.3 *ArrayIterator*.prototype.@@iterator ( )

The following steps are taken:

1. Return the **this** value.

15.4.6.2.4 *ArrayIterator*.prototype.@@toStringTag

The initial value of the @@toStringTag property is the string value **"Array Iterator"**.

15.4.6.3 Properties of Array Iterator Instances

Array Iterator instances inherit properties from the Array Iterator prototype (the intrinsic, %ArrayIteratorPrototype%.) Array Iterator instances are initially created with the internal properties listed in Table 31.

Table 31 — Internal Data Properties of Array Iterator Instances

Internal Data Property Name	Description
[[IteratedObject]]	The object whose array elements are being iterated.
[[ArrayIteratorNextIndex]]	The integer index of the next array index to be examined by this iteration.
[[ArrayIterationKind]]	A string value that identifies what is to be returned for each element of the iteration. The possible values are: **"key"**, **"value"**, **"key+value"**, **"sparse:key"**, **"sparse:value"**, **"sparse:key+value"**.

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 exotic String object.

15.5.2.1 new String ([value])

The [[Prototype]] internal data 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 [[Extensible]] internal data property of the newly constructed object is set to **true**.

The [[StringData]] internal data property of the newly constructed object is set to ToString(*value*), or to the empty String if *value* is not supplied.

15.5.3 Properties of the String Constructor

The value of the [[Prototype]] internal data property of the String constructor is the standard built-in Function prototype object (15.3.4).

Besides 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 (…codeUnits)

The **`String.fromCharCode`** function may be called with a variable number of arguments which form the rest parameter *codeUnits*. The following steps are taken:

1. Assert: *codeUnits* is a well-formed rest parameter object.
2. Let *length* be the result of Get(*codeUnits*, **`"length"`**).
3. Let *elements* be a new List.
4. Let *nextIndex* be 0.
5. Repeat while *nextIndex* < *length*
    a. Let *next* be the result of Get(*codeUnits*, ToString(*nextIndex*)).
    b. Let *nextCU* be ToUint16(*next*).
    c. ReturnIfAbrupt(*nextCU*).
    d. Append *nextCU* to the end of *elements*.
    e. Let *nextIndex* be *nextIndex* + 1.
6. Return the string value whose elements are, in order, the elements in the List *elements*. If *length* is 0, the empty string is returned.

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

15.5.3.3 String.fromCodePoint (…codePoints)

The **`String.fromCodePoint`** function may be called with a variable number of arguments which form the rest parameter *codePoints*. The following steps are taken:

1. Assert: *codePoints* is a well-formed rest parameter object.
2. Let *length* be the result of Get(*codePoints*, **`"length"`**).
3. Let *elements* be a new List.
4. Let *nextIndex* be 0.
5. Repeat while *nextIndex* < *length*
    a. Let *next* be the result of Get(*codePoints*, ToString(*nextIndex*)).
    b. Let *nextCP* be ToNumber(*next*).
    c. ReturnIfAbrupt(*nextCP*).
    d. If SameValue(*nextCP*, ToInteger(*nextCP*)) is **false**,then throw a **RangeError** exception.
    e. If *nextCP* < 0 or *nextCP* > 0x10FFFF, then throw a **RangeError** exception.
    f. Append the elements of the UTF-16 Encoding (clause 6) of *nextCP* to the end of *elements*.
    g. Let *nextIndex* be *nextIndex* + 1.
6. Return the string value whose elements are, in order, the elements in the List *elements*. If *length* is 0, the empty string is returned.

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

15.5.3.4 String.raw (callSite, …substitutions)

The **`String.raw`** function may be called with a variable number of arguments. The first argument is *callSite* and the remainder of the arguments form the rest parameter *substitutions*. The following steps are taken:

1. Assert: *substitutions* is a well-formed rest parameter object.
2. Let *cooked* be ToObject(*callSite*).
3. ReturnIfAbrupt(*cooked*).
4. Let *rawValue* be the result of Get(*cooked*, **`"raw"`**).
5. Let *raw* be ToObject(*rawValue*).
6. ReturnIfAbrupt(*raw*).

7. Let *len* be the result of Get(*raw*, **"length"**).
8. Let *literalSegments* be ToUint(*len*).
9. ReturnIfAbrupt(*literalSegments*).
10. If *literalSegments* = 0, then return the empty string.
11. Let *stringElements* be a new List.
12. Let *nextIndex* be 0.
13. Repeat while *nextIndex* < *literalSegments*
    a. Let *nextKey* be ToString(*nextIndex*).
    b. Let *next* be the result of Get(*raw*, *nextKey*).
    c. Let *nextSeg* be ToString(*next*).
    d. ReturnIfAbrupt(*nextSeg*).
    e. Append in order the code unit elements of *nextSeg* to the end of *stringElements*.
    f. If *nextIndex* + 1 = *literalSegments*, then
        i. Return the string value whose elements are, in order, the elements in the List *stringElements*. If *length* is 0, the empty string is returned.
    g. Let *next* be the result of Get(*substitutions*, *nextKey*).
    h. Let *nextSub* be ToString(*next*).
    i. ReturnIfAbrupt(*nextSub*).
    j. Append in order the code unit elements of *nextSub* to the end of *stringElements*.
    k. Let *nextIndex* be *nextIndex* + 1.

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

NOTE String.raw is intended for use as a tag function of a Tagged Template String (11.2.6). When called as such the first argument will be a well formed template call site object and the rest parameter will contain the substitution values.

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 [[BuiltinBrand]] internal data property with value BuiltinStringWrapper.

The value of the [[Prototype]] internal data 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)

NOTE Returns a single element String containing the code unit at element position *pos* in the String value resulting from converting this object to a String. If there is no element 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 elements in *S*.
7. If *position* < 0 or *position* ≥ *size*, return the empty String.
8. Return a String of length 1, containing one code unit from *S*, namely the code unit at position *position*, where the first (leftmost) code unit in *S* is considered to be at position 0, the next one at position 1, and so on.

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)

NOTE Returns a Number (a nonnegative integer less than 2^{16}) that is the code unit value of the string element at position *pos* in the String resulting from converting this object to a String. If there is no element 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 elements 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 element at position *position* in the String *S*, where the first (leftmost) element 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 (…args)

NOTE When the **`concat`** method is called with zero or more arguments, it returns a String consisting of the string elements of this object (converted to a String) followed by the string elements of each of the arguments converted to a String. The result is a String value, not a String object.

The following steps are taken:

1. Assert: *args* is a well-formed rest parameter object.
2. ReturnIfAbrupt(CheckObjectCoercible(**this** value)).
3. Let *S* be the result of calling ToString, giving it the **this** value as its argument.
4. ReturnIfAbrupt(*S*).
5. Let *args* be an internal list that is a copy of the argument list passed to this function.
6. Let *R* be *S*.
7. 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 string elements in the previous value of *R* followed by the string elements of *nextString*.

8. 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*).
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 elements in *S*.
9. Let *start* be min(max(*pos*, 0), *len*).
10. Let *searchLen* be the number of elements 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 code unit at position *k*+*j* of *S* is the same as the code unit 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 elements in *S*.
10. Let *start* be min(max(*pos*, 0), *len*).
11. Let *searchLen* be the number of elements 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 code unit at position *k*+*j* of *S* is the same as the code unit at position *j* of *searchStr*; but if there is no such integer *k*, then return the value **`-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 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*).
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 [[BuiltinBrand]] internal data property whose value is BuiltinRegExp, then let *rx* be *regexp*;
5. Else, let *rx* be the result of the abstract operation RegExpCreate (15.10.4.1) with arguments *regexp* and **undefined**.
6. ReturnIfAbrupt(*rx*).

7. Let *global* be the result of Get(*rx*, **"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 Put(*rx*, **"lastIndex"**, 0, **true**).
    b. ReturnIfAbrupt(*putStatus*).
    c. Let *A* be the result of the abstract operation ArrayCreate 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**,
            1. Let *thisIndex* be the result of Get(*rx*, **"lastIndex"**).
            2. ReturnIfAbrupt(*thisIndex*).
            3. If *thisIndex* = *previousLastIndex* then
                a Let *putStatus* be the result of Put(*rx*, **"lastIndex"**, *thisIndex*+1, **true**).
                b ReturnIfAbrupt(*putStatus*).
                c Set *previousLastIndex* to *thisIndex*+1.
            4. Else,
                a Set *previousLastIndex* to *thisIndex*.
            5. Let *matchStr* be the result of Get(*result*, **"0"**).
            6. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*n*) and the Property Descriptor {[[Value]]: *matchStr*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[configurable]]: **true**}.
            7. ReturnIfAbrupt(*defineStatus*).
            8. 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 [[BuiltinBrand]] internal data property whose value is BuiltinRegExp), 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 elements in *newstring* by replacement text as specified in Table 32. 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 32 — Replacement Text Symbol Substitutions

Code unit	Unicode Characters	Replacement text
0x0024, 0x0024	**$$**	**$**
0x0024, 0x0026	**$&**	The matched substring.
0x0024, 0x0060	**$`**	The portion of *string* that precedes the matched substring.
0x0024, 0x0027	**$'**	The portion of *string* that follows the matched substring.
0x0024, N where 0x0030 ≤ N ≤ 0x0039	**$n** where **n** is one of **0 1 2 3 4 5 6 7 8 9**	The n^{th} capture, where *n* is a single digit in the range 1 to 9 and **$***n* is not followed by a decimal digit. If *n*≤*m* and the *n*th capture is **undefined**, use the empty String instead. If *n*>*m*, the result is implementation-defined.
0x0024, N, N where 0x0030 ≤ N ≤ 0x0039	**$nn** where **n** is one of **0 1 2 3 4 5 6 7 8 9**	The nn^{th} capture, where *nn* is a two-digit decimal number in the range 01 to 99. If *nn*≤*m* and the nn^{th} capture is **undefined**, use the empty String instead. If *nn*>*m*, 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(*string*).
4. If Type(*regexp*) is Object and *regexp* has a [[BuiltinBrand]] internal data property whose value is BuiltinRegExp , then,
   a. Let *rx* be *regexp*;
5. Else,
   a. Let *rx* be the result of the abstract 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 element position *start* and running to, but not including, element 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 elements 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 elements from *S* beginning with the element 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.

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 [[BuiltinBrand]] internal data property whose value is BuiltinRegExp ; 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 code unit elements; the length of the result array equals the length of the String, and each substring contains one code unit.) 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 **`["","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(/<(\/)?([^<>]+)>/)
```

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* = ToUint32(*limit*).
7. Let *s* be the number of elements in *S*.
8. Let *p* = 0.
9. If *separator* has a [[BuiltinBrand]] internal data property whose value is BuiltinRegExp , 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"` and Property Descriptor {[[Value]]: *S*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
    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"` and Property Descriptor {[[Value]]: *S*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
    d. Assert: the previous step will never result in an abrupt completion.
    e. Return *A*.
14. Let *q* = *p*.
15. Repeat, while *q* ≠ *s*
    a. Call *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* ≠ *p*,
            1. Let *T* be a String value equal to the substring of *S* consisting of the elements at positions *p* (inclusive) through *q* (exclusive).
            2. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*lengthA*) and Property Descriptor {[[Value]]: *T*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
            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 ToString(*lengthA*) and Property Descriptor {[[Value]]: *cap*[*i*], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
                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 elements at positions *p* (inclusive) through *s* (exclusive).
17. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*lengthA*) and Property Descriptor {[[Value]]: *T*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
18. Assert: the previous step will never result in an abrupt completion.
19. Return *A*.

Runtime Semantics: SplitMatch Abstract Operation

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 [[BuiltinBrand]] internal data property whose value is BuiltinRegExp , 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 elements in *R*.
3. Let *s* be the number of elements 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 code unit at position *q*+*i* of *S* is different from the code unit 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)

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 element position *start* and running to, but not including, element position *end* of the String (or through the end of the String is *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 elements 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 code units from *S*, namely the code units with indices *from* through *to* –1, in ascending order.

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 ( )

This function interprets a string value as a sequence of code points, as described in 8.4. 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 *cpList* be a List containing in order the code points as defned in 8.4 of *S*, starting at the first element of *S*.
5. For each code point *c* in *cpList*, if the Unicode Character Database provides a language insensitive lower case equivalent of *c* then replace *c* in *cpList* with that equivalent code point(s).
6. Let *cuList* be a new List.
7. For each code point *c* in *cpList*, in order, append to *cuList* the elements of the UTF-16 Encoding (clause 6) of *c*.
8. Let *L* be a String whose elements are, in order, the elements of *cuList* .
9. Return *L*.

The result must be derived according to the case mappings in the Unicode character database (this explicitly includes not only the UnicodeData.txt file, but also the SpecialCasings.txt file that accompanies it).

NOTE 1 The case mapping of some code points may produce multiple code points . 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()`**.

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 interprets a string value as a sequence of code points, as described in 8.4.

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 interprets a string value as a sequence of code points, as described in 8.4.

This function behaves in exactly the same way as **`String.prototype.toLowerCase`**, except that code points 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 interprets a string value as a sequence of code points, as described in 8.4.

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 ( )

This function interprets a string value as a sequence of code points, as described in 8.4.

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*. When determining whether a Unicode character is in Unicode general category "Zs", code unit sequences are interpreted as UTF-16 encoded code point sequences as specified in 8.4.
5. Return *T*.

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. If *n* is +∞, then throw a **RangeError** exception.
8. Let *T* be a String value that is made from *n* copies of *S* appended together.
9. Return *T*.

NOTE 1 This method creates a String consisting of the string elements 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 elements in *S*.
9. Let *start* be min(max(*pos*, 0), *len*).
10. Let *searchLength* be the number of elements in *searchString*.
11. If *searchLength*+*start* is greater than *len*, return **false**.
12. If the *searchLength* sequence of elements of *S* starting at *start* is the same as the full element 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 elements of *searchString* converted to a String is the same as the corresponding elements 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 elements 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 elements in *searchString*.
11. Let *start* be *end* - *searchLength*.
12. If *start* is less than 0, return **false**.
13. If the *searchLength* sequence of elements of *S* starting at *start* is the same as the full element 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 elements of *searchString* converted to a String is the same as the corresponding elements 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 = 0)

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 elements 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.codePointAt (pos)

NOTE Returns a Number (a nonnegative integer less than 1114112) that is the UTF-16 encoded code point value starting at the string element at position *pos* in the String resulting from converting this object to a String. If there is no element at that position, the result is **NaN**. If a valid UTF-16 surrogate pair does not begin at *pos*, the result is the code unit at *pos*.

When the **`codePointAt`** 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 elements in *S*.
7. If *position* < 0 or *position* ≥ *size*, return **undefined**.
8. Let *first* be the code unit value of the element at index *position* in the String *S*.
9. If *first* < 0xD800 or *first* > 0xDBFF or *position*+1 = *size*, then return *first*.
10. Let *second* be the code unit value of the element at index *position*+1 in the String *S*.
11. If *second* < 0xDC00 or *second* > 0xDFFF, then return *first*.
12. Return ((*first* – 0xD800) × 1024) + (*second* – 0xDC00) + 0x10000.

NOTE The **`codePointAt`** 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 are String exotic objects and have the internal methods and internal data properties specified for such objects. String instance inherit properties from the String prototype object. String instances also have a **`length`** property, and a set of enumerable properties with array index names.

15.5.5.1 length

The number of elements 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** }.

15.6 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 ordinary object.

15.6.2.1 new Boolean (value)

The [[Prototype]] internal data 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 [[BuiltinBrand]] internal data property with value BuiltinBooleanWrapper.

The [[BooleanValue]] internal data property of the newly constructed Boolean object is set to ToBoolean(*value*).

The [[Extensible]] internal data property of the newly constructed object is set to **true**.

15.6.3 Properties of the Boolean Constructor

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

Besides 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 [[BooleanValue]] internal data property has the value **false**. The Boolean prototype object has a [[BuiltinBrand]] internal data property whose value is BuiltinBooleanWrapper.

The value of the [[Prototype]] internal data 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 [[BooleanData]] internal data property, then let *b* be the value of the [[BooleanData]] internal data property of *B*.
4. Else throw a **TypeError** exception.
5. If *b* is **true**, then return **`"true"`**; else return **`"false"`**.

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 [[BooleanData]] internal data property, then let *b* be the value of the [[BooleanData]] internal data property of *B*.
4. Else throw a **TypeError** exception.
5. Return *b*.

15.6.5 Properties of Boolean Instances

Boolean instances are ordinary objects that inherit properties from the Boolean prototype object. Boolean instance objects have a [[BuiltinBrand]] internal data property whose value is BuiltinBooleanWrapper. Boolean instances also have a [[BooleanData]] internal data property.

The [[BooleanData]] internal data property is the Boolean value represented by this Boolean object.

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 ordinary object.

15.7.2.1 new Number ([value])

The [[Prototype]] internal data 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 has a [[BuiltinBrand]] internal data property whose value is BuiltinNumberWrapper.

The [[NumberData]] internal data property of the newly constructed object is set to ToNumber(*value*) if *value* was supplied, else to +0.

The [[Extensible]] internal data property of the newly constructed object is set to **true**.

15.7.3 Properties of the Number Constructor

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

Besides 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 5×10^{-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 $2.220446049250313080847263336181 6 \times 10^{-16}$.

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

15.7.3.8 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** }.

15.7.3.9 Number.parseInt (string, radix)

Same as 15.1.2.2.

15.7.3.10 Number.parseFloat (string)

Same as 15.1.2.3.

15.7.3.11 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.

15.7.3.12 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**, +∞, or −∞, return **false**.
3. Otherwise, return **true**.

15.7.3.13 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**.

15.7.3.14 Number.toInt (number)

When the **`Number.toInt`** is called with one argument *number*, the following steps are taken:

1. Return ToInteger(*number*).

15.7.4 Properties of the Number Prototype Object

The Number prototype object is itself a Number object with a [[BuiltinBrand]] internal data property whose value is BuiltinNumberWrapper. Its value is +0.

The value of the [[Prototype]] internal data property of the Number prototype object is the standard built-in Object prototype object (15.2.4).

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 [[NumberData]] internal data property.

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 [[NumberData]] internal data 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 [[NumberData]] internal data property or a value whose type is Number.

15.7.4.1 Number.prototype.constructor

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

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.

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* = 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 elements in *m*.
        ii. If $k \leq f$, then
            1. Let *z* be the String consisting of *f*+1–*k* occurrences of the code unit 0x0030.
            2. Let *m* be the concatenation of Strings *z* and *m*.
            3. Let *k* = *f* + 1.
        iii. Let *a* be the first *k*–*f* elements of *m*, and let *b* be the remaining *f* elements 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 ToInteger(*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 *fractionDigits* is **undefined**, then let $f = 0$.
    b. Let *m* be the String consisting of $f+1$ occurrences of the code unit 0x0030.
    c. Let $e = 0$.
11. Else $x \neq 0$,
    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 $f \geq 0$, $10^f \leq n < 10^{f+1}$, the number value for $n \times 10^{e-f}$ is *x*, and *f* is as small as possible. Note that the decimal representation of *n* has $f+1$ digits, *n* is not divisible by 10, and the least significant digit of *n* is not necessarily uniquely determined by these criteria.
    c. Let *m* be the String consisting of the digits of the decimal representation of *n* (in order, with no leading zeroes).
12. If $f \neq 0$, then
    a. Let *a* be the first element of *m*, and let *b* be the remaining *f* elements 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 $e \leq 0$,
        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*.

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 code unit 0x0030 (the Unicode 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 element of *m*, and let *b* be the remaining *p*–1 elements 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* ≥ 0, then
a. Let *m* be the concatenation of the first *e*+1 elements of *m*, the code unit 0x002E (Unicode character '.'), and the remaining *p*– (*e*+1) elements of *m*.
15. Else *e* < 0,
a. Let *m* be the concatenation of the String **"0."**, –(*e*+1) occurrences of code unit 0x0030 (the Unicode 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.

15.7.4.8 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 of the 32-bit binary encoding of *n* is 1, *p* will be 0.

15.7.5 Properties of Number Instances

Number instances are ordinary objects that inherit properties from the Number prototype object and have a [[BuiltinBrand]] internal data property whose value is BuiltinNumberWrapper. Number instances also have a [[NumberValue]] internal data property.

The [[NumberValue]] internal data property is the Number value represented by this Number object.

15.8 The Math Object

The Math object is a single ordinary object that has some named properties, some of which are functions.

The value of the [[Prototype]] internal data property of the Math object is the standard built-in Object prototype object (15.2.4). The Math object has a [[BuiltinBrand]] internal data property whose value is BuiltinMath.

The Math is not a function object. It does not have a [[Construct]] internal method; 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 method; 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 Math.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** }.

15.8.1.2 Math.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 Math.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 Math.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 Math.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 Math.PI

The Number value for π, 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 Math.SQRT1_2

The Number value for the square root of ½, 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 Math.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 operation 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, functction performs a computation on the resulting Number value(s).

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 Math.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 Math.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 Math.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 Math.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 Math.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 –π to +π.

- If either *x* or *y* is NaN, the result is NaN.
- If *y*>0 and *x* is +0, the result is an implementation-dependent approximation to +π/2.
- If *y*>0 and *x* is –0, the result is an implementation-dependent approximation to +π/2.
- If *y* is +0 and *x*>0, the result is +0.
- If *y* is +0 and *x* is +0, the result is +0.
- If *y* is +0 and *x* is –0, the result is an implementation-dependent approximation to +π.
- If *y* is +0 and *x*<0, the result is an implementation-dependent approximation to +π.
- If *y* is –0 and *x*>0, the result is –0.
- If *y* is –0 and *x* is +0, the result is –0.
- If *y* is –0 and *x* is –0, the result is an implementation-dependent approximation to –π.
- If *y* is –0 and *x*<0, the result is an implementation-dependent approximation to –π.
- If *y*<0 and *x* is +0, the result is an implementation-dependent approximation to –π/2.
- If *y*<0 and *x* is –0, the result is an implementation-dependent approximation to –π/2.
- If *y*>0 and *y* is finite and *x* is +∞, the result is +0.
- If *y*>0 and *y* is finite and *x* is –∞, the result if an implementation-dependent approximation to +π.
- If *y*<0 and *y* is finite and *x* is +∞, the result is –0.
- If *y*<0 and *y* is finite and *x* is –∞, the result is an implementation-dependent approximation to –π.
- If *y* is +∞ and *x* is finite, the result is an implementation-dependent approximation to +π/2.
- If *y* is –∞ and *x* is finite, the result is an implementation-dependent approximation to –π/2.
- If *y* is +∞ and *x* is +∞, the result is an implementation-dependent approximation to +π/4.
- If *y* is +∞ and *x* is –∞, the result is an implementation-dependent approximation to +3π/4.
- If *y* is –∞ and *x* is +∞, the result is an implementation-dependent approximation to –π/4.
- If *y* is –∞ and *x* is –∞, the result is an implementation-dependent approximation to –3π/4.

15.8.2.6 Math.ceil (x)

Returns the smallest (closest to –∞) 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 +∞, the result is +∞.
- If *x* is –∞, the result is –∞.
- 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 Math.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 +∞, the result is NaN.
- If *x* is –∞, the result is NaN.

15.8.2.8 Math.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 +∞, the result is +∞.
- If *x* is –∞, the result is +0.

15.8.2.9 Math.floor (x)

Returns the greatest (closest to **+∞**) 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 +∞, the result is +∞.
- If *x* is –∞, the result is –∞.
- 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 Math.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 –∞.
- If *x* is 1, the result is +0.
- If *x* is +∞, the result is +∞.

15.8.2.11 Math.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 –∞.
- 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.

The **`length`** property of the **`max`** method is **2**.

15.8.2.12 Math.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 +∞.
- 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.

The **`length`** property of the **`min`** method is **2**.

15.8.2.13 Math.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 NaN.
- If *y* is –0, the result is 1, even if *x* is NaN.
- If *x* is NaN and *y* is nonzero, the result is NaN.
- If abs(*x*)>1 and *y* is +∞, the result is +∞.
- If abs(*x*)>1 and *y* is –∞, the result is +0.
- If abs(*x*) is 1 and *y* is +∞, the result is NaN.
- If abs(*x*) is 1 and *y* is –∞, the result is NaN.
- If abs(*x*)<1 and *y* is +∞, the result is +0.
- If abs(*x*)<1 and *y* is –∞, the result is +∞.
- If *x* is +∞ and *y*>0, the result is +∞.
- If *x* is +∞ and *y*<0, the result is +0.
- If *x* is –∞ and *y*>0 and *y* is an odd integer, the result is –∞.
- If *x* is –∞ and *y*>0 and *y* is not an odd integer, the result is +∞.
- If *x* is –∞ and *y*<0 and *y* is an odd integer, the result is –0.
- If *x* is –∞ 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 +∞.
- 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 –∞.
- If *x* is –0 and *y*<0 and *y* is not an odd integer, the result is +∞.
- If *x*<0 and *x* is finite and *y* is finite and *y* is not an integer, the result is NaN.

15.8.2.14 Math.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 Math.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 +∞. 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 +∞, the result is +∞.
- If *x* is –∞, the result is –∞.
- 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 Math.sin (x)

Returns an implementation-dependent approximation to the sine 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 +∞ or –∞, the result is NaN.

15.8.2.17 Math.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 +∞, the result is +∞.

15.8.2.18 Math.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 +∞ or –∞, the result is NaN.

15.8.2.19 Math.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 –∞.
- If *x* is –0, the result is –∞.
- If *x* is 1, the result is +0.
- If *x* is +∞, the result is +∞.

15.8.2.20 Math.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 –∞.
- If *x* is –0, the result is –∞.
- If *x* is 1, the result is +0.
- If *x* is +∞, the result is +∞.

15.8.2.21 Math.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 -∞.

- If *x* is +0, the result is +0.
- If *x* is –0, the result is –0.
- If *x* is +∞, the result is +∞.

15.8.2.22 Math.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 +∞, the result is +∞.
- If *x* is –∞, the result is -1.

15.8.2.23 Math.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 +∞, the result is +∞.
- If *x* is –∞, the result is +∞.

NOTE The value of cosh(x) is the same as *(exp(x) + exp(-x))/2*.

15.8.2.24 Math.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 +∞, the result is +∞.
- If x is –∞, the result is –∞.

NOTE The value of cosh(x) is the same as *(exp(x) - exp(-x))/2*.

15.8.2.25 Math.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 +∞, the result is +1.
- If *x* is –∞, 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 Math.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 +∞, the result is +∞.

15.8.2.27 Math.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 +∞, the result is +∞.
- If x is −∞, the result is −∞.

15.8.2.28 Math.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 −∞.
- If *x* is +1, the result is +∞.
- If *x* is +0, the result is +0.
- If *x* is −0, the result is −0.

15.8.2.29 Math.hypot(value1 , value2, value3 = 0)

Given two or three arguments, hypot returns an implementation-dependent approximation of the square root of the sum of squares of up to three arguments.

- If any argument is +∞, the result is +∞.
- If any argument is −∞, the result is +∞.
- If no argument is +∞ or −∞, 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 Math.hypot2(value1 , value2 [, value3])~~

~~Given two or three arguments, hypot2 returns an implementation-dependent approximation of the sum of squares of its arguments.~~

- ~~If no arguments are given, the result is +0.~~
- ~~If any argument is +∞, the result is +∞.~~
- ~~If any argument is −∞, the result is +∞.~~
- ~~If no argument is +∞ or −∞, 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 Math.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 +∞, the result is +∞.

- If x is −∞, the result is −∞.

15.8.2.31 Math.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 and not −0, the result is −1.
- If *x* is positive and not +0, the result is +1.

15.8.2.32 Math.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 Operations

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

$$\mathrm{Day}(t) = \mathrm{floor}(t / \mathrm{msPerDay})$$

where the number of milliseconds per day is

$$\mathrm{msPerDay} = 86400000$$

The remainder is called the time within the day:

$$\mathrm{TimeWithinDay}(t) = t \bmod \mathrm{msPerDay}$$

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

$$\begin{aligned}\text{DaysInYear}(y) &= 365 \text{ if } (y \text{ modulo } 4) \neq 0\\ &= 366 \text{ if } (y \text{ modulo } 4) = 0 \text{ and } (y \text{ modulo } 100) \neq 0\\ &= 365 \text{ if } (y \text{ modulo } 100) = 0 \text{ and } (y \text{ modulo } 400) \neq 0\\ &= 366 \text{ if } (y \text{ modulo } 400) = 0\end{aligned}$$

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:

$$\text{YearFromTime}(t) = \text{the largest integer } y \text{ (closest to positive infinity) such that TimeFromYear}(y) \leq t$$

The leap-year function is 1 for a time within a leap year and otherwise is zero:

$$\begin{aligned}\text{InLeapYear}(t) &= 0 \text{ if DaysInYear(YearFromTime}(t)) = 365\\ &= 1 \text{ if DaysInYear(YearFromTime}(t)) = 366\end{aligned}$$

15.9.1.4 Month Number

Months are identified by an integer in the range 0 to 11, inclusive. The mapping MonthFromTime(*t*) from a time value *t* to a month number is defined by:

$$\begin{aligned}\text{MonthFromTime}(t) &= 0 \text{ if } 0 &&\leq \text{DayWithinYear}(t) < 31\\ &= 1 \text{ if } 31 &&\leq \text{DayWithinYear}(t) < 59+\text{InLeapYear}(t)\\ &= 2 \text{ if } 59+\text{InLeapYear}(t) &&\leq \text{DayWithinYear}(t) < 90+\text{InLeapYear}(t)\\ &= 3 \text{ if } 90+\text{InLeapYear}(t) &&\leq \text{DayWithinYear}(t) < 120+\text{InLeapYear}(t)\\ &= 4 \text{ if } 120+\text{InLeapYear}(t) &&\leq \text{DayWithinYear}(t) < 151+\text{InLeapYear}(t)\\ &= 5 \text{ if } 151+\text{InLeapYear}(t) &&\leq \text{DayWithinYear}(t) < 181+\text{InLeapYear}(t)\\ &= 6 \text{ if } 181+\text{InLeapYear}(t) &&\leq \text{DayWithinYear}(t) < 212+\text{InLeapYear}(t)\\ &= 7 \text{ if } 212+\text{InLeapYear}(t) &&\leq \text{DayWithinYear}(t) < 243+\text{InLeapYear}(t)\\ &= 8 \text{ if } 243+\text{InLeapYear}(t) &&\leq \text{DayWithinYear}(t) < 273+\text{InLeapYear}(t)\\ &= 9 \text{ if } 273+\text{InLeapYear}(t) &&\leq \text{DayWithinYear}(t) < 304+\text{InLeapYear}(t)\\ &= 10 \text{ if } 304+\text{InLeapYear}(t) &&\leq \text{DayWithinYear}(t) < 334+\text{InLeapYear}(t)\\ &= 11 \text{ if } 334+\text{InLeapYear}(t) &&\leq \text{DayWithinYear}(t) < 365+\text{InLeapYear}(t)\end{aligned}$$

where

$$\text{DayWithinYear}(t) = \text{Day}(t) - \text{DayFromYear(YearFromTime}(t))$$

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 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 DateFromTime(*t*) from a time value *t* to a month number is defined by:

$$\begin{aligned}\text{DateFromTime}(t) &= \text{DayWithinYear}(t)+1 &&\text{if MonthFromTime}(t)=0\\ &= \text{DayWithinYear}(t)-30 &&\text{if MonthFromTime}(t)=1\\ &= \text{DayWithinYear}(t)-58-\text{InLeapYear}(t) &&\text{if MonthFromTime}(t)=2\end{aligned}$$

= 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

WeekDay(*t*) = (Day(*t*) + 4) 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 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 LocalTZA measured in milliseconds which when added to UTC represents the local *standard* time. Daylight saving time is *not* reflected by LocalTZA.

NOTE It is recommended that implementations use the time zone information of the IANA Time Zone Database.

15.9.1.8 Daylight Saving Time Adjustment

An implementation of ECMAScript is expected to make its best effort to determine the local daylight saving time adjustment. An implementation dependent algorithm using best available information on time zones to determine the local daylight saving time adjustment DaylightSavingTA(*t*), measured in milliseconds.

15.9.1.9 Local Time

Conversion from UTC to local time is defined by

LocalTime(*t*) = *t* + LocalTZA + DaylightSavingTA(*t*)

Conversion from local time to UTC is defined by

UTC(*t*) = *t* – LocalTZA – DaylightSavingTA(*t* – LocalTZA)

Note that UTC(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:

HourFromTime(*t*) = floor(*t* / msPerHour) modulo HoursPerDay
MinFromTime(*t*) = floor(*t* / msPerMinute) modulo MinutesPerHour
SecFromTime(*t*) = floor(*t* / msPerSecond) modulo SecondsPerMinute
msFromTime(*t*) = *t* modulo msPerSecond

where

HoursPerDay = 24
MinutesPerHour = 60
SecondsPerMinute = 60
msPerSecond = 1000

msPerMinute = 60000 = msPerSecond × SecondsPerMinute

msPerHour = 3600000 = msPerMinute × MinutesPerHour

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* ***** msPerHour **+** *m* ***** msPerMinute **+** *s* ***** 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* + floor(*m* /12).
6. Let *mn* be *m* modulo 12.
7. Find a value *t* such that YearFromTime(*t*) is *ym* and MonthFromTime(*t*) is *mn* and DateFromTime(*t*) is 1; but if this is not possible (because some argument is out of range), return **NaN**.
8. Return 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* × 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*) > 8.64×10^{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.

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 ordinary 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 data 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 [[BuiltinBrand]] internal data property whose value is BuiltinDate.

The [[Extensible]] internal data property of the newly constructed object is set to **true**.

The [[DateValue]] internal data 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 ≤ ToInteger(*y*) ≤ 99, then let *yr* be 1900+ToInteger(*y*); otherwise, let *yr* be *y*.
16. Let *finalDate* be MakeDate(MakeDay(*yr*, *m*, *dt*), MakeTime(*h*, *min*, *s*, *milli*)).
17. Set the [[DateValue]] internal data property of the newly constructed object to TimeClip(UTC(*finalDate*)).

15.9.3.2 new Date (value)

The [[Prototype]] internal data 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 [[BuiltinBrand]] internal data property whose value is BuiltinDate.

The [[Extensible]] internal data property of the newly constructed object is set to **true**.

The [[DateValue]] internal data property of the newly constructed object is set as follows:

1. Let *v* be ToPrimitive(*value*).
2. If Type(*v*) is String, then
    a. Let *V* be the result of parsing *v* as a date, in exactly the same manner as for the **`parse`** method (15.9.4.2). If the parse resulted in an abrupt completion, *V* is the Completion Record.
3. Else,
    a. Let *V* be ToNumber(*v*).
4. ReturnIfAbrupt(*V*).
5. Set the [[DateValue]] internal data property of the newly constructed object to TimeClip(*V*).
6. Return the newly constructed object.

15.9.3.3 new Date ( )

The [[Prototype]] internal data 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 [[BuiltinBrand]] internal data property whose value is BuiltinDate.

The [[Extensible]] internal data property of the newly constructed object is set to **true**.

The [[DateValue]] internal data 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 data property of the Date constructor is the Function prototype object (15.3.4).

Besides the **`length`** property (whose value is **`7`**), the Date constructor has the following properties:

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 ToString operator to its argument. If ToString 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**.

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 ≤ ToInteger(*y*) ≤ 99, then let *yr* be 1900+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 [[BuiltinBrand]] internal data property whose value is BuiltinDate. Its [[DateValue]] is **NaN**.

The value of the [[Prototype]] internal data 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 [[DateValue]] internal data property. 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 [[DateValue]] internal data 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.

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. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return YearFromTime(LocalTime(*t*)).

15.9.5.11 Date.prototype.getUTCFullYear ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return YearFromTime(*t*).

15.9.5.12 Date.prototype.getMonth ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return MonthFromTime(LocalTime(*t*)).

15.9.5.13 Date.prototype.getUTCMonth ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return MonthFromTime(*t*).

15.9.5.14 Date.prototype.getDate ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return DateFromTime(LocalTime(*t*)).

15.9.5.15 Date.prototype.getUTCDate ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return DateFromTime(*t*).

15.9.5.16 Date.prototype.getDay ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return WeekDay(LocalTime(*t*)).

15.9.5.17 Date.prototype.getUTCDay ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return WeekDay(*t*).

15.9.5.18 Date.prototype.getHours ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return HourFromTime(LocalTime(*t*)).

15.9.5.19 Date.prototype.getUTCHours ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return HourFromTime(*t*).

15.9.5.20 Date.prototype.getMinutes ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return MinFromTime(LocalTime(*t*)).

15.9.5.21 Date.prototype.getUTCMinutes ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return MinFromTime(*t*).

15.9.5.22 Date.prototype.getSeconds ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return SecFromTime(LocalTime(*t*)).

15.9.5.23 Date.prototype.getUTCSeconds ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return SecFromTime(*t*).

15.9.5.24 Date.prototype.getMilliseconds ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return msFromTime(LocalTime(*t*)).

15.9.5.25 Date.prototype.getUTCMilliseconds ( )

1. Let *t* be this time value.
2. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. 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. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return (*t* – LocalTime(*t*)) / msPerMinute.

15.9.5.27 Date.prototype.setTime (time)

1. Let *v* be TimeClip(ToNumber(*time*)).
2. ReturnIfAbrupt(*v*).
3. Set the [[DateValue]] internal data property of this Date object to *v*.
4. 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 [[DateValue]] internal data 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. ReturnIfAbrupt(*t*).
3. Let *time* be MakeTime(HourFromTime(*t*), MinFromTime(*t*), SecFromTime(*t*), ToNumber(*ms*)).
4. Let *v* be TimeClip(MakeDate(Day(*t*), *time*)).
5. Set the [[DateValue]] internal data property of this Date object to *v*.
6. 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 [[DateValue]] internal data 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. ReturnIfAbrupt(*t*).
3. 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*), MinFromTime(*t*), *s*, *milli*)).
6. Let *v* be TimeClip(*date*).
7. Set the [[DateValue]] internal data property of this Date object to *v*.
8. 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 [[DateValue]] internal data 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.setUTCMinutes (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. ReturnIfAbrupt(*t*).
3. 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(HourFromTime(*t*), *m*, *s*, *milli*)).
7. Let *v* be TimeClip(*date*).
8. Set the [[DateValue]] internal data property of this Date object to *v*.
9. Return *v*.

The **`length`** property of the **`setUTCMinutes`** 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 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 [[DateValue]] internal data 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 **`getUTCMinutes()`**.

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. ReturnIfAbrupt(*t*).
3. Let *h* be ToNumber(*hour*).
4. If *min* is not specified, then let *m* be MinFromTime(*t*); otherwise, let *m* be ToNumber(*min*).
5. If *sec* is not specified, then let *s* be SecFromTime(*t*); otherwise, let *s* be ToNumber(*sec*).
6. If *ms* is not specified, then let *milli* be msFromTime(*t*); otherwise, let *milli* be ToNumber(*ms*).
7. Let *newDate* be MakeDate(Day(*t*), MakeTime(*h*, *m*, *s*, *milli*)).
8. Let *v* be TimeClip(*newDate*).
9. Set the [[DateValue]] internal data property of this Date object to *v*.
10. 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 LocalTime(this time value).
2. Let *dt* be ToNumber(*date*).
3. Let *newDate* be MakeDate(MakeDay(YearFromTime(*t*), MonthFromTime(*t*), *dt*), TimeWithinDay(*t*)).
4. Let *u* be TimeClip(UTC(*newDate*)).
5. Set the [[DateValue]] internal data 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. ReturnIfAbrupt(*t*).
3. Let *dt* be ToNumber(*date*).
4. Let *newDate* be MakeDate(MakeDay(YearFromTime(*t*), MonthFromTime(*t*), *dt*), TimeWithinDay(*t*)).
5. Let *v* be TimeClip(*newDate*).
6. Set the [[DateValue]] internal data property of this Date object to *v*.
7. 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 LocalTime(this time value).
2. Let *m* be ToNumber(*month*).
3. If *date* is not specified, then let *dt* be DateFromTime(*t*); otherwise, let *dt* be ToNumber(*date*).
4. Let *newDate* be MakeDate(MakeDay(YearFromTime(*t*), *m*, *dt*), TimeWithinDay(*t*)).
5. Let *u* be TimeClip(UTC(*newDate*)).
6. Set the [[DateValue]] internal data 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. ReturnIfAbrupt(*t*).
3. 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(YearFromTime(*t*), *m*, *dt*), TimeWithinDay(*t*)).
6. Let *v* be TimeClip(*newDate*).
7. Set the [[DateValue]] internal data property of this Date object to *v*.
8. 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 [[DateValue]] internal data 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. ReturnIfAbrupt(*t*).
3. Let *y* be ToNumber(*year*).
4. If *month* is not specified, then let *m* be MonthFromTime(*t*); otherwise, let *m* be ToNumber(*month*).
5. If *date* is not specified, then let *dt* be DateFromTime(*t*); otherwise, let *dt* be ToNumber(*date*).
6. Let *newDate* be MakeDate(MakeDay(*y*, *m*, *dt*), TimeWithinDay(*t*)).
7. Let *v* be TimeClip(*newDate*).
8. Set the [[DateValue]] internal data property of this Date object to *v*.
9. 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 parsable. 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 represent 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 Get(*O*, **`"toISOString"`**).
5. ReturnIfAbrupt(*toISO*).
6. If IsCallable(*toISO*) is **false**, throw a **TypeError** exception.
7. Return the result of calling the [[Call]] internal method of *toISO* with *O* as *thisArgument* and an empty List as *argumentsList*.

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.

15.9.5.4d Date.prototype.@@ToPrimitive (hint)

This function is called by ECMAScript language operators to convert an object to a primitive value. The allowed values for *hint* are "**`default`**", "**`number`**", and "**`string`**". Date objects, are unique among built-in ECMAScript object in that they treat "**`default`**" as being equivalent to "**`string`**", All other built-in ECMAScript objects treat "**`default`**" as being equivalent to "**`number`**".

When the **`@@ToPrimitive`** method is called with argument *hint*, the following steps are taken:

1. Let *O* be the **this**
2. If Type(*O*) is not Object, then throw a **TypeError** exception.
3. If *hint* is "**`string`**" or "**`default`**" , then
    a. Let *tryFirst* be " **`string`** ".
4. Else if *hint* is "**`number`**", then
    a. Let *tryFirst* be " **`number`** ".
5. Else, throw a **TypeError** exception.
6. Return the result of OrdinaryToPrimitive(*O*,*tryFirst*).

15.9.6 Properties of Date Instances

Date instances are ordinary objects that inherit properties from the Date prototype object and have a [[BuiltinBrand]] internal whose value is BuiltinDate. Date instances also have a [[DateValue]] internal data property.

The [[DateValue]] internal data 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 data 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^{th} 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^{th} element of *captures* is either a String that represents the value obtained by the n^{th} set of capturing parentheses or **undefined** if the n^{th} set of capturing parentheses hasn't been reached yet. Due to backtracking, many *States* may be in use at any time during the matching process.
- 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^{th} 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 *m2*.
3. Return an internal Matcher closure that takes two arguments, a State *x* and a Continuation *c*, and performs the following:
    1. Call *m1*(*x*, *c*) and let *r* be its result.
    2. If *r* isn't **failure**, return *r*.
    3. Call *m2*(*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 *m1*.
2. Evaluate *Term* to obtain a Matcher *m2*.
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 *m2*(*y*, *c*).
    2. Call *m1*(*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 ∞) *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.

Runtime Semantics: RepeatMatcher Abstract Operation

The abstract operation RepeatMatcher takes eight parameters, a Matcher *m*, an integer *min*, an integer (or ∞) *max*, a Boolean *greedy*, a State *x*, a Continuation *c*, an integer *parenIndex*, and an integer *parenCount*, and performs the following:

1. If *max* is zero, then call *c*(*x*) and return its result.
2. Create an internal Continuation closure *d* that takes one State argument *y* and performs the following:
    1. If *min* is zero and *y*'s *endIndex* is equal to *x*'s *endIndex*, then return **failure**.
    2. If *min* is zero then let *min2* be zero; otherwise let *min2* be *min*–1.
    3. If *max* is ∞, then let *max2* be ∞; otherwise let *max2* be *max*–1.
    4. Call RepeatMatcher(*m*, *min2*, *max2*, *greedy*, *y*, *c*, *parenIndex*, *parenCount*) and return its result.
3. Let *cap* be a fresh copy of *x*'s *captures* internal array.
4. For every integer *k* that satisfies *parenIndex* < *k* and *k* ≤ *parenIndex*+*parenCount*, set *cap*[*k*] to **undefined**.
5. Let *e* be *x*'s *endIndex*.
6. Let *xr* be the State (*e*, *cap*).
7. If *min* is not zero, then call *m*(*xr*, *d*) and return its result.
8. If *greedy* is **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("abcdefghi")
```

which returns **"abcde"** with

```
/a[a-z]{2,4}?/.exec("abcdefghi")
```

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:

```
"aaaaaaaaaa,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.

Runtime Semantics: IsWordChar Abstract Operation

The abstract operation *IsWordChar* takes an integer parameter *e* and performs the following:

1. If *e* is –1 or *e* is *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 ∞) *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 ∞) *max*.
2. Return the three results *min*, *max*, and **false**.

The production *QuantifierPrefix* **::** ***** evaluates by returning the two results 0 and ∞.

The production *QuantifierPrefix* **::** **+** evaluates by returning the two results 1 and ∞.

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 ∞.

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.

Runtime Semantics: CharacterSetMatcher Abstract Operation

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* is *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*) is *cc*, return **failure**.
    6. Else *invert* is **true**,
        a. If there exists a member *a* of set *A* such that *Canonicalize*(*a*) is *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.

Runtime Semantics: Canonicalize Abstract Operation

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("baaabaac")
```

looks for an **`a`** not immediately followed by some positive number n of **`a`**'s, a **`b`**, another n **`a`**'s (specified by the first **`\2`**) and a **`c`**. The second **`\2`** is outside the negative lookahead, so it matches against **undefined** and therefore always succeeds. The whole expression returns the array:

```
["baaabaac", "ba", undefined, "abaac"]
```

In case-insignificant 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 **`"ß"`** (`\u00DF`) 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 33.

Table 33 — ControlEscape Character Values

ControlEscape	*Code Unit*	*Name*	*Symbol*
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>

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*.

Runtime Semantics: CharacterRange Abstract Operation

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 [[BuiltinBrand]] internal data property whose value is BuiltinRegExp and *flags* is **undefined**, then return *pattern*.
2. Return the result of the abstract 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 abstract operation RegExpCreate with arguments *pattern* and *flags*.

Runtime Semantics: RegExpCreate Abstract Operation

The abstract operation RegExpCreate with arguments pattern and flags does the following:

If *pattern* is an object *R* that has a [[BuiltinBrand]] internal data property whose value is BuiltinRegExp 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 [[BuiltinBrand]] internal data property whose value is BuiltinRegExp 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 data 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 data 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 **"(?:)"**.

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 **multiline** 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 data 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 [[BuiltinBrand]] internal data property whose value is BuiltinRegExp

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 data property of the RegExp constructor is the standard built-in Function prototype object (15.3.4).

Besides 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 data 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 [[BuiltinBrand]] internal data property whose value is BuiltinRegExp . 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 [[BuiltinBrand]] internal data property whose value is BuiltinRegExp.

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 abstract operation with arguments *R* and *S*.

Runtime Semantics: RegExpExec Abstract Operation

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 Get(*R*,"**`lastIndex`**").
3. Let *i* be the value of ToInteger(*lastIndex*).

4. ReturnIfAbrupt(*i*).
5. Let *global* be the result of Get(*R*, "**`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 Put(*R*, "**`lastIndex`**", 0, **true**).
        ii. ReturnIfAbrupt(*putStatus*).
        iii. Return **null**.
    b. Let *r* be the result of calling the [[Match]] internal method of *R* with arguments *S* and *i*.
    c. If *r* is **failure**, then
        i. *Let i = i+1.*
    d. else
        i. Assert: *r* is a State.
        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 Put(*R*, "**`lastIndex`**", *e*, **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`**" and Property Descriptor {[[Value]]: *S*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
18. Call the [[DefineOwnProperty]] internal method of *A* with arguments "**`length`**" and Property Descriptor {[[Value]]: *n* + 1}.
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`**" and Property Descriptor {[[Value]]: *matchedSubstr*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
21. For each integer *i* such that *i* > 0 and *i* ≤ *n*
    a. Let *captureI* be i^{th} element of *r*'s *captures* array.
    b. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*i*) and Property Descriptor {[[Value]]: *captureI*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **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 abstract 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 [[BuiltinBrand]] internal data property whose value is BuiltinRegExp. RegExp instances also have a [[Match]] internal data property and a **`length`** property.

The value of the [[Match]] internal data 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** }.

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 data 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 [[BuiltinBrand]] internal data property whose value is BuiltinError.

The [[Extensible]] internal data 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*).

15.11.2 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 data 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 [[BuiltinBrand]] internal data property whose value is BuiltinError .

The [[Extensible]] internal data 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*).

15.11.3 Properties of the Error Constructor

The value of the [[Prototype]] internal data 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 [[BuiltinBrand]] internal data property whose value is BuiltinError .

The value of the [[Prototype]] internal data property of the Error prototype object is the standard built-in Object prototype object (15.2.4).

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 Get(*O*, **`"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 Get(*O*, **`"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*.

15.11.5 Properties of Error Instances

Error instances are ordinary objects that inherit properties from the Error prototype object and have a [[BuiltinBrand]] internal data property whose value is BuiltinError

15.11.6 Native Error Types Used in This Standard

One of the *BuiltinError* 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.

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.

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 data property of the newly constructed object is set to the prototype object for this error constructor. The newly constructed object has a [[BuiltinBrand]] internal data property whose value is BuiltinError . The [[Extensible]] internal data 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*).

15.11.7.3 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 data property of the newly constructed object is set to the prototype object for this *NativeError* constructor. The newly constructed object has a [[BuiltinBrand]] internal data property whose value is BuiltinError . The [[Extensible]] internal data 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 data property of a *NativeError* constructor is the Function prototype object (15.3.4).

Besides the **`length`** property (whose value is **1**), each *NativeError* constructor has the following property:

15.11.7.5.1 *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** }.

15.11.7.6 Properties of the *NativeError* Prototype Objects

Each *NativeError* prototype object is an Error object and has a [[BuiltinBrand]] internal data property whose value is BuiltinError .

The value of the [[Prototype]] internal data property of each *NativeError* prototype object is the standard built-in Error prototype object (15.11.4).

15.11.7.6.1 *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).

15.11.7.6.2 *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*).

15.11.7.6.3 *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.

15.11.7.8 Properties of *NativeError* Instances

NativeError instances inherit properties from their *NativeError* prototype object and have a [[BuiltinBrand]] internal data property whose value is BuiltinError . *NativeError* instances have no special properties.

15.12 The JSON Object

The **JSON** object is a single ordinary 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 data property of the JSON object is the standard built-in Object prototype object (15.2.4). The JSON object has a [[BuiltinBrand]] internal data property whose value is BuiltinJSON . The value of the [[Extensible]] internal data property of the JSON object is set to **true**.

The JSON object does not have a [[Construct]] internal method; 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 method; 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 Unicode 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 **::**
" *JSONStringCharacters*$_{opt}$ **"**

JSONStringCharacters **::**
JSONStringCharacter JSONStringCharacters$_{opt}$

JSONStringCharacter **::**
SourceCharacter **but not one of** **"** **or** **** **or** *U+0000* **through** *U+001F*
**** *JSONEscapeSequence*

JSONEscapeSequence **::**
JSONEscapeCharacter
u *HexDigit HexDigit HexDigit HexDigit*

JSONEscapeCharacter **:: one of**
" / \ b f n r t

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

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 JSON.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* interpreted as UTF-16 encoded Unicode characters 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 *scriptText* be the result of concatenating **"("**, *JText* , and , **");"**.
5. Let *completion* be the result of parsing and evaluating *scriptText* as if it was the source text of an ECMAScript *Script* 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*.
6. Let *unfiltered* be *completion*.[[value]].
7. If IsCallable(*reviver*) is **true**, then
    a. Let *root* be the result of the abstract operation ObjectCreate.
    b. Call CreateOwnDataProperty(*root*, the empty String, *unfiltered*).
    c. Return the result of calling the abstract operation Walk, passing *root* and the empty String. The abstract operation Walk is described below.
8. Else
    a. Return *unfiltered*.

Runtime Semantics: Walk Abstract Operation

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 Get(*holder*, *name*).
2. ReturnIfAbrupt(*val*).
3. If *val* is an object, then
    a. If *val* has a [[BuiltinBrand]] internal data property with value BuiltinArray, then
        i. Set *I* to 0.
        ii. Let *len* be the result of Get(*val*, **"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*) as the argument.
            3. Else
                a Let *status* be the result of calling the [[DefineOwnProperty]] internal method of *val* with arguments ToString(*I*) and Property Descriptor {[[Value]]: *newElement*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
            4. ReturnIfAbrupt(*status*).
            5. Add 1 to *I*.
    b. Else
        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* as the argument.
            3. Else
                a Let *status* be the result of calling the [[DefineOwnProperty]] internal method of *val* with arguments *P* and Property Descriptor {[[Value]]: *newElement*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
            4. ReturnIfAbrupt(*status*).
4. Return the result of calling the [[Call]] internal method of *reviver* passing *holder* as *thisArgument* and with a List containing *name* and *val* as *argumentsList*.

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 JSON.stringify (value [, replacer [, space]])

The **stringify** function returns a String in UTF-16 encoded 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 [[BuiltinBrand]] internal data property with value BuiltinArray , 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 a [[StringData]] or [[NumberData]] internal data property, 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 [[NumberData]] internal data property then,
        i. Let *space* be ToNumber(*space*).
    b. Else if *space* has a [[StringData]] internal data property then,
        i. Let *space* be ToString(*space*).
6. If Type(*space*) is Number
    a. Let *space* be min(10, ToInteger(*space*)).
    b. Set *gap* to a String containing *space* occurrences of code unit 0x0020 (the Unicode space character). This will be the empty String if *space* is less than 1.
7. Else if Type(*space)* is String
    a. If the number of elements in *space* is 10 or less, set *gap* to *space* otherwise set *gap* to a String consisting of the first 10 elements of *space*.
8. Else
    a. Set *gap* to the empty String.
9. Let *wrapper* be the result of the abstract operation ObjectCreate.
10. Call CreateOwnDataProperty(*wrapper*, the empty String, *value*).
11. Return the result of calling the abstract operation Str with the empty String and *wrapper*.

Runtime Semantics: Str Abstract Operation

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 Get(*holder*, *key*).
2. ReturnIfAbrupt(*value*).
3. If Type(*value*) is Object, then
    a. Let *toJSON* be the result of Get(*value*, **`"toJSON"`**).
    b. If IsCallable(*toJSON*) is **true**
        i. Let *value* be the result of calling the [[Call]] internal method of *toJSON* passing *value* as *thisArgument* and a List containing *key* as *argumentsList*.
        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 [[NumberData]] internal data property then,
        i. Let *value* be ToNumber(*value*).
    b. Else if *value* has an [[StringData]] internal data property then,
        i. Let *value* be ToString(*value*).

   c. Else if *value* has an [[BooleanData]] internal data property then,
      i. Let *value* be the value of the [[BooleanData]] internal data 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 [[BuiltinBrand]] internal data property with value BuiltinArray then
       i. Return the result of calling the abstract operation *JA* with argument *value*.
    b. Else, return the result of calling the abstract operation *JO* with argument *value*.
12. Return **undefined**.

Runtime Semantics: Quote Abstract Operation

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

1. Let *product* be code unit 0x0022 (the Unicode double quote character).
2. For each code unit *C* in *value*
   a. If *C* is 0x0022 or 0x005C (the Unicode reverse solidus character)
      i. Let *product* be the concatenation of *product* and code unit 0x005C (the Unicode backslash character).
      ii. Let *product* be the concatenation of *product* and code unit 0x005C.
   b. Else if *C* is backspace, formfeed, newline, carriage return, or tab
      i. Let *product* be the concatenation of *product* and code unit 0x005C (the Unicode backslash character).
      ii. Let *abbrev* be the string value corresponding to the value of *C* as follows:

backspace	**"b"**
formfeed	**"f"**
newline	**"n"**
carriage return	**"r"**
tab	**"t"**

      iii. Let *product* be the concatenation of *product* and *abbrev*.
   c. Else if *C* has a code unit value less than 0x0020 (the Unicode space character)
      i. Let *product* be the concatenation of *product* and code unit 0x005C (the Unicode backslash character).
      ii. Let *product* be the concatenation of *product* and **"u"**.
      iii. Let *hex* be the string 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 code unit 0x0022 (the Unicode double quote character).
4. Return *product*.

Runtime Semantics: JO Abstract Operation

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.

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 string **":"**.
      iii. If *gap* is not the empty String
         1. Let *member* be the concatenation of *member* and code unit 0x0020 (the Unicode 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 code unit 0x002C (the Unicode 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 code unit 0x002C (the comma character), code unit 0x000A (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 **"{"**, code unit 0x000A (the line feed character), *indent*, *properties*, code unit 0x000A, *stepback*, and **"}"**.
11. Remove the last element of *stack*.
12. Let *indent* be *stepback*.
13. Return *final*.

Runtime Semantics: JA Abstract Operation

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.

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 standard array object and hence its **"length"** property is a non-negative integer.
7. Let *len* be the result Get(*value*, **"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 code unit 0x002C (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 code unit 0x002C (the comma character), code unit 0x000A (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 **"["**, code unit 0x000A (the line feed character), *indent*, *properties*, code unit 0x000A, *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 **\u**HHHH, 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.

15.13 Binary Data Objects

15.13.1 The BinaryData Module

15.13.2 The BinaryData.Type Object

15,13.2.5 BinaryData.*ScalarType* Type Instance Objects

15.13.3 The BinaryData.ArrayType Object

15.13.4 The BinaryData.StructType Object

The following sections defined "typed arrays" derived from the Kronos specification. This material is a very early draft based upon the strawman at http://wiki.ecmascript.org/doku.php?id=strawman:typed_arrays . This material still needs significant work to fully integrate it into the ES6 spec. and also to integrate typed arrays with ES6 binary data.

Don't waste a lot of time reviewing this material until it is closer to a finished state.

15.13.5 ArrayBufferObjects

15.13.5.1 The ArrayBuffer Object Called as a Function

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

15.13.5.2 The ArrayBuffer Constructor

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

15.13.5.2.1 new ArrayBuffer(len)

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

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

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

15.13.5.3 Properties of the ArrayBuffer Constructor

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

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

15.13.5.3.1 ArrayBuffer.prototype

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

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

15.13.5.4 Properties of the ArrayBuffer Prototype Object

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

15.13.5.4.1 ArrayBuffer.prototype.constructor

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

15.13.5.5 Properties of the ArrayBuffer Instances

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

15.13.5.5.1 byteLength

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

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

15.13.6 *TypeArray* Object Structures

For each constructor in the following table is a separate *TypeArray* constructor object, with corresponding prototype and instances. Each of these *TypeArray* constructor objects has the structure described below, differing only in the name used as the constructor name instead of *TypeArray*, in XXXXXXX.

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

In the definitions below, references to *TypeArray* should be replaced with the appropriate constructor name from the above table. The phrase "the element size in bytes" refers to the value in the Element Size column of the table in the row corresponding to the constructor. The phrase "element Type" refers to the value in the Element Type column for that row.

15.13.6.1 *TypeArray* Constructors Called as a Function

When a *TypeArray* constructor is called as a function rather than as a constructor, it creates and initialises a new object. A call of the constructor as a function is equivalent to calling it as a constructor with the same arguments.

15.13.6.2 The *TypeArray* Constructors

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

15.13.6.2.1 new *TypeArray*(arg0 [, arg1, [, arg2])

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

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

1. If Type(*arg0*) is Number, then
    a. Let *length* be ToUInt32(*arg0*).
    b. ReturnIfAbrupt(*length*).
    c. The length property of the newly constructed object is set to *length*.
    d. The byteLength property of the newly constructed object is set to *length* multiplied by the element size in bytes.
    e. Let arrayBuffer be an object constructed as if by a call to the built-in ArrayBuffer constructor, as "new ArrayBuffer(byteLength)".
    f. The buffer property of the newly constructed object is set to arrayBuffer.
    g. The byteOffset property of the newly constructed object is set to 0.
2. Else,
    a. Let *O* be the result of calling ToObject(arg0).
    b. ReturnIfAbrupt(*O*).
    c. Let *class* be the value of the [[Class]] internal data property of *O*.
    d. If *class* is "ArrayBuffer", then
        i. Let *byteOffset* be the result of calling ToUInt32 on arg1, if provided, or else 0.
        ii. If *byteOffset* is not an integer multiple of the element size in bytes, throw a RangeError exception. data
        iii. Let *bufferLength* be the result of Get(*O*, **"byteLength"**).
        iv. Let *byteLength* be the result of calling ToUInt32 on arg2, if provided, or else bufferLength – byteOffset.
        v. If *byteOffset* + *byteLength* is greater than *bufferLength*, throw a RangeError exception.
        vi. Let length be the result of dividing *byteLength* by the element size in bytes.
        vii. If ToUInt32(*length*) ≠ *length*, throw a RangeError exception.
        viii. The length property of the newly constructed object is set to *length*.
        ix. The byteLength property of the newly constructed object is set to *byteLength*.
        x. The buffer property of the newly constructed object is set to *O*.
        xi. The byteOffset property of the newly constructed object is set to *byteOffset*.
    e. Else,
        i. Let *n* to be the result of Get(*V*, **"length"**).
        ii. Let length be the result of calling ToUInt32(*n*).
        iii. The length property of the newly constructed object is set to length.
        iv. The byteLength property of the newly constructed object is set to length multiplied by the element size in bytes.
        v. Let *arrayBuffer* be an object constructed as if by a call to the built-in ArrayBuffer constructor, as "new ArrayBuffer(byteLength)".
        vi. Let *i* to be 0.
        vii. While *i* < *length*:
            1. Let *x* be the result of Get(*arrayBuffer*, ToString(*i*)).

2. Let *indexDesc* be Property Descriptor {[[Value]]: *x*,[[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **false**}.
3. Call the [[DefineOwnProperty]] on the newly constructed object with arguments ToString(*i*) and *indexDesc*.
4. Set *i* to *i* + 1.

viii. The buffer property of the newly constructed object is set to *arrayBuffer*.

ix. The byteOffset property of the newly constructed object is set to 0.

15.13.6.3 Properties of the *TypeArray* Constructors

The value of the [[Prototype]] internal data property of each *TypeArray* constructor is the Function prototype object (15.3.4).

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

15.13.6.3.1 *TypeArray*.prototype

The initial value of *TypeArray*.prototype is the *TypeArray* prototype object (15.13.2.4).

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

15.13.6.3.2 *TypeArray*.BYTES_PER_ELEMENT

The initial value of *TypeArray*.BYTES_PER_ELEMENT is the element size in bytes.

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

15.13.6.4 Properties of the *TypeArray* Prototype Object

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

15.13.6.4.1 *TypeArray*.prototype.constructor

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

15.13.6.4.2 *TypeArray*.prototype.set(array [, offset])

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

1. If this does not have class "TypeArray", throw a **TypeError** exception.
2. Let offsetIndex be ToUInt32(offset)
3. Let O be the result of calling ToObject(array).
4. Let srcLength be the result Get(*O*, **`"length"`**).
5. Let targetLength be the result of Get(this, **`"length"`**).
6. If srcLength + offset > targetLength, throw a **RangeError** exception.
7. Let temp be a new TypeArray created as if by a call to "new TypeArray(srcLength)"
8. Let k be 0
9. While *k* < *srcLength*
    a. Let *v* be the result of Get(*src*, toString(*k*)).
    b. Call Put(*temp*, ToString(*k*), *v*, **false**).
10. Let *k* be *offset*.
11. While k < targetLength
    c. Let *v* be the result of Get(*temp*, ToString(*k*-*offset*)).
    d. Call Put(*temp*, ToString(*k*), *v*, **false**).

15.13.6.4.3 *TypeArray*.prototype.subarray(begin [, end])

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

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

15.13.6.5 Properties of *TypeArray* instances

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

15.13.6.5.1 [[DefineOwnProperty]] (p, desc, throw)

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

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

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

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

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

15.13.6.5.2 [[GetOwnProperty]] (P)

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

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

1. Let desc be the result of calling the default [[GetOwnProperty]] internal method (8.12.1) on A with argument P.
2. If desc is not undefined return desc.
3. If ToString(abs(ToInteger(P))) is not the same value as P, return undefined.
4. Let length be the result of Get(*A*, **`"length"`**).
5. Let index be ToInteger(P).
6. If length ≤ index, return undefined.
7. Let isLittleEndian be **true** if the platform endianness is little endian, else **false**.
8. Let value be the result of calling the GetValueFromBuffer internal operation with arguments A.buffer.[[NativeBuffer]], A.byteOffset, index, Type, and isLittleEndian.
9. Return a Property Descriptor { [[Value]]: value, [[Enumerable]]: **true**, [[Writable]]: **true**, [[Configurable]]: **false** }

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

1. Let size be the size in bytes of the type valueType.
2. Let bytes be the array of bytes from nativeBuffer between offset byteOffset+(index*size) and offset byteOffset+((index+1) × size)-1 inclusive.
3. Let rawValue be the result of convert the array bytes to a value of type valueType, using little endian if isLittleEndian is **true**, otherwise big endian.
4. If valueType is Float32 and rawValue is a Float32 representation of IEEE754 NaN, return the NaN Number value.
5. Else, if valueType is Float64 and rawValue is a Float64 representation of IEEE754 NaN, return the NaN Number value.
6. Else, return the Number value that that represents the same numeric value as rawValue

15.13.6.5.3 length

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

15.13.6.5.4 byteLength

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

15.13.6.5.5 buffer

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

15.13.6.5.6 byteOffset

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

15.13.7 DataView Objects

15.13.7.1 The DataView Constructor Called as a Function

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

15.13.7.2 The DataView Constructor

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

15.13.7.2.1 new DataView(buffer [, byteOffset [, byteLength]])

The [[Prototype]] internal data property of the newly constructed object is set to the original DataView prototype object, the one that is the initial value of DataView.prototype (15.13.3.3.1). The [[Class]] internal data property of the newly constructed object is set to "DataView". The [[Extensible]] internal data property of the newly constructed object is set to true.

The remaining properties are set as follows:

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

15.13.7.3 Properties of the DataView Constructor

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

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

15.13.7.3.1 DataView.prototype

The initial value of DataView.prototype is the DataView prototype object (15.13.3.4).

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

15.13.7.4 Properties of the DataView Prototype Object

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

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

1. Let byteOffsetInt be ToUInt32(byteOffset)
2. Let totalOffset be byteOffsetInt plus the result of Get(this, **"byteOffset"**).
3. Let byteLength be the result of Get(this, **"byteLength"**).
4. If totalOffset ≥ byteLength, throw a RangeError exception.

5. Let value be the result of calling the GetValueFromBuffer internal operation (2.5.2) with arguments this.buffer.[[NativeBuffer]], totalOffset, 0 and type.
6. Return value

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

1. Let byteOffsetInt be ToUInt32(byteOffset)
2. Let totalOffset be byteOffsetInt plus the result of Get(this, **"byteOffset"**).
3. Let byteLength be the result of Get(this, **"byteLength"**).
4. If totalOffset ≥ byteLength, throw a RangeError exception.
5. Let value be the result of calling the SetValueInBuffer internal operation (2.5.2) with arguments this.buffer.[[NativeBuffer]], totalOffset, 0, value and type.
6. Return value

15.13.7.4.1 DataView.prototype.constructor

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

15.13.7.4.2 DataView.prototype.getInt8(byteOffset)

Gets the Int8 value at offset byteOffset in the DataView.

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

15.13.7.4.3 DataView.prototype.getUint8(byteOffset)

Gets the UInt8 value at offset byteOffset in the DataView.

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

15.13.7.4.4 DataView.prototype.getInt16(byteOffset, littleEndian)

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

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

15.13.7.4.5 DataView.prototype.getUint16(byteOffset, littleEndian)

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

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

15.13.7.4.6 DataView.prototype.getInt32(byteOffset, littleEndian)

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

1. Let O be ToObject(this)
2. Let isLittleEndian be ToBoolean(littleEndian) if provided, else false

3. If the [[Class]] internal data property of O is not “DataView”, throw a TypeError exception.
4. Return GetValue(byteOffset, isLittleEndian, Int32)

15.13.7.4.7 DataView.prototype.getUint32(byteOffset, littleEndian)

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

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

15.13.7.4.8 DataView.prototype.getFloat32(byteOffset, littleEndian)

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

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

15.13.7.4.9 DataView.prototype.getFloat64(byteOffset, littleEndian)

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

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

15.13.7.4.10 DataView.prototype.setInt8(byteOffset, value)

Sets the Int8 value at offset byteOffset in the DataView.

1. Let O be ToObject(this)
2. If the [[Class]] internal data property of O is not “DataView”, throw a TypeError exception.
3. Return SetValue(byteOffset, **true**, Int8, ToInt8(value))

15.13.7.4.11 DataView.prototype.setUint8(byteOffset, value)

Sets the Uint8 value at offset byteOffset in the DataView.

4. Let O be ToObject(this)
5. If the [[Class]] internal data property of O is not “DataView”, throw a TypeError exception.
6. Return SetValue(byteOffset, **true**, Uint8, ToUint8(value))

15.13.7.4.12 DataView.prototype.setInt16(byteOffset, value, littleEndian)

Sets the Int16 value at offset byteOffset in the DataView.

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

15.13.7.4.13 DataView.prototype.setUint16(byteOffset, value, littleEndian)

Sets the Uint16 value at offset byteOffset in the DataView.

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

15.13.7.4.14 DataView.prototype.setInt32(byteOffset, value, littleEndian)

Sets the Int32 value at offset byteOffset in the DataView.

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

15.13.7.4.15 DataView.prototype.setUint32(byteOffset, value, littleEndian)

Sets the Uint32 value at offset byteOffset in the DataView.

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

15.13.7.4.16 DataView.prototype.setFloat32(byteOffset, value, littleEndian)

Sets the Float32 value at offset byteOffset in the DataView.

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

15.13.7.4.17 DataView.prototype.setUint16(byteOffset, value, littleEndian)

Sets the Float64 value at offset byteOffset in the DataView.

1. Let O be ToObject(this)
2. Let isLittleEndian be ToBoolean(littleEndian) if provided, else false
3. If the [[Class]] internal data property of O is not “DataView”, throw a TypeError exception.
4. Return SetValue(byteOffset, isLittleEndian, Float64, ToFloat64(value))

15.13.7.5 Propeties of DataView Instances

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

15.13.7.5.1 byteLength

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

15.13.7.5.2 buffer

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

15.13.7.5.3 byteOffset

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

15.14 Map Objects

Map objects are collections of key/value pairs where both the keys and values may be arbitrary ECMAScript language values. A Map object can also iterate its elements in insertion order. Map object must be implemented using hash tables or other mechanisms that, on average, provide access times that are sublinear on the number of elements in the collection. The data structures used in this Map objects specification is only intended to describe the required observable semantics of Map objects. It is not intended to be a viable implementation model.

15.14.1 Abstract Operations For Map Objects

15.14.1.1 MapInitialisation

The abstract operation MapInitialisation with arguments *obj* and *iterable* is used to initialize an object as a map. It performs the following steps:

1. If Type(*obj*) is not Object, throw a **TypeError** exception.
2. If *obj* already has a [[MapData]] internal data property, throw a **TypeError** exception.
3. If the result of calling the [[GetExtensible]] internal method of *obj* is **false**, throw a TypeError exception.
4. If *iterable* is not **undefined**, then
    a. Let *iterable* be ToObject(*iterable*).
    b. ReturnIfAbrupt(*iterable*)
    c. Let *iterator* be the intrinsic symbol @@iterator.
    d. Let *itr* be the result of Invoke(*obj*, *iterator*).
    e. ReturnIfAbrupt(*itr*).
    f. Let *adder* be the result of Get(*obj*, **`"set"`**).
    g. ReturnIfAbrupt(*adder*).
    h. If IsCallable(*adder*) is **false**, throw a **TypeError** Exception.
5. Add a [[MapData]] internal data property to *obj*.
6. Set *obj*'s [[MapData]] internal data property to a new empty List.
7. If *iterable* is **undefined**, return *obj*.
8. Repeat
    a. Let *next* be the result of Invoke(*itr*, **`"next"`**).
    b. If IteratorComplete(*next*) is **true**, then return NormalCompleti)on(*obj*).
    c. Let *next* be ToObject(*next*).
    d. ReturnIfAbrupt(*next*).
    e. Let *k* be the result of Get(*next* ,, **`"0"`**).
    f. ReturnIfAbrupt(*k*).
    g. Let *v* be the result of Get(*next*, **`"1"`**).
    h. ReturnIfAbrupt(*v*).
    i. Let *status* be the result of calling the [[Call]] internal method of *adder* with *obj* as *thisArgument* and a List whose elements are *k* and *v* as *argumentsList*.
    j. ReturnIfAbrupt(*status*).

15.14.2 The Map Constructor Called as a Function

When **`Map`** is called as a function rather than as a constructor, it initializes its **this** value with the internal state necessary to support the **`Map.prototype`** internal methods. This permits super invocation of the **`Map`** constructor by **`Map`** subclasses.

15.14.2.1 Map (iterable = undefined)

1. Let *m* be the **this** value.
2. If *m* is **undefined** or the intrinsic %MapPrototype%

   a. Let *map* be the result of the abstract operation ObjectCreate with the intrinsic %MapPrototype% as the argument.
3. Else
   a. Let *map* be the result of ToObject(*m*).
4. ReturnIfAbrupt(*map*).
5. If *iterable* is not present, let *iterable* be **undefined**.
6. Let *status* be the result of MapInitialisation with *map* and *iterable* as arguments.
7. ReturnIfAbrupt(*status*).
8. Return *map*.

NOTE If the parameter iterable is present, it is expected to be an object that implements an @@iterator method that returns an iterator object that produces two element array-like objects whose first element is a value that will be used as an Map key and whose second element is the value to associate with that key.

15.14.3 The Map Constructor

When **`Map`** is called as part of a **`new`** expression it is a constructor: it initialises the newly created object.

15.14.3.1 new Map (iterable = undefined)

1. Let *map* be the result of the abstract operation ObjectCreate with the intrinsic %MapPrototype% as the argument.
2. If *iterable* is not present, let *iterable* be **undefined**.
3. Let *status* be the result of MapInitialisation with *map* and *iterable* as arguments.
4. ReturnIfAbrupt(*status*).
5. Return *map*.

NOTE If the parameter iterable is present, it is expected to be an object that implements an @@iterator method that returns an iterator object that produces two element array-like objects whose first element is a value that will be used as an Map key and whose second element is the value to associate with that key.

15.14.4 Properties of the Map Constructor

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

Besides the **`length`** property (whose value is **0**), the Map constructor has the following property:

15.14.4.1 Map.prototype

The initial value of **`Map.prototype`** is the Map prototype object (15.14.4).

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

15.14.5 Properties of the Map Prototype Object

The value of the [[Prototype]] internal data property of the Map prototype object is the standard built-in Object prototype object (15.2.4).

15.14.5.1 Map.prototype.constructor

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

15.14.5.2 Map.prototype.clear ()

The following steps are taken:

1. Let *M* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*M*).
3. If *M* does not have a [[MapData]] internal data property throw a **TypeError** exception.

4. Set the value of *M*'s [[MapData]] internal data property to a new empty List.
5. Return **undefined**.

15.14.5.3 Map.prototype.delete (key)

The following steps are taken:

1. Let *M* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*M*).
3. If *M* does not have a [[MapData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *M*'s [[MapData]] internal data property.
5. Repeat for each Record {[[key]], [[value]]} *p* that is an element of *entries*,
    a. If SameValue(*p*.[[key]], *key*), then
        i. Set *p*.[[key]] to empty.
        ii. Set *p*.[[value]] to empty.
        iii. Return **true**.
6. Return **false**.

15.14.5.4 Map.prototype.forEach (callbackfn , thisArg = undefined)

callbackfn should be a function that accepts three arguments. **`forEach`** calls *callbackfn* once for each key/value pair present in the map object, in key insertion order. *callbackfn* is called only for keys of the map which actually exist; it is not called for keys that have been deleted from the map.

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.

NOTE If *callbackfn* is an Arrow Function, **this** was lexically bound when the function was created so *thisArg* will have no effect.

callbackfn is called with three arguments: the value of the item, the key of the item, and the Map 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*.

NOTE Each key is visited only once with the value that is current at the time of the visit. If the value associated with a key is modified after it has been visited, it is not re-visited. Keys that are deleted after the call to **`forEach`** begins and before being visited are not visited. New keys added, after the call to **`forEach`** begins are visited.

When the **`forEach`** method is called with one or two arguments, the following steps are taken:

1. Let *M* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*M*).
3. If *M* does not have a [[MapData]] internal data property throw a **TypeError** exception.
4. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
5. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
6. Let *entries* be the List that is the value of *M*'s [[MapData]] internal data property.
7. Repeat for each Record {[[key]], [[value]]} *e* that is an element of *entries*, in original key insertion order
    a. If *e*.[[key]] is not empty, then
        i. Let *funcResult* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as *thisArgument* and a *List* containing *e*.[[value]], *e*.[[key]], and *M as argumentsList*.
        ii. ReturnIfAbrupt(*funcResult*).
8. Return **undefined**.

The **`length`** property of the **`forEach`** method is **1**.

15.14.5.5 Map.prototype.get (key)

The following steps are taken:

1. Let *M* be the result of calling ToObject with the **this** value the as its argument.
2. ReturnIfAbrupt(*M*).
3. If *M* does not have a [[MapData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *M*'s [[MapData]] internal data property.
5. Repeat for each Record {[[key]], [[value]]} *p* that is an element of *entries*,
    a. If SameValue(*p*.[[key]], *key*), then return *p*.[[value]]
6. Return **undefined**.

15.14.5.6 Map.prototype.has (key)

The following steps are taken:

1. Let *M* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*M*).
3. If *M* does not have a [[MapData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *M*'s [[MapData]] internal data property.
5. Repeat for each Record {[[key]], [[value]]} *p* that is an element of *entries*,
    a. If SameValue(*p*.[[key]], *key*), then return **true**.
6. Return **false**.

15.14.5.7 Map.prototype.items ( )

The following steps are taken:

1. Let *M* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*M*).
3. Return the result of calling the CreateMapIterator abstract operation with arguments *M* and "**`key+value`**".

15.14.5.8 Map.prototype.keys ( )

The following steps are taken:

1. Let *M* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*M*).
3. Return the result of calling the CreateMapIterator abstract operation with arguments *M* and "**`key`**".

15.14.5.9 Map.prototype.set (key , value)

The following steps are taken:

1. Let *M* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*M*).
3. If *M* does not have a [[MapData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *M*'s [[MapData]] internal data property.
5. Repeat for each Record {[[key]], [[value]]} *p* that is an element of *entries*,
    a. If SameValue(*p*.[[key]], *key*), then
        i. Set *p*.[[value]] to *value*.
        ii. Return **undefined**.
6. Let *p* be the Record {[[key]]: *key*, [[value]]: *value*}
7. Append *p* as the last element of *entries*.
8. Return **undefined**.

15.14.5.10 get Map.prototype.size

Map.prototype.size is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *M* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*M*).

3. If *M* does not have a [[MapData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *M*'s [[MapData]] internal data property.
5. Let *count* be 0.
6. For each Record {[[key]], [[value]]} *p* that is an element of *entries*
    a. If *p*.[[key]] is not empty then
        i. Set *count* to *count*+1.
7. Return *count*.

15.14.5.11 Map.prototype.values ( )

The following steps are taken:

1. Let *M* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*M*).
3. Return the result of calling the CreateMapIterator abstract operation with arguments *M* and "**`value`**".

15.14.5.12 Map.prototype.@@iterator ( )

The initial value of the @@iterator property is the same function object as the initial value of the **items** property.

15.14.5.13 Map.prototype.@@toStringTag

The initial value of the @@toStringTag property is the string value **"`Map`"**.

15.14.6 Properties of Map Instances

Map instances inherit properties from the Map prototype. After initialisation by the Map constructor, Map instances also have a [[MapData]] internal data property.

15.14.7 Map Iterator Object Structure

A Map Iterator is an object, with the structure defined below, that represent a specific iteration over some specific Map instance object. There is not a named constructor for Map Iterator objects. Instead, map iterator objects are created by calling certain methods of Map instance objects.

15.14.7.1 CreateMapIterator Abstract Operation

Several methods of Map objects return interator objects. The abstract operation CreateMapIterator with arguments *map* and *kind* is used to create and such iterator objects. It performs the following steps:

1. Let *M* be the result of calling ToObject(*map*).
2. ReturnIfAbrupt(*M*).
3. If *M* does not have a [[MapData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *M*'s [[MapData]] internal data property.
5. Let *itr* be the result of the abstract operation ObjectCreate with the intrinsic object %MapIteratorPrototype% as its argument.
6. Add a [[Map]] internal data property to *itr* with value *M*.
7. Add a [[MapNextIndex]] internal data property to *itr* with value 0.
8. Add a [[MapIterationKind]] internal data property of *itr* with value *kind*.
9. Return *itr*.

15.14.7.2 The Map Iterator Prototype

All Map Iterator Objects inherit properties from a common Map Iterator Prototype object. The [[Prototype]] internal data property of the Map Iterator Prototype is the %ObjectPrototype% intrinsic object. In addition, the Map Iterator Prototype as the following properties:

15.14.7.2.1 *MapIterator*.prototype.constructor

15.14.7.2.2 *MapIterator*.prototype.next( )

1. Let *O* be the **this** value.
2. If Type(*O*) is not Object, throw a **TypeError** exception.
3. If *O* does not have all of the internal properties of a Map Iterator Instance (15.14.7.1.2), throw a **TypeError** exception.
4. Let *m* be the value of the [[Map]] internal data property of *O*.
5. Let *index* be the value of the [[MapNextIndex]] internal data property of *O*.
6. Let *itemKind* be the value of the [[MapIterationKind]] internal data property of *O*.
7. Assert: *m* has a [[MapData]] internal data property.
8. Let *entries* be the List that is the value of the [[MapData]] internal data property of *m*.
9. Repeat while *index* is less than the total number of element of *entries*. The number of elements must be redetermined each time this method is evaluated.
    a. Let *e* be the Record {[[key]], [[value]]} at 0-origined insertion position *index* of *entries*.
    b. Set *index* to *index*+1;
    c. Set the [[MapNextIndex]] internal data property of *O* to *index*.
    d. If *e*.[[key]] is not empty, then
        i. If *itemKind* is "**key**" then, let *result* be *e*.[[key]].
        ii. Else if *itemKind* is "**value**" then, let *result* be *e*.[[value]].
        iii. Else,
            1. Assert: *itemKind* is "**key+value**".
            2. Let *result* be the result of the abstract operation ArrayCreate with argument 2.
            3. Assert: *result* is a new, well-formed Array object so the following operations will never fail.
            4. Call CreateOwnDataProperty(*result*, "**0**", Property Descriptor {[[Value]]: *e*.[[key]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}) .
            5. Call CreateOwnDataProperty(*result*, "**1**", Property Descriptor {[[Value]]: *e*.[[value]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}).
        iv. Return *result*.
10. Return Completion {[[type]]: throw, [[value]]: %StopIteration%, [[target]]: empty}.

15.14.7.2.3 *MapIterator*.prototype.@@iterator ( )

The following steps are taken:

1. Return the **this** value.

15.14.7.2.4 *MapIterator*.prototype.@@toStringTag

The initial value of the @@toStringTag property is the string value "**Map Iterator**".

15.14.7.3 Properties of Map Iterator Instances

Map Iterator instances inherit properties from the Map Iterator prototype (the intrinsic, %MapIteratorPrototype%.) Map Iterator instances are initially created with the internal properties described in Table 34.

Table 34 — Internal Data Properties of Map Iterator Instances

Internal Data Property Name	Description
[[Map]]	The Map object that is being iterated.
[[MapNextIndex]]	The integer index of the next Map data element to be examined by this iteration.
[[MapIterationKind]]	A string value that identifies what is to be returned for each element of the iteration. The possible values are: "**key**", "**value**",

	"key+value".

15.15 WeakMap Objects

WeakMap objects are collections of key/value pairs where the keys are ECMAScript objects and values may be arbitrary ECMAScript language values. A WeakMap may be queried to see if it contains an key/value pair with a specific key, but no mechanisms is provided for enumerating the objects it holds as keys. If an object that is being used as the key of a WeakMap key/value pair is only reachable by following a chain of references that start within that WeakMap, then that key/value pair is inaccessible and is automatically removed from the WeakMap. WeakMap implementations must detect and remove such key/value pairs and any associated resources.

An implementation may impose an arbitrarily determined latency between the time a key/value pair of a Weakmap becomes inaccessible and the time when the key/value pair is removed from the Weakmap. If this latency was observable to ECMAScript program, it would be a source of indeterminacy that could impact program execution. For that reason, an ECMAScript implementation must not provide any means to directly test for the presense of any specific key value.

WeakMap objects must be implemented using hash tables or other mechanisms that, on average, provide access times that are sublinear on the number of key/value pairs in the collection. The data structures used in this WeakMap objects specification are only intended to describe the required observable semantics of WeakMap objects. It is not intended to be a viable implementation model.

NOTE WeakMap are intended to provide a mechanism for dynamically associating state with an object in a manner that does not “leak” memory resources if, in the absence of the WeakMap, otherwise became inaccessible and subject to resource reclamation by the implementation’s garbage collection. Achieving this characteristic requires coordination between the WeakMap implementation and the garbage collections. The following references describe mechanism that may be useful to implementations of WeakMap:

> Barry Hayes. 1997. Ephemerons: a new finalization mechanism. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '97)*, A. Michael Berman (Ed.). ACM, New York, NY, USA, 176-183. http://doi.acm.org/10.1145/263698.263733
>
> Alexandra Barros, Roberto Ierusalimschy, Eliminating Cycles in Weak Tables. Journal of Universal Computer Science - J.UCS , vol. 14, no. 21, pp. 3481-3497, 2008. http://www.jucs.org/jucs_14_21/eliminating_cycles_in_weak

15.15.1 Abstract Operations For WeakMap Objects

15.15.1.1 WeakMapInitialisation

The abstract operation WeakMapInitialisation with arguments *obj* and *iterable* is used to initialize an object as a map. It performs the following steps:

1. If Type(*obj*) is not Object, throw a **TypeError** exception.
2. If *obj* already has a [[WeakMapData]] internal data property, throw a **TypeError** exception.
3. If the result of calling the [[GetExtensible]] internal method of *obj* is **false**, throw a **TypeError** exception.
4. If *iterable* is not **undefined**, then
    a. Let *iterable* be ToObject(*iterable*).
    b. ReturnIfAbrupt(*iterable*)
    c. Let *iterator* be the intrinsic symbol @@iterator.
    d. Let *itr* be the result of Invoke(*obj*, *iterator*).
    e. ReturnIfAbrupt(*itr*).
    f. Let *adder* be the result of Get(*obj*, **"set"**).
    g. ReturnIfAbrupt(*adder*).
    h. If IsCallable(*adder*) is **false**, throw a **TypeError** Exception.
5. Add a [[WeakMapData]] internal data property to *obj*.
6. Set *obj*’s [[WeakMapData]] internal data property to a new empty List.

7. If *iterable* is **undefined**, return *obj*.
8. Repeat
    a. Let *next* be the result of Invoke(*itr*, `"next"`).
    b. If IteratorComplete(*next*) is **true**, then return NormalCompletion(*obj*).
    c. Let *next* be ToObject(*next*).
    d. ReturnIfAbrupt(*next*).
    e. Let *k* be the result of Get(*next*, `"0"`).
    f. ReturnIfAbrupt(*k*).
    g. Let *v* be the result of Get(*next*, `"1"`).
    h. ReturnIfAbrupt(*v*).
    i. Let *status* be the result of calling the [[Call]] internal method of *adder* with *obj* as *thisArgument* and a List whose elements are *k* and *v* as *argumentsList*.
    j. ReturnIfAbrupt(*status*).

15.15.2 The WeakMap Constructor Called as a Function

When **`WeakMap`** is called as a function rather than as a constructor, it initializes its **this** value with the internal state necessary to support the **`WeakMap.prototype`** internal methods. This permits super invocation of the **`WeakMap`** constructor by **`WeakKap`** subclasses.

15.15.2.1 WeakMap (iterable = undefined)

1. Let *m* be the **this** value.
2. If *m* is **undefined** or the intrinsic %WeakMapPrototype%
    a. Let *map* be the result of the abstract operation ObjectCreate with the intrinsic %WeakWeakMapPrototype% as the argument.
3. Else
    a. Let *map* be the result of ToObject(*m*).
4. ReturnIfAbrupt(*map*).
5. If *iterable* is not present, let *iterable* be **undefined**.
6. Let *status* be the result of MapInitialisation with *map* and *iterable* as arguments.
7. ReturnIfAbrupt(*status*).
8. Return *map*.

NOTE If the parameter iterable is present, it is expected to be an object that implements an @@iterator method that returns an iterator object that produces two element array-like objects whose first element is a value that will be used as a WeakMap key and whose second element is the value to associate with that key.

15.15.3 The WeakMap Constructor

When **`WeakMap`** is called as part of a **`new`** expression it is a constructor: it initialises the newly created object.

15.15.3.1 new WeakMap (iterable = undefined)

1. Let *map* be the result of the abstract operation ObjectCreate with the intrinsic %WeakMapPrototype% as the argument.
2. If *iterable* is not present, let *iterable* be **undefined**.
3. Let *status* be the result of WeakMapInitialisation with *map* and *iterable* as arguments.
4. ReturnIfAbrupt(*status*).
5. Return *map*.

NOTE If the parameter iterable is present, it is expected to be an object that implements an @@iterator method that returns an iterator object that produces two element array-like objects whose first element is a value that will be used as a WeakMap key and whose second element is the value to associate with that key.

15.15.4 Properties of the WeakMap Constructor

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

Besides the **`length`** property (whose value is **0**), the WeakMap constructor has the following property:

15.15.4.1 WeakMap.prototype

The initial value of **`WeakMap.prototype`** is the WeakMap prototype object (15.15.4).

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

15.15.5 Properties of the WeakMap Prototype Object

The value of the [[Prototype]] internal data property of the WeakMap prototype object is the standard built-in Object prototype object (15.2.4).

15.15.5.1 WeakMap.prototype.constructor

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

15.15.5.2 WeakMap.prototype.clear ()

The following steps are taken:

1. Let *M* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*M*).
3. If *M* does not have a [[WeakMapData]] internal data property throw a **TypeError** exception.
4. Set the value of *M*'s [[WeakMapData]] internal data property to a new empty List.
5. Return **undefined**.

15.15.5.3 WeakMap.prototype.delete (key)

The following steps are taken:

1. Let *M* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*M*).
3. If *M* does not have a [[WeakMapData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *M*'s [[WeakMapData]] internal data property.
5. Repeat for each Record {[[key]], [[value]]} *p* that is an element of *entries*,
    a. If SameValue(*p*.[[key]], *key*), then
        i. Set *p*.[[key]] to empty.
        ii. Set *p*.[[value]] to empty.
        iii. Return **true**.
6. Return **false**.

15.15.5.4 WeakMap.prototype.get (key)

The following steps are taken:

1. Let *M* be the result of calling ToObject with the **this** value the as its argument.
2. ReturnIfAbrupt(*M*).
3. If *M* does not have a [[WeakMapData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *M*'s [[WeakMapData]] internal data property.
5. Repeat for each Record {[[key]], [[value]]} *p* that is an element of *entries*,
    a. If SameValue(*p*.[[key]], *key*), then return *p*.[[value]]
6. Return **undefined**.

15.15.5.5 WeakMap.prototype.has (key)

The following steps are taken:

1. Let *M* be the result of calling ToObject with the **this** value as its argument.

2. ReturnIfAbrupt(*M*).
3. If *M* does not have a [[WeakMapData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *M*'s [[WeakMapData]] internal data property.
5. Repeat for each Record {[[key]], [[value]]} *p* that is an element of *entries*,
    a. If SameValue(*p*.[[key]], *k*), then return **true**.
6. Return **false**.

15.15.5.6 WeakMap.prototype.set (key , value)

The following steps are taken:

1. Let *M* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*M*).
3. If *M* does not have a [[WeakMapData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *M*'s [[WeakMapData]] internal data property.
5. If Type(*key*) is not Object, then throw a **TypeError** exception.
6. Repeat for each Record {[[key]], [[value]]} *p* that is an element of *entries*,
    a. If SameValue(*p*.[[key]], *key*), then
        i. Set *p*.[[value]] to *value*.
        ii. Return **undefined**.
7. Let *p* be the Record {[[key]]: *key*, [[value]]: *value*}
8. Append *p* as the last element of *entries*.
9. Return **undefined**.

15.15.5.7 WeakMap.prototype.@@toStringTag

The initial value of the @@toStringTag property is the string value **"`WeakMap`"**.

15.15.6 Properties of WeakMap Instances

WeakMap instances inherit properties from the WeakMap prototype. After initialisation by the WeakMap constructor, WeakMap instances also have a [[WeakMapData]] internal data property.

15.16 Set Objects

Set objects are collections of arbitrary ECMAScript language values. Each distinct value, as determined using the SameValue algorithm, may only occur once within a Set. A Set object can also iterate its elements in insertion order. Set objects must be implemented using hash tables or other mechanisms that, on average, provide access times that are sublinear on the number of elements in the collection. The data structures used in this Set objects specification is only intended to describe the required observable semantics of Set objects. It is not intended to be a viable implementation model.

15.16.1 Abstract Operations For Set Objects

15.16.1.1 SetInitialisation

The abstract operation SetInitialisation with arguments *obj* and *iterable* is used to initialize an object as a set instance. It performs the following steps:

1. If Type(*obj*) is not Object, throw a **TypeError** exception.
2. If *obj* already has a [[SetData]] internal data property, throw a **TypeError** exception.
3. If the result of calling the [[GetExtensible]] internal method of *obj* is **false**, throw a TypeError exception.
4. If *iterable* is not **undefined**, then
    a. Let *iterable* be ToObject(*iterable*).
    b. ReturnIfAbrupt(*iterable*)
    c. Let *hasValues* be the result of HasProperty(*iterable*, "**`values`**").
    d. ReturnIfAbrupt(*hasValues*).
    e. If *hasValues* is **true**, then
        i. Let *itr* be the result of Invoke(*obj*, **"`values`"**).

f. Else,
   i. Let *iterator* be the @@iterator symbol.
   ii. Let *itr* be the result Invoke(*obj*, *iterator*).
g. ReturnIfAbrupt(*itr*).
h. Let *adder* be the result of Get(*obj*, **"add").**
i. ReturnIfAbrupt(*adder*).
j. If IsCallable(*adder*) is **false**, throw a **TypeError** Exception.

5. Add a [[SetData]] internal data property to *obj*.
6. Set *obj*'s [[SetData]] internal data property to a new empty List.
7. If *iterable* is **undefined**, return *obj*.
8. Repeat
   a. Let *next* be the result of Invoke(*itr*, **"next"**).
   b. If IteratorComplete(*next*) is **true**, then return NormalCompletion(*obj*).
   c. Let *status* be the result of calling the [[Call]] internal method of *adder* with *obj* as *thisArgument* and a List whose sole element is *next* as *argumentsList*.
   d. ReturnIfAbrupt(*status*).

15.16.2 The Set Constructor Called as a Function

When **Set** is called as a function rather than as a constructor, it initializes its **this** value with the internal state necessary to support the **Set.prototype** internal methods. This permits super invocation of the **Set** constructor by **Set** subclasses.

15.16.2.1 Set (iterable = undefined)

1. Let *O* be the **this** value.
2. If *O* is **undefined** or the intrinsic %SetPrototype%
   a. Let *set* be the result of the abstract operation ObjectCreate with the intrinsic %SetPrototype% as the argument.
3. Else
   a. Let *set* be the result of ToObject(*O*).
4. ReturnIfAbrupt(*set*).
5. If *iterable* is not present, let *iterable* be **undefined**.
6. Let *status* be the result of SetInitialisation with *set* and *iterable* as arguments.
7. ReturnIfAbrupt(*status*).
8. Return *set*.

NOTE If the parameter iterable is present, it is expected to be an object that implements an @@iterator method that returns an iterator object that produces two element array-like objects whose first element is a value that will be used as an Map key and whose second element is the value to associate with that key.

15.16.3 The Set Constructor

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

15.16.3.1 new Set (iterable = undefined)

1. Let *set* be the result of the abstract operation ObjectCreate with the intrinsic %SetPrototype% as the argument.
2. If *iterable* is not present, let *iterable* be **undefined**.
3. Let *status* be the result of SetInitialisation with *set* and *iterable* as arguments.
4. ReturnIfAbrupt(*status*).
5. Return *set*.

NOTE If the parameter iterable is present, it is expected to be an object that implements either a **values** method or an @@iterator method. Either method is expected to return an interator object that returns the values that will be the initial elements of the set.

15.16.4 Properties of the Set Constructor

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

Besides the **`length`** property (whose value is **0**), the Set constructor has the following property:

15.16.4.1 Set.prototype

The initial value of **`Set.prototype`** is the intrinsic %SetPrototype% object (15.16.4).

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

15.16.5 Properties of the Set Prototype Object

The value of the [[Prototype]] internal data property of the Set prototype object is the standard built-in Object prototype object (15.2.4).

15.16.5.1 Set.prototype.constructor

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

15.16.5.2 Set.prototype.add (value)

The following steps are taken:

1. Let *S* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*S*).
3. If *S* does not have a [[SetData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *S*'s [[SetData]] internal data property.
5. Repeat for each *p* that is an element of *entries*,
    a. If *p* is not empty and SameValue(*p*, *value*) is **true**, then
        i. Return **undefined**.
6. Append *value* as the last element of *entries*.
7. Return **undefined**.

15.16.5.3 Set.prototype.clear ()

The following steps are taken:

1. Let *S* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*S*).
3. If *S* does not have a [[SetData]] internal data property throw a **TypeError** exception.
4. Set the value of *S*'s [[SetData]] internal data data property to a new empty List.
5. Return **undefined**.

15.16.5.4 Set.prototype.delete (value)

The following steps are taken:

1. Let *S* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*S*).
3. If *S* does not have a [[SetData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *S*'s [[SetData]] internal data property.
5. Repeat for each *e* that is an element of *entries*, in original insertion order
    a. If *e* is not empty and SameValue(*e*, *value*) is **true**, then
        i. Replace the element of *entries* whose value is *e* with an element whose value is empty.
        ii. Return **true**.
6. Return **false**.

15.16.5.5 Set.prototype.forEach (callbackfn , thisArg = undefined)

callbackfn should be a function that accepts three arguments. **`forEach`** calls *callbackfn* once for each value present in the set object, in value insertion order. *callbackfn* is called only for values of the Set which actually exist; it is not called for keys that have been deleted from the set.

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.

NOTE If *callbackfn* is an Arrow Function, **this** was lexically bound when the function was created so *thisArg* will have no effect.

callbackfn is called with three arguments: the first two arguments are a value contained in the Set. The same value of passed for both arguments. The Set object being traversed is passed as the third argument.

NOTE The *callbackfn* is called with three arguments to be consistent with the call back functions used by **`forEach`** methods for Map and Array. For Sets, each item value is considered to be both the key and the value.

`forEach` does not directly mutate the object on which it is called but the object may be mutated by the calls to *callbackfn*.

NOTE Each value is normally visited only once. However, a value will be revisited if it is deleted after it has been visited and then re-added before the to **`forEach`** call completes. Values that are deleted after the call to **`forEach`** begins and before being visited are not visited unless the value is added again before the to **`forEach`** call completes. New values added, after the call to **`forEach`** begins are visited.

When the **`forEach`** method is called with one or two arguments, the following steps are taken:

1. Let *S* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*S*).
3. If *S* does not have a [[SetData]] internal data property throw a **TypeError** exception.
4. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
5. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
6. Let *entries* be the List that is the value of *S*'s [[SetData]] internal data property.
7. Repeat for each *e* that is an element of *entries*, in original insertion order
    a. If *e* is not empty, then
        i. Let *funcResult* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as *thisArgument* and a List containing *e* and *S* as *argumentsList*.
        ii. ReturnIfAbrupt(*funcResult*).
8. Return **undefined**.

The **`length`** property of the **`forEach`** method is **1**.

15.16.5.6 Set.prototype.has (value)

The following steps are taken:

1. Let *S* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*S*).
3. If *S* does not have a [[SetData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *S*'s [[SetData]] internal data property.
5. Repeat for each *e* that is an element of *entries*,
    a. If *e* is not empty and SameValue(*e*, *value*), then return **true**.
6. Return **false**.

15.16.5.7 get Set.prototype.size

`Set.prototype.size` is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps are taken:

1. Let *S* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*S*).
3. If *S* does not have a [[SetData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *S*'s [[SetData]] internal data property.
5. Let *count* be 0.
6. For each *e* that is an element of *entries*
    a. If *e* is not empty then
        i. Set *count* to *count*+1.
7. Return *count*.

15.16.5.8 Set.prototype.values ( )

The following steps are taken:

1. Let *S* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*S*).
3. Return the result of calling the CreateSetIterator abstract operation with argument *S*.

15.16.5.9 Set.prototype.@@iterator ( )

The initial value of the @@iterator property is the same function object as the initial value of the **`values`** property.

15.16.5.10 Set.prototype.@@toStringTag

The initial value of the @@toStringTag property is the string value **`"Set"`**.

15.16.6 Properties of Set Instances

Set instances inherit properties from the Set prototype. After initialisation by the Set constructor, Set instances also have a [[SetData]] internal data property.

15.16.7 Set Iterator Object Structure

A Set Iterator is an ordinary object, with the structure defined below, that represents a specific iteration over some specific Set instance object. There is not a named constructor for Set Iterator objects. Instead, set iterator objects are created by calling certain methods of Set instance objects.

15.16.7.1 CreateSetIterator Abstract Operation

The **value** and @@iterator methods of Set objects return interator objects. The abstract operation CreateSetIterator with argument *set* is used to create and such iterator objects. It performs the following steps:

1. Let *S* be the result of calling ToObject(*set*).
2. ReturnIfAbrupt(*S*).
3. If *S* does not have a [[SetData]] internal data property throw a **TypeError** exception.
4. Let *entries* be the List that is the value of *S*'s [[SetData]] internal data property.
5. Let *itr* be the result of the abstract operation ObjectCreate with the intrinsic object %SetIteratorPrototype% as its argument.
6. Add a [[IteratedSet]] internal data property to *itr* with value *S*.
7. Add a [[SetNextIndex]] internal data property to *itr* with value 0.
8. Return *itr*.

15.16.7.2 The Set Iterator Prototype

All Set Iterator Objects inherit properties from a common Set Iterator Prototype object. The [[Prototype]] internal data property of the Set Iterator Prototype is the %ObjectPrototype% intrinsic object. In addition, the Set Iterator Prototype as the following properties:

15.16.7.2.1 *SetIterator*.prototype.constructor

15.16.7.2.2 *SetIterator*.prototype.next( )

1. Let *O* be the **this** value.
2. If Type(*O*) is not Object, throw a **TypeError** exception.
3. If *O* does not have all of the internal properties of a Set Iterator Instance (15.16.7.1.2), throw a **TypeError** exception.
4. Let *s* be the value of the [[IteratedSet]] internal data property of *O*.
5. Let *index* be the value of the [[SetNextIndex]] internal data property of *O*.
6. Assert: *s* has a [[SetData]] internal data property.
7. Let *entries* be the List that is the value of the [[SetData]] internal data property of *s*.
8. Repeat while *index* is less than the total number of element of *entries*. The number of elements must be redetermined each time this method is evaluated.
    a. Let *e* be the element at 0-origined insertion position *index* of *entries*.
    b. Set *index* to *index*+1;
    c. Set the [[SetNextIndex]] internal data property of *O* to *index*.
    d. If *e* is not empty, then
        i. Return *e*.
9. Return Completion {[[type]]: throw, [[value]]: %StopIteration%, [[target]]: empty}.

15.16.7.2.3 *SetIterator*.prototype.@@iterator ( )

The following steps are taken:

1. Return the **this** value.

15.16.7.2.4 *SetIterator*.prototype.@@toStringTag

The initial value of the @@toStringTag property is the string value **"Set Iterator"**.

15.16.7.3 Properties of Set Iterator Instances

Set Iterator instances inherit properties from the Set Iterator prototype (the intrinsic, %SetIteratorPrototype%.) Set Iterator instances are initially created with the internal properties specified in Table 35.

Table 35 — Internal Data Properties of Set Iterator Instances

Internal Data Property Name	Description
[[IteratedSet]]	The Set object that is being iterated.
[[SetNextIndex]]	The integer index of the next Set data element to be examined by this iteration.

15.17 The Reflect Module

This is a place holder for the material in http://wiki.ecmascript.org/doku.php?id=harmony:reflect_api

15.17.1 Exported Function Properties Reflecting the Essentional Internal Methods

15.17.1.1 Reflect.getPrototypeOf (target)

When the **`getPrototypeOf`** function is called with argument *target* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Return the result of calling the [[GetInheritence]] internal method of *obj*.

15.17.1.2 Reflect.setPrototypeOf (target, proto)

When the **`setPrototypeOf`** function is called with arguments *target* and *propertyKey*, the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. If Type(*proto*) is not Object and *proto* is not **null**, then throw a **TypeError** exception
4. Return the result of calling the [[SetInheritance]] internal method of *obj* with argument *proto*.

15.17.1.3 Reflect.isExtensible (target)

When the **`isExtensible`** function is called with argument *target* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Return the result of calling the [[IsExtensible]] internal method of *obj*.

15.17.1.4 Reflect.preventExtensions (target)

When the **`preventExtensions`** function is called with argument *target*, the following steps are taken:

5. Let *obj* be ToObject(*target*).
6. ReturnIfAbrupt(*obj*).
7. Return the result of calling the [[PreventExtensions]] internal method of *obj*.

15.17.1.5 Reflect.hasOwn (target, propertyKey)

When the **`hasOwn`** function is called with arguments *target* and *propertyKey*, the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Let *key* be ToPropertyKey(*propertyKey*).
4. ReturnIfAbrupt(*key*).
5. Return the result of calling the [[HasOwnProperty]] internal method of *obj* with argument *key*.

15.17.1.6 Reflect.getOwnPropertyDescriptor(target, propertyKey)

When the **`getOwnPropertyDescriptor`** function is called with arguments *target* and *propertyKey*, the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Let *key* be ToPropertyKey(*propertyKey*).
4. ReturnIfAbrupt(*key*).
5. Return the result of calling the [[GetOwnProperty]] internal method of *obj* with argument *key*.

15.17.1.7 Reflect.get (target, propertyKey, receiver=target)

When the **`get`** function is called with arguments *target*, *propertyKey*, and *receiver* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Let *key* be ToPropertyKey(*propertyKey*).
4. ReturnIfAbrupt(*key*).
5. If *receiver* is not present, then
    a. Let *receiver* be *target*.
6. Return the result of calling the [[GetP]] internal method of *obj* with arguments *key*, and *receiver*.

15.17.1.8 Reflect.set (target, propertyKey, V, receiver=target)

When the **`set`** function is called with arguments *target*, *V*, *propertyKey*, and *receiver* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Let *key* be ToPropertyKey(*propertyKey*).
4. ReturnIfAbrupt(*key*).
5. If *receiver* is not present, then
    a. Let *receiver* be *target*.
6. Return the result of calling the [[SetP]] internal method of *obj* with arguments *key*, *V*, and *receiver*.

15.17.1.9 Reflect.deleteProperty (target, propertyKey)

When the **`deleteProperty`** function is called with arguments *target* and *propertyKey*, the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Let *key* be ToPropertyKey(*propertyKey*).
4. ReturnIfAbrupt(*key*).
5. Return the result of calling the [[Delete]] internal method of *obj* with argument *key*.

15.17.1.10 Reflect.defineProperty(target, propertyKey, Attributes)

When the **`defineProperty`** function is called with arguments *target*, *propertyKey*, and *Attributes* the following steps are taken:

7. Let *obj* be ToObject(*target*).
8. ReturnIfAbrupt(*obj*).
9. Let *key* be ToPropertyKey(*propertyKey*).
10. ReturnIfAbrupt(*key*).
11. Let *desc* be the result of calling ToPropertyDescriptor with *Attributes* as the argument.
12. ReturnIfAbrupt(*desc*).
13. Return the result of calling the [[DefineOwnProperty]] internal method of *obj* with arguments *key*, and *desc*.

15.17.1.11 Reflect.enumerate (target)

When the **`enumerate`** function is called with argument *target* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Let *itr* be the result of calling the [[Enumerate]] internal method of *obj*.
4. Return *itr*.

15.17.1.12 Reflect.keys (target)

When the **`keys`** function is called with argument *target* the following steps are taken:

5. Let *obj* be ToObject(*target*).
6. ReturnIfAbrupt(*obj*).
7. Let *keys* be the result of calling the [[Keys]] internal method of *obj*.
8. ReturnIfAbrupt(*keys*).
9. Return CreateArrayFromList(*keys*).

15.17.1.13 Reflect.getOwnPropertyNames (target)

When the **`getOwnPropertyNames`** function is called with argument *target* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Let *keys* be the result of calling the [[OwnPropertyKeys]] internal method of *obj*.
4. ReturnIfAbrupt(*keys*).
5. Return CreateArrayFromList(*keys*).

15.17.1.14 Reflect.freeze (target)

When the **`freeze`** function is called with argument *target* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Return the result of calling the [[Freeze]] internal method of *obj*.

15.17.1.15 Reflect.seal (target)

When the **`seal`** function is called with argument *target* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Return the result of calling the [[Freeze]] internal method of *obj*.

15.17.1.16 Reflect.isFrozen (target)

When the **`isFrozen`** function is called with argument *target* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Return the result of calling the [[IsFrozen]] internal method of *obj*.

15.17.1.17 Reflect.isSealed (target)

When the **`isSealed`** function is called with argument *target* the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Return the result of calling the [[IsSealed]] internal method of *obj*.

15.17.2 Exported Function Properties Derived from the Essentional Internal Methods

15.17.2.1 Reflect.has (target, propertyKey)

When the **`has`** function is called with arguments *target* and *propertyKey*, the following steps are taken:

1. Let *obj* be ToObject(*target*).
2. ReturnIfAbrupt(*obj*).
3. Let *key* be ToPropertyKey(*propertyKey*).
4. ReturnIfAbrupt(*key*).
5. Return the result of HasProperty(*obj*, *key*).

15.17.2.1 Reflect.instanceOf (target, O)

When the **`instanceOf`** function is called with arguments *target* and *O*, the following steps are taken:

1. Return the result of OrdinaryInstanceOf(*target*, *O*).

15.18 Proxy Objects

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 *Script* containing the error. An implementation must report early errors in a *Script* prior to the first evaluation of that *Script*. 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 script 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 script 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.

Annex A
(informative)

Grammar Summary

A.1 Lexical Grammar

SourceCharacter **::** *See clause 6*
any Unicode code unit

InputElementDiv **::** *See clause 7*
WhiteSpace
LineTerminator
Comment
Token
DivPunctuator

InputElementRegExp **::** *See clause 7*
WhiteSpace
LineTerminator
Comment
Token
RegularExpressionLiteral

WhiteSpace **::** *See 7.2*
<TAB>
<VT>
<FF>
<SP>
<NBSP>
<BOM>
<USP>

LineTerminator **::** *See 7.3*
<LF>
<CR>
<LS>
<PS>

LineTerminatorSequence **::** *See 7.3*
<LF>
<CR> [lookahead ∉ <LF>]
<LS>
<PS>
<CR> <LF>

Comment **::** *See 7.4*
MultiLineComment
SingleLineComment

MultiLineComment **::** *See 7.4*
/* *MultiLineCommentChars*$_{opt}$ ***/**

MultiLineCommentChars **::** *See 7.4*
MultiLineNotAsteriskChar MultiLineCommentChars$_{opt}$
***** *PostAsteriskCommentChars*$_{opt}$

PostAsteriskCommentChars **::** *See 7.4*
MultiLineNotForwardSlashOrAsteriskChar MultiLineCommentChars$_{opt}$
***** *PostAsteriskCommentChars*$_{opt}$

MultiLineNotAsteriskChar **::** *See 7.4*
SourceCharacter **but not ***

MultiLineNotForwardSlashOrAsteriskChar **::** *See 7.4*
SourceCharacter **but not one of / or ***

SingleLineComment **::** *See 7.4*
// *SingleLineCommentChars*$_{opt}$

SingleLineCommentChars **::** *See 7.4*
SingleLineCommentChar SingleLineCommentChars$_{opt}$

SingleLineCommentChar **::** *See 7.4*
SourceCharacter **but not** *LineTerminator*

Token **::** *See 7.5*
IdentifierName
Punctuator
NumericLiteral
StringLiteral

Identifier **::** *See 7.6*
IdentifierName **but not** *ReservedWord*

IdentifierName **::** *See 7.6*
IdentifierStart
IdentifierName IdentifierPart

IdentifierStart **::** *See 7.6*
UnicodeLetter
$
_
**** *UnicodeEscapeSequence*

IdentifierPart **::** *See 7.6*

IdentifierStart
UnicodeCombiningMark
UnicodeDigit
UnicodeConnectorPunctuation
<ZWNJ>
<ZWJ>

UnicodeLetter **::** *See 7.6*

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

UnicodeCombiningMark **::** *See 7.6*

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

UnicodeDigit **::** *See 7.6*

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

UnicodeConnectorPunctuation **::** *See 7.6*

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

ReservedWord **::** *See 7.6.1*

Keyword
FutureReservedWord
NullLiteral
BooleanLiteral

Keyword **:: one of** *See 7.6.1.1*

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**	

FutureReservedWord **:: one of** *See 7.6.1.2*

class	**enum**	**extends**	**super**
const	**export**	**import**	

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			

Punctuator **:: one of** *See 7.7*

`{`	`}`	`(`	`)`	`[`	`]`
`.`	`;`	`,`	`<`	`>`	`<=`
`>=`	`==`	`!=`	`===`	`!==`	
`+`	`-`	`*`	`%`	`++`	`--`
`<<`	`>>`	`>>>`	`&`	`|`	`^`
`!`	`~`	`&&`	`||`	`?`	`:`
`=`	`+=`	`-=`	`*=`	`%=`	`<<=`
`>>=`	`>>>=`	`&=`	`|=`	`^=`	

DivPunctuator **:: one of** *See 7.7*

`/` `/=`

Literal **::** *See 7.8*

NullLiteral
BooleanLiteral
NumericLiteral
StringLiteral
RegularExpressionLiteral

NullLiteral **::** *See 7.8.1*

`null`

BooleanLiteral **::** *See 7.8.2*

`true`
`false`

NumericLiteral **::** *See 7.8.3*

DecimalLiteral
HexIntegerLiteral

DecimalLiteral **::** *See 7.8.3*

DecimalIntegerLiteral `.` *DecimalDigits*$_{opt}$ *ExponentPart*$_{opt}$
`.` *DecimalDigits ExponentPart*$_{opt}$
DecimalIntegerLiteral ExponentPart$_{opt}$

DecimalIntegerLiteral **::** *See 7.8.3*

`0`
NonZeroDigit DecimalDigits$_{opt}$

DecimalDigits **::** *See 7.8.3*

DecimalDigit
DecimalDigits DecimalDigit

DecimalDigit **:: one of** *See 7.8.3*

`0` `1` `2` `3` `4` `5` `6` `7` `8` `9`

NonZeroDigit **:: one of** *See 7.8.3*
`1 2 3 4 5 6 7 8 9`

ExponentPart **::** *See 7.8.3*
ExponentIndicator SignedInteger

ExponentIndicator **:: one of** *See 7.8.3*
`e E`

SignedInteger **::** *See 7.8.3*
DecimalDigits
+ *DecimalDigits*
– *DecimalDigits*

HexIntegerLiteral **::** *See 7.8.3*
0x *HexDigit*
0X *HexDigit*
HexIntegerLiteral HexDigit

HexDigit **:: one of** *See 7.8.3*
`0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`

StringLiteral **::** *See 7.8.4*
`"` *DoubleStringCharacters*$_{opt}$ `"`
`'` *SingleStringCharacters*$_{opt}$ `'`

DoubleStringCharacters **::** *See 7.8.4*
DoubleStringCharacter DoubleStringCharacters$_{opt}$

SingleStringCharacters **::** *See 7.8.4*
SingleStringCharacter SingleStringCharacters$_{opt}$

DoubleStringCharacter **::** *See 7.8.4*
SourceCharacter **but not one of** `"` **or** **** **or** *LineTerminator*
**** *EscapeSequence*
LineContinuation

SingleStringCharacter **::** *See 7.8.4*
SourceCharacter **but not one of** `'` **or** **** **or** *LineTerminator*
**** *EscapeSequence*
LineContinuation

LineContinuation **::** *See 7.8.4*
**** *LineTerminatorSequence*

EscapeSequence **::** *See 7.8.4*
CharacterEscapeSequence
0 [lookahead ∉ *DecimalDigit*]
HexEscapeSequence
UnicodeEscapeSequence

CharacterEscapeSequence **::** *See 7.8.4*
SingleEscapeCharacter
NonEscapeCharacter

SingleEscapeCharacter **:: one of** *See 7.8.4*
`' " \ b f n r t v`

NonEscapeCharacter **::** *See 7.8.4*
SourceCharacter **but not one of** *EscapeCharacter* **or** *LineTerminator*

EscapeCharacter **::** *See 7.8.4*
SingleEscapeCharacter
DecimalDigit
x
u

HexEscapeSequence **::** *See 7.8.4*
x *HexDigit HexDigit*

UnicodeEscapeSequence **::** *See 7.8.4*
u *HexDigit HexDigit HexDigit HexDigit*

RegularExpressionLiteral **::** *See 7.8.5*
/ *RegularExpressionBody* **/** *RegularExpressionFlags*

RegularExpressionBody **::** *See 7.8.5*
RegularExpressionFirstChar RegularExpressionChars

RegularExpressionChars **::** *See 7.8.5*
[empty]
RegularExpressionChars RegularExpressionChar

RegularExpressionFirstChar **::** *See 7.8.5*
RegularExpressionNonTerminator **but not one of * or \ or / or [**
RegularExpressionBackslashSequence
RegularExpressionClass

RegularExpressionChar **::** *See 7.8.5*
RegularExpressionNonTerminator **but not \ or / or [**
RegularExpressionBackslashSequence
RegularExpressionClass

RegularExpressionBackslashSequence **::** *See 7.8.5*
**** *RegularExpressionNonTerminator*

RegularExpressionNonTerminator **::** *See 7.8.5*
SourceCharacter **but not** *LineTerminator*

RegularExpressionClass **::** *See 7.8.5*
[*RegularExpressionClassChars* **]**

RegularExpressionClassChars **::** *See 7.8.5*
[empty]
RegularExpressionClassChars RegularExpressionClassChar

RegularExpressionClassChar **::** *See 7.8.5*
RegularExpressionNonTerminator **but not] or **
RegularExpressionBackslashSequence

RegularExpressionFlags **::** *See 7.8.5*
[empty]
RegularExpressionFlags IdentifierPart

A.2 Number Conversions

StringNumericLiteral **:::** *See 9.1.3.1*
StrWhiteSpace$_{opt}$
StrWhiteSpace$_{opt}$ *StrNumericLiteral StrWhiteSpace*$_{opt}$

StrWhiteSpace **:::** *See 9.1.3.1*
StrWhiteSpaceChar StrWhiteSpace$_{opt}$

StrWhiteSpaceChar **:::** *See 9.1.3.1*
WhiteSpace
LineTerminator

StrNumericLiteral **:::** *See 9.1.3.1*
StrDecimalLiteral
HexIntegerLiteral

StrDecimalLiteral **:::** *See 9.1.3.1*
StrUnsignedDecimalLiteral
+ *StrUnsignedDecimalLiteral*
- *StrUnsignedDecimalLiteral*

StrUnsignedDecimalLiteral **:::** *See 9.1.3.1*
`Infinity`
DecimalDigits **.** *DecimalDigits*$_{opt}$ *ExponentPart*$_{opt}$
. *DecimalDigits ExponentPart*$_{opt}$
DecimalDigits ExponentPart$_{opt}$

DecimalDigits **:::** *See 9.1.3.1*
DecimalDigit
DecimalDigits DecimalDigit

DecimalDigit **::: one of** *See 9.1.3.1*
`0 1 2 3 4 5 6 7 8 9`

ExponentPart **:::** *See 9.1.3.1*
ExponentIndicator SignedInteger

ExponentIndicator **::: one of** *See 9.1.3.1*
`e E`

SignedInteger **:::** *See 9.1.3.1*
DecimalDigits
+ *DecimalDigits*
- *DecimalDigits*

HexIntegerLiteral **:::** *See 9.1.3.1*

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

HexDigit **::: one of** *See 9.1.3.1*

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

A.3 Expressions

PrimaryExpression **:** *See 11.1*

this
Identifier
Literal
ArrayLiteral
ObjectLiteral
(*Expression* **)**

ArrayLiteral **:** *See 11.1.4*

[$Elision_{opt}$ **]**
[*ElementList* **]**
[*ElementList* **,** $Elision_{opt}$ **]**

ElementList **:** *See 11.1.4*

$Elision_{opt}$ *AssignmentExpression*
ElementList **,** $Elision_{opt}$ *AssignmentExpression*

Elision **:** *See 11.1.4*

,
Elision **,**

ObjectLiteral **:** *See 11.1.5*

{ }
{ *PropertyDefinitionList* **}**
{ *PropertyDefinitionList* **, }**

PropertyDefinitionList **:** *See 11.1.5*

PropertyDefinition
PropertyDefinitionList **,** *PropertyDefinition*

PropertyDefinition **:** *See 11.1.5*

PropertyName **:** *AssignmentExpression*
get *PropertyName* **( ) {** *FunctionBody* **}**
set *PropertyName* **(** *PropertySetParameterList* **) {** *FunctionBody* **}**

PropertyName **:** *See 11.1.5*

IdentifierName
StringLiteral
NumericLiteral

PropertySetParameterList **:** *See 11.1.5*

Identifier

MemberExpression **:** *See 11.2*

PrimaryExpression
FunctionExpression
MemberExpression **[** *Expression* **]**
MemberExpression **.** *IdentifierName*
new *MemberExpression Arguments*

NewExpression **:** *See 11.2*

MemberExpression
new *NewExpression*

CallExpression **:** *See 11.2*

MemberExpression Arguments
CallExpression Arguments
CallExpression **[** *Expression* **]**
CallExpression **.** *IdentifierName*

Arguments **:** *See 11.2*

( )
(*ArgumentList* **)**

ArgumentList **:** *See 11.2*

AssignmentExpression
ArgumentList **,** *AssignmentExpression*

LeftHandSideExpression **:** *See 11.2*

NewExpression
CallExpression

PostfixExpression **:** *See 11.3*

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

UnaryExpression **:** *See 11.4*

PostfixExpression
delete *UnaryExpression*
void *UnaryExpression*
typeof *UnaryExpression*
++ *UnaryExpression*
-- *UnaryExpression*
+ *UnaryExpression*
- *UnaryExpression*
~ *UnaryExpression*
! *UnaryExpression*

MultiplicativeExpression **:** *See 11.5*

UnaryExpression
MultiplicativeExpression ***** *UnaryExpression*
MultiplicativeExpression **/** *UnaryExpression*
MultiplicativeExpression **%** *UnaryExpression*

AdditiveExpression **:** *See 11.6*

MultiplicativeExpression
AdditiveExpression **+** *MultiplicativeExpression*
AdditiveExpression **–** *MultiplicativeExpression*

ShiftExpression **:** *See 11.7**

AdditiveExpression
ShiftExpression **<<** *AdditiveExpression*
ShiftExpression **>>** *AdditiveExpression*
ShiftExpression **>>>** *AdditiveExpression*

RelationalExpression **:** *See 11.8*

ShiftExpression
RelationalExpression **<** *ShiftExpression*
RelationalExpression **>** *ShiftExpression*
RelationalExpression **<=** *ShiftExpression*
RelationalExpression **>=** *ShiftExpression*
RelationalExpression **instanceof** *ShiftExpression*
RelationalExpression **in** *ShiftExpression*

RelationalExpressionNoIn **:** *See 11.8*

ShiftExpression
RelationalExpressionNoIn **<** *ShiftExpression*
RelationalExpressionNoIn **>** *ShiftExpression*
RelationalExpressionNoIn **<=** *ShiftExpression*
RelationalExpressionNoIn **>=** *ShiftExpression*
RelationalExpressionNoIn **instanceof** *ShiftExpression*

EqualityExpression **:** *See 11.9*

RelationalExpression
EqualityExpression **==** *RelationalExpression*
EqualityExpression **!=** *RelationalExpression*
EqualityExpression **===** *RelationalExpression*
EqualityExpression **!==** *RelationalExpression*

EqualityExpressionNoIn **:** *See 11.9*

RelationalExpressionNoIn
EqualityExpressionNoIn **==** *RelationalExpressionNoIn*
EqualityExpressionNoIn **!=** *RelationalExpressionNoIn*
EqualityExpressionNoIn **===** *RelationalExpressionNoIn*
EqualityExpressionNoIn **!==** *RelationalExpressionNoIn*

BitwiseANDExpression **:** *See 11.10*

EqualityExpression
BitwiseANDExpression **&** *EqualityExpression*

BitwiseANDExpressionNoIn **:** *See 11.10*

EqualityExpressionNoIn
BitwiseANDExpressionNoIn **&** *EqualityExpressionNoIn*

BitwiseXORExpression **:** *See 11.10*
BitwiseANDExpression
BitwiseXORExpression **^** *BitwiseANDExpression*

BitwiseXORExpressionNoIn **:** *See 11.10*
BitwiseANDExpressionNoIn
BitwiseXORExpressionNoIn **^** *BitwiseANDExpressionNoIn*

BitwiseORExpression **:** *See 11.10*
BitwiseXORExpression
BitwiseORExpression **|** *BitwiseXORExpression*

BitwiseORExpressionNoIn **:** *See 11.10*
BitwiseXORExpressionNoIn
BitwiseORExpressionNoIn **|** *BitwiseXORExpressionNoIn*

LogicalANDExpression **:** *See 11.11*
BitwiseORExpression
LogicalANDExpression **&&** *BitwiseORExpression*

LogicalANDExpressionNoIn **:** *See 11.11*
BitwiseORExpressionNoIn
LogicalANDExpressionNoIn **&&** *BitwiseORExpressionNoIn*

LogicalORExpression **:** *See 11.11*
LogicalANDExpression
LogicalORExpression **||** *LogicalANDExpression*

LogicalORExpressionNoIn **:** *See 11.11*
LogicalANDExpressionNoIn
LogicalORExpressionNoIn **||** *LogicalANDExpressionNoIn*

ConditionalExpression **:** *See 11.12*
LogicalORExpression
LogicalORExpression **?** *AssignmentExpression* **:** *AssignmentExpression*

ConditionalExpressionNoIn **:** *See 11.12*
LogicalORExpressionNoIn
LogicalORExpressionNoIn **?** *AssignmentExpression* **:** *AssignmentExpressionNoIn*

AssignmentExpression **:** *See 11.13*
ConditionalExpression
LeftHandSideExpression **=** *AssignmentExpression*
LeftHandSideExpression AssignmentOperator AssignmentExpression

AssignmentExpressionNoIn **:** *See 11.13*
ConditionalExpressionNoIn
LeftHandSideExpression **=** *AssignmentExpressionNoIn*
LeftHandSideExpression AssignmentOperator AssignmentExpressionNoIn

AssignmentOperator : **one of** *See 11.13*

***=** **/=** **%=** **+=** **-=** **<<=** **>>=** **>>>=** **&=** **^=** **|=**

Expression : *See 11.14*

AssignmentExpression
Expression **,** *AssignmentExpression*

ExpressionNoIn : *See 11.14*

AssignmentExpressionNoIn
ExpressionNoIn **,** *AssignmentExpressionNoIn*

A.4 Statements

Statement : *See clause 12*

Block
VariableStatement
EmptyStatement
ExpressionStatement
IfStatement
IterationStatement
ContinueStatement
BreakStatement
ReturnStatement
WithStatement
LabelledStatement
SwitchStatement
ThrowStatement
TryStatement
DebuggerStatement

Block : *See 12.1*

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

StatementList : *See 12.1*

Statement
StatementList Statement

VariableStatement : *See 12.2*

var *VariableDeclarationList* **;**

VariableDeclarationList : *See 12.2*

VariableDeclaration
VariableDeclarationList **,** *VariableDeclaration*

VariableDeclarationListNoIn : *See 12.2*

VariableDeclarationNoIn
VariableDeclarationListNoIn **,** *VariableDeclarationNoIn*

VariableDeclaration : *See 12.2*

Identifier Initialiser$_{opt}$

VariableDeclarationNoIn : *See 12.2*

Identifier InitialiserNoIn$_{opt}$

Initialiser **:** *See 12.2*
= *AssignmentExpression*

InitialiserNoIn **:** *See 12.2*
= *AssignmentExpressionNoIn*

EmptyStatement **:** *See 12.3*
;

ExpressionStatement **:** *See 12.4*
[lookahead ∉ {**{**, **function**}] *Expression* **;**

IfStatement **:** *See 12.5*
if (*Expression* **)** *Statement* **else** *Statement*
if (*Expression* **)** *Statement*

IterationStatement **:** *See 12.6*
do *Statement* **while (** *Expression* **);**
while (*Expression* **)** *Statement*
for (*ExpressionNoIn*$_{opt}$**;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*
for (var *VariableDeclarationListNoIn***;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*
for (*LeftHandSideExpression* **in** *Expression* **)** *Statement*
for (var *VariableDeclarationNoIn* **in** *Expression* **)** *Statement*

ContinueStatement **:** *See 12.7*
continue ;
continue [no *LineTerminator* here] *Identifier* **;**

BreakStatement **:** *See 12.8*
break ;
break [no *LineTerminator* here] *Identifier* **;**

ReturnStatement **:** *See 12.9*
return ;
return [no *LineTerminator* here] *Expression* **;**

WithStatement **:** *See 12.10*
with (*Expression* **)** *Statement*

SwitchStatement **:** *See 12.11*
switch (*Expression* **)** *CaseBlock*

CaseBlock **:** *See 12.11*
{ *CaseClauses*$_{opt}$ **}**
{ *CaseClauses*$_{opt}$ *DefaultClause CaseClauses*$_{opt}$ **}**

CaseClauses **:** *See 12.11*
CaseClause
CaseClauses CaseClause

CaseClause **:** *See 12.11*
case *Expression* **:** *StatementList*$_{opt}$

DefaultClause **:** *See 12.11*
default : *StatementList*$_{opt}$

LabelledStatement **:** *See 12.12*
Identifier **:** *Statement*

ThrowStatement **:** *See 12.13*
throw [no *LineTerminator* here] *Expression* **;**

TryStatement **:** *See 12.14*
try *Block Catch*
try *Block Finally*
try *Block Catch Finally*

Catch **:** *See 12.14*
catch (*Identifier* **)** *Block*

Finally **:** *See 12.14*
finally *Block*

DebuggerStatement **:** *See 12.15*
debugger **;**

A.5 Functions and Scripts

FunctionDeclaration **:** *See clause 13*
function *Identifier* **(** *FormalParameterList*$_{opt}$ **) {** *FunctionBody* **}**

FunctionExpression **:** *See clause 13*
function *Identifier*$_{opt}$ **(** *FormalParameterList*$_{opt}$ **) {** *FunctionBody* **}**

FormalParameterList **:** *See clause 13*
Identifier
FormalParameterList **,** *Identifier*

FunctionBody **:** *See clause 13*
SourceElements$_{opt}$

Program **:** *See clause 14*
SourceElements$_{opt}$

SourceElements **:** *See clause 14*
SourceElement
SourceElements SourceElement

SourceElement **:** *See clause 14*
Statement
FunctionDeclaration

A.6 Universal Resource Identifier Character Classes

uri **:::** *See 15.1.3*
uriCharacters$_{opt}$

uriCharacters **:::** *See 15.1.3*
uriCharacter uriCharacters$_{opt}$

uriCharacter **:::** *See 15.1.3*
uriReserved
uriUnescaped
uriEscaped

uriReserved **::: one of** *See 15.1.3*
`;` `/` `?` `:` `@` `&` `=` `+` `$` `,`

uriUnescaped **:::** *See 15.1.3*
uriAlpha
DecimalDigit
uriMark

uriEscaped **:::** *See 15.1.3*
`%` *HexDigit HexDigit*

uriAlpha **::: one of** *See 15.1.3*
`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** *See 15.1.3*
`-` `_` `.` `!` `~` `*` `'` `(` `)`

A.7 Regular Expressions

Pattern **::** *See 15.10.1*
Disjunction

Disjunction **::** *See 15.10.1*
Alternative
Alternative `|` *Disjunction*

Alternative **::** *See 15.10.1*
[empty]
Alternative Term

Term **::** *See 15.10.1*
Assertion
Atom
Atom Quantifier

Assertion **::** *See 15.10.1*

^
$
\ b
\ B
(? = *Disjunction* **)**
(? ! *Disjunction* **)**

Quantifier **::** *See 15.10.1*

QuantifierPrefix
QuantifierPrefix **?**

QuantifierPrefix **::** *See 15.10.1*

*****
+
?
{ *DecimalDigits* **}**
{ *DecimalDigits* **, }**
{ *DecimalDigits* **,** *DecimalDigits* **}**

Atom **::** *See 15.10.1*

PatternCharacter
.
**** *AtomEscape*
CharacterClass
(*Disjunction* **)**
(? : *Disjunction* **)**

PatternCharacter :: *See 15.10.1*

SourceCharacter **but not one of-**
^ $ \ . * + ? ( ) [ ] { } |

AtomEscape **::** *See 15.10.1*

DecimalEscape
CharacterEscape
CharacterClassEscape

CharacterEscape **::** *See 15.10.1*

ControlEscape
c *ControlLetter*
HexEscapeSequence
UnicodeEscapeSequence
IdentityEscape

ControlEscape **:: one of** *See 15.10.1*

f n r t v

ControlLetter **:: one of** *See 15.10.1*

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 **::** *See 15.10.1*
SourceCharacter **but not** *IdentifierPart*
<ZWJ>
<ZWNJ>

DecimalEscape **::** *See 15.10.1*
DecimalIntegerLiteral [lookahead ∉ *DecimalDigit*]

CharacterClassEscape **:: one of** *See 15.10.1*
d D s S w W

CharacterClass **::** *See 15.10.1*
[[lookahead ∉ {**^**}] *ClassRanges* **]**
[^ *ClassRanges* **]**

ClassRanges **::** *See 15.10.1*
[empty]
NonemptyClassRanges

NonemptyClassRanges **::** *See 15.10.1*
ClassAtom
ClassAtom NonemptyClassRangesNoDash
ClassAtom **–** *ClassAtom ClassRanges*

NonemptyClassRangesNoDash **::** *See 15.10.1*
ClassAtom
ClassAtomNoDash NonemptyClassRangesNoDash
ClassAtomNoDash **–** *ClassAtom ClassRanges*

ClassAtom **::** *See 15.10.1*
-
ClassAtomNoDash

ClassAtomNoDash **::** *See 15.10.1*
SourceCharacter **but not one of \ or] or -**
**** *ClassEscape*

ClassEscape **::** *See 15.10.1*
DecimalEscape
b
CharacterEscape
CharacterClassEscape

A.8 JSON

A.8.1 JSON Lexical Grammar

JSONWhiteSpace **::** *See 15.12.1.1*
<TAB>
<CR>

<LF>
<SP>

JSONString **::** *See 15.12.1.1*
" *JSONStringCharacters*$_{opt}$ **"**

JSONStringCharacters **::** *See 15.12.1.1*
JSONStringCharacter JSONStringCharacters$_{opt}$

JSONStringCharacter **::** *See 15.12.1.1*
SourceCharacter **but not one of " or \ or** *U+0000* **through** *U+001F*
**** *JSONEscapeSequence*

JSONEscapeSequence **::** *See 15.12.1.1*
JSONEscapeCharacter
UnicodeEscapeSequence

JSONEscapeCharacter **:: one of** *See 15.12.1.1*
" / \ b f n r t

JSONNumber **::** *See 15.12.1.1*
-$_{opt}$ *DecimalIntegerLiteral JSONFraction*$_{opt}$ *ExponentPart*$_{opt}$

JSONFraction **::** *See 15.12.1.1*
. *DecimalDigits*

JSONNullLiteral **::** *See 15.12.1.1*
NullLiteral

JSONBooleanLiteral **::** *See 15.12.1.1*
BooleanLiteral

A.8.2 JSON Syntactic Grammar

JSONText **:** *See 15.12.1.2*
JSONValue

JSONValue **:** *See 15.12.1.2*
JSONNullLiteral
JSONBooleanLiteral
JSONObject
JSONArray
JSONString
JSONNumber

JSONObject **:** *See 15.12.1.2*
{ }
{ *JSONMemberList* **}**

JSONMember **:** *See 15.12.1.2*
JSONString **:** *JSONValue*

JSONMemberList **:** *See 15.12.1.2*
JSONMember
JSONMemberList **,** *JSONMember*

JSONArray **:** *See 15.12.1.2*
[ ]
[*JSONElementList* **]**

JSONElementList **:** *See 15.12.1.2*

JSONValue

JSONElementList **,** *JSONValue*

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

Static 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**

FourToSeven :: **one of**
4 5 6 7

Static Semantics

- The CV of *EscapeSequence* :: *OctalEscapeSequence* is the CV of the *OctalEscapeSequence*.
- The CV of *OctalEscapeSequence* :: *OctalDigit* [lookahead ∉ *DecimalDigit*] is the character whose code unit value is the MV of the *OctalDigit*.
- The CV of *OctalEscapeSequence* :: *ZeroToThree OctalDigit* [lookahead ∉ *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 additional properties of the standard built-in objects are defined.

B.2.1 Additional Properties of the Global Object

B.2.1.1 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 **%u***xxxx* 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 **"%u***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.2.1.2 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 Result(1).
3. Let *R* be the empty String.
4. Let *k* be 0.
5. If *k* equals Result(2), return *R*.
6. Let *c* be the character at position *k* within Result(1).
7. If *c* is not `%`, go to step 18.
8. If *k* is greater than Result(2)–6, go to step 14.
9. If the character at position *k*+1 within 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 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 Result(1).
12. Increase k by 5.
13. Go to step 18.
14. If *k* is greater than Result(2)–3, go to step 18.
15. If the two characters at positions *k*+1 and *k*+2 within 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 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.2.2 Additional Properties of the String.prototype Object

B.2.2.1 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 (*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. ReturnIfAbrupt(CheckObjectCoercible(**this** value)).
2. Let *S* be the result of performing ToString, giving it the **this** value as its argument.
3. Let *intStart* be ToInteger(*start*).
4. ReturnIfAbrupt(*intStart*).
5. If *length* is **undefined**, let *end* be +∞; otherwise let *end* be ToInteger(*length*).
6. ReturnIfAbrupt(*end*).

7. Let *size* be the number of characters in *S*.
8. If *intStart* is negative, then let *intStart* be max(*size* + *intStart*,0).
9. Let *resultLength* be min(max(*end*,0), *size* – *intStart*).
10. If *resultLength* ≤ 0, return the empty String **""**.
11. Return a String containing *resultLength* consecutive characters from *S* beginning with the character at position *intStart*.

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.2.2.2 String.prototype.anchor (name)

When the **anchor** method is called with argument *name*, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, **"a"**, **"name"** and *name*.

The abstract operation CreateHTML is called with arguments *string*, *tag*, *attribute*, and *value*. The arguments *tag* and *attribute* must be string values. The following steps are taken:

1. ReturnIfAbrupt(CheckObjectCoercible(*string*)).
2. Let *S* be the result of performing ToString(*string*).
3. ReturnIfAbrupt(*S*).
4. Let *p1* be the string value that is the concatenation of **"<"** and *tag*.
5. If *attribute* is not the empty String, then
    a. Let *V* be the result of performing ToString(*value*).
    b. ReturnIfAbrupt(*V*).
    c. Let *escapedV* be the string value that is the same as *V* except that each occurrence of the character **"** (code unit value 0x0022) in *V* has been replaced with the six character sequence **"""**.
    d. Let *p1* be the string value that is the concatenation of the following string values:
        - *p1*
        - a single space code unit 0x0020
        - *attribute*
        - **"="**
        - **'"'**
        - *escapedV*
        - **'"'**
6. Let *p2* be the string value that is the concatenation of *p1* and **">"**.
7. Let *p3* be the string value that is the concatenation of *p2*, **"</"**, *tag*, and **">"**.
8. Return *p3*.

B.2.2.3 String.prototype.big ()

When the **big** method is called with no arguments, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, **"big"**, **""** and **""**.

B.2.2.4 String.prototype.blink ()

When the **blink** method is called with no arguments, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, **"blink"**, **""** and **""**.

B.2.2.5 String.prototype.bold ()

When the **bold** method is called with no arguments, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"b"`, `""` and `""`.

B.2.2.6 String.prototype.fixed ()

When the **fixed** method is called with no arguments, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"tt"`, `""` and `""`.

B.2.2.7 String.prototype.fontcolor (color)

When the **fontcolor** method is called with argument *color*, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"font"`, `"color"` and *color*.

B.2.2.8 String.prototype.fontsize (size)

When the **fontsize** method is called with argument *size*, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"font"`, `"size"` and *size*.

B.2.2.9 String.prototype.italics ()

When the **italics** method is called with no arguments, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"i"`, `""` and `""`.

B.2.2.10 String.prototype.link (url)

When the **link** method is called with argument *url*, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"a"`, `"href"` and *url*.

B.2.2.11 String.prototype.small ()

When the **small** method is called with no arguments, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, `"small"`, `""` and `""`.

B.2.2.12 String.prototype.strike ()

When the **strike** method is called with no arguments, the following steps are taken:

1. Let *S* be the **this** value.

2. Return the result of performing the abstract operation CreateHTML with arguments *S*, **"strike"**, **""** and **""**.

B.2.2.13 String.prototype.sub ()

When the **sub** method is called with no arguments, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, **"sub"**, **""** and **""**.

B.2.2.14 String.prototype.sup ()

When the **sup** method is called with no arguments, the following steps are taken:

1. Let *S* be the **this** value.
2. Return the result of performing the abstract operation CreateHTML with arguments *S*, **"sup"**, **""** and **""**.

B.2.3 Additional Properties of the Date.prototype Object

B.2.3.1 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. ReturnIfAbrupt(*t*).
3. If *t* is **NaN**, return **NaN**.
4. Return YearFromTime(LocalTime(*t*)) − 1900.

B.2.3.2 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. Let *y* be ToNumber(*year*).
3. If *y* is **NaN**, set the [[DateValue]] internal data property of this Date object to **NaN** and return **NaN**.
4. If *y* is not **NaN** and 0 ≤ ToInteger(*y*) ≤ 99 then let *yyyy* be ToInteger(*y*) + 1900. Otherwise, let *yyyy* be *y*.
5. Let *d* be MakeDay(*yyyy*, MonthFromTime(*t*), DateFromTime(*t*)).
6. Let *date* be UTC(MakeDate(*d*, TimeWithinDay(*t*))).
7. Set the [[DateValue]] internal data property of this Date object to TimeClip(*date*).
8. Return the value of the [[DateValue]] internal data property of this Date object.

B.2.3.3 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** }.

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 [[GetP]] internal method given in 8.12.3 is replaced with the following:

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 data 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 data 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 the abstract operation ObjectCreate.
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 data property of *obj* to *v*.
    d. Return *obj*.
4. Call the [[DefineOwnProperty]] internal method of *obj* with arguments *propId*.name, *propId*.descriptor, and **false**.
5. Return *obj*.

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 data property of *obj* to *v*.
    d. Return *obj*.
5. Call the [[DefineOwnProperty]] internal method of *obj* with arguments *propId*.name, *propId*.descriptor, and **false**.
6. Return *obj*.

Annex C
(informative)

The Strict Mode of ECMAScript

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 data 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).

- 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, 13.6, 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.

Annex D
(informative)
Corrections and Clarifications with Possible Compatibility Impact

In Edition 6

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".

15.9.5.2: Previous editions did not specify the value returned by Date.prototype.toString when this time value is NaN. The 6th Edition species the result to be the String value is **"Invalid Date"**

In 5.1 Edition 5.1

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 3rd 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.l.ii and 3.l.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 *JA* 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.

In 5th Edition 5

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.

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.

15.10.6: RegExp.prototype is now a RegExp object rather than an instance of Object. The value of its [[Class]] internal data property which is observable using Object.prototype.toString is now "RegExp" rather than "Object".

Annex E
(informative)

Additions and Changes that Introduce Incompatibilities with Prior Editions

In the 6th Edition

12.6: In Edition 6, a terminating semi-colon is no longer required at the end of a do-while statement.

12.14: In Edition 6, it is an early error for a *Catch* clause to contained a **`var`** declaration for the same *Identifier* that appears as the *Catch* clause parameter. In previous editions, such a variable declaration would be instantiated in the enclosing variable environment but the declaration's *Initializer* value would be assigned to the *Catch* parameter.

13.3 In Edition 6, the function objects that are created as the values of the [[Get]] or [[Set]] attribute of accessor properties in an *ObjectLiteral* are not constructor functions. In Edition 5, they were constructors.

15.2.3.5 and 15.2.3.7: In Edition 6, all property additions and changes are processed, even if one of them throws an acception. If an exception occurs during such processing, the first such exception is thrown after all propertie are processed. In Edition 5, processing of property additions and changes immediately terminated when the first exception occurred.

In the 5th 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 data 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 [[StringData] 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.

Annex F
(informative)

Static Semantic Rule Cross Reference

Routine Name	Purpose	Definitions	Uses
BoundNames	Produces a list of the Identifiers bound by a production. Does not include Identifiers that are bound within inner environments associated with the production.	12.2.1, 12.2.2, 12.2.4, 12.6.4, 13.1, 13.2, 13.5	
ConstructorMethod	From a *ClassBody* return the first *ClassElement* whose PropName is **"constructor"**. Returns empty if the *ClassBody* does not contain one.	13.5	
Contains	Determine if a grammar production either directly or indirectly includes a grammar symbol.	5.3, 13.1, 13.2, 13.5	
CoveredFormalsList	Reparse a covered *Expression* using *FormalsList* as the goal symbol.	13.2	
CV	Determines the "character value" of a component of a *StringLiteral*.	7.8.4	
Elision Width	Determine the number of commas in an *Elision*.	11.1.4.1	
ExpectedArgumentCount	Determine the "length" of an argument list for the purpose of initializing the "length" property of a function object.	13.1, 13.2, 13.3	
HasInitialiser	Determines whether the production contains an *Initialiser* production.	12.2.4, 13.1	
IsConstantDeclaration	Determines whether the production introduces a immutable environment record binding	12.2, 13.1, 13.5	
IsInvalidAssignmentPattern	Determines if a *LeftHandSideExpression* is a valid assignment target. Primarily for dealing with destructuring assignment targets.	11.2	
LexicalDeclarations	Return a List containing the components of a production that are processed as lexical declarations	12.1, 12.11, 12.5	
LexicallyDeclaredNames	Returns a list of the lexically scoped identifiers declared by a production.	12.1, 13.1, 13.2, 13.5	
MethodDefinitions	Return a list of the *MethodDefinition* productions that are part of a *ClassElementList*.	13.5	
MV	Determines the "mathematical value" of a numeric	7.8.3	

	lirteral or component of a numeric literal.		
PropName	Determines the string value of the property name referenced by a production.	11.1.5.1, 13.3, 13.5	
PropNameList	Returns a List of the string values of the property names referenced by a production. The list reflects the order of the references in the source text. The list may contain duplicate elements.	11.5.1, 13.5	
ReferencesSuper	Determine if a *MethodDefinition* contains any references to the *ReservedWord* **`super`**.	13.3	
SpecialMethod	Determine if a *MethodDefinition* defines a generator method or an accessor property.	13.3	
SV	Determines the "string value" of a *StringLiteral* or component of a *StringLiteral*.	7.8.4	
VarDeclaredNames	Returns a list of the local top-level scoped identifiers declared by a production. These are identifier that are scoped as if by a var statement.	12.1, 12.5, 12.6.1, 12.6.2, 12.6.3, 12.6.4, 12.12, 13.1, 13.5	

Scrap Heap

A place to temporarily hand on to stuff that's been deleted

MemberExpression **:**
MemberExpression **<|** *TriangleLiteral*

TriangleLiteral **:**
SealedArrayLiteral
SealedObjectLiteral
FunctionExpression
ArrowFunction
ValueLiteral

CallExpression **:**
CallExpression **<|** *TriangleLiteral*

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**.

15.5.4.25 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 abstract operation ArrayCreate 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.

Static Semantics: TopLevelLexicallyDeclaredNames

OuterStatementList **:** *OuterStatementList OuterItem*

1. Let *names* be TopLevelLexicallyDeclaredNames of *OuterStatementList*.
2. Append to *names* the elements of the TopLevelLexicallyDeclaredNames of *OuterItem*.
3. Return *names*.

OuterItem **:** *StatementListItem*

1. Return a new empty List.

StatementListItem **:** *Declaration*

1. If *Declaration* is *Declaration* **:** *FunctionDeclaration*, then return a new empty List.
2. Return the BoundNames of *Declaration*.

8.3.10 [[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.

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 data 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*.[[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*.

This follow version places function body declarations in scope of parameter initializers

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 declarated 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*.
5. Let *varDeclarations* be the VarScopedDeclarations of *code*.
6. Let *functionsToInitialize* be an emptyList.
7. Let *argumentsObjectNotNeeded* be **false**.
8. For each *d* in *varDeclarations*, in reverse list order do
    a. If *d* is a *FunctionDeclaration* then
        i. NOTE If there are multiple *FunctionDeclarations* for the same name, the last declaration is used.
        ii. Let *fn* be the sole element of the BoundNames of *d*.
        iii. If *fn* is "**`arguments`**", then let *argumentsObjectNotNeeded* be **true**.
        iv. Let *alreadyDeclared* be the result of calling *env's* HasBinding concrete method passing *fn* as the argument.
        v. If *alreadyDeclared* is **false**, then
            1. Let *status* be the result of calling *env's* CreateMutableBinding concrete method passing *fn* as the argument.
            2. Assert: *status* is never an Abrupt Completion.
            3. Append *d* to *functionsToInitialize*.
9. For each String *paramName* in *parameterNames*, do
    a. Let *alreadyDeclared* be the result of calling *env's* HasBinding concrete method passing *paramName* as the argument.
    b. NOTE Duplicate parameter names can only occur in non-strict functions. Parameter names that are the same as function declaration names do not get initialized to **undefined**.
    c. If *alreadyDeclared* is **false**, then
        i. If *paramName* is "**`arguments`**", then let *argumentsObjectNotNeeded* be **true**.
        ii. Let *status* be the result of calling *env's* CreateMutableBinding concrete method passing *paramName* as the argument.
        iii. Assert: *status* is never an Abrupt Completion
        iv. Call *env*'s InitializeBinding concrete method passing *paramName*, and **undefined** as the arguments.
10. NOTE If there is a function declaration or formal parameter with the name **`"arguments"`** then an argument object is not created.
11. If *argumentsObjectNotNeeded* 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.
12. Let *varNames* be the VarDeclaredNames of *code*.
13. 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 initialied here if it is not also the name of a formal parameter or a *FunctionDeclarations*.

c. If *alreadyDeclared* is **false**, then
   i. Call *env's* CreateMutableBinding concrete method passing *varName* as the argument.

14. Let *lexDeclarations* be the LexicalDeclarations of *code*.
15. For each element *d* in *lexDeclarations* do
    a. NOTE A lexically declared name can not be the same as a function declaration, formal parameter, or a var name. Lexically declarated names are only instantiated here but not initialized.
    b. 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.
16. For each *FunctionDeclaration f* in *functionsToInitialize*, do
    a. Let *fn* be the sole element of the BoundNames of *f*.
    b. Let *fo* be the result of performing InstantiateFunctionObject for *f* with argument *env*.
    c. Call *env*'s SetMutableMinding concrete method passing *fn*, *fo*, and **false** as the arguments.
17. NOTE Function declaration are initialised prior to parameter initialisation so that default value expressions may reference them. it is not extended code. "**`arguments`**" is not initialized until after parameter initialization.
18. Let *ao* be the result of InstantiateArgumentsObject with argument *argumentsList.*
19. NOTE If *argumentsObjectNotNeeded* 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.
20. If *argumentsObjectNotNeeded* is **false**, then
    a. If *strict* is **true**, then
       i. Perform the abstract operation CompleteStrictArgumentsObject with argument *a0*.
    b. Else,
       i. Perform the abstract operation CompleteMappedArgumentsObject with arguments *a0, func*, *formals*, and *env*.
    c. Call *env*'s InitializeBinding concrete method passing "**`arguments`**" and *ao* as arguments.
21. Let *formalStatus* be the result of performing Binding Initialisation for *formals* with *ao* and **undefined** as arguments.
22. ReturnIfAbrupt(*formalStatus*).
23. Return NormalCompletion(empty).

Table 36 — Internal Properties Only Defined for Some Objects

Internal Property	*Value Type Domain*	*Description*
[[BuiltinBrand]]	The BuiltinBrand enumeration.	A tag value used by this specification to categorize various kinds of ECMAScript objects defined in this specification.
[[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]].
[[Scope]]	Lexical Environment	A lexical environment that is the environment in which a Function object is executed. Of the standard built-in ECMAScript objects, only Function objects implement [[Scope]].
[[FormalParameters]]	Parse Tree	A parse tree for ECMAScript code parsed with *FormalParameterList* as the goal symbol. Of the standard built-in ECMAScript objects, only Function objects implement [[FormalParameters]].
[[Code]]	Parse Tree	A parse tree for ECMAScript code parsed with *FunctionBody* as the goal symbol. Of the standard built-in ECMAScript objects, only Function objects implement [[Code]].
[[Strict]]	Boolean	**true** if a Function object is a strict mode function. Of the standard built-in ECMAScript objects, only Function objects implement [[Strict]].
[[BoundTargetFunction]]	Object	The target function of a function object created using the standard built-in Function.prototype.bind method. Only ECMAScript objects created using Function.prototype.bind have a [[BoundTargetFunction]] internal property.
[[BoundThis]]	*any*	The pre-bound this value of a function Object created using the standard built-in Function.prototype.bind method. Only ECMAScript objects created using Function.prototype.bind have a [[BoundThis]] internal property.
[[BoundArguments]]	List of *any*	The pre-bound argument values of a function Object created by the standard built-in Function.prototype.bind method. Only objects created by Function.prototype.bind have a [[BoundArguments]] internal property.
[[Match]]	SpecOp(*String*, *index*) → *MatchResult*	Tests for a regular expression match and returns a MatchResult value (see 15.10.2.1). Of the standard built-in ECMAScript objects, only RegExp 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.

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>