ECMAScript Language Specification

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

Component: choose an appropriate one
Version: Rev 21, November 8, 2013 Draft
Contents

Introduction ............................................................................................................................... vii
1 Scope ................................................................................................................................... 1
2 Conformance ......................................................................................................................... 1
3 Normative references .......................................................................................................... 1
4 Overview ............................................................................................................................... 1
  4.1 Web Scripting .................................................................................................................. 2
  4.2 ECMAScript Overview .................................................................................................... 3
  4.2.1 Objects ....................................................................................................................... 3
  4.2.2 The Strict Variant of ECMAScript .............................................................................. 34
  4.3 Terms and definitions ...................................................................................................... 5
  4.4 Organization of This Specification ................................................................................ 8
5 Notational Conventions ...................................................................................................... 9
  5.1 Syntactic and Lexical Grammars ................................................................................... 9
  5.1.1 Context-Free Grammars .......................................................................................... 9
  5.1.2 The Lexical and RegExp Grammars ........................................................................... 9
  5.1.3 The Numeric String Grammar ................................................................................. 9
  5.1.4 The Syntactic Grammar .......................................................................................... 9
  5.1.5 Grammar Notation .................................................................................................. 10
  5.2 Algorithm Conventions ................................................................................................. 15
  5.3 Static Semantic Rules ..................................................................................................... 16
6 ECMAScript Data Types and Values ................................................................................ 17
  6.1 ECMAScript Language Types ....................................................................................... 17
  6.1.1 The Undefined Type .................................................................................................. 17
  6.1.2 The Null Type .......................................................................................................... 17
  6.1.3 The Boolean Type ..................................................................................................... 17
  6.1.4 The String Type ....................................................................................................... 17
  6.1.5 The Symbol Type ..................................................................................................... 18
  6.1.6 The Number Type .................................................................................................... 19
  6.1.7 The Object Type ....................................................................................................... 20
  6.2 ECMAScript Specification Types .................................................................................. 22
  6.2.1 The List and Record Specification Type .................................................................... 22
  6.2.2 The Completion Record Specification Type .............................................................. 31
  6.2.3 The Reference Specification Type ............................................................................ 31
  6.2.4 The Property Descriptor Specification Type .............................................................. 32
  6.2.5 The Lexical Environment and Environment Record Specification Types .............. 36
  6.2.6 Data Blocks ............................................................................................................. 37
7 Abstract Operations ............................................................................................................ 37
  7.1 Type Conversion and Testing ....................................................................................... 37
  7.1.1 ToPrimitive ............................................................................................................. 37
  7.1.2 ToBoolean .............................................................................................................. 39
  7.1.3 ToNumber .............................................................................................................. 39
  7.1.4 ToInt32: (Signed 32 Bit Integer) ............................................................................... 42
  7.1.5 ToInt32: (Unsigned 32 Bit Integer) ......................................................................... 43
  7.1.6 ToUint32: (Signed 32 Bit Integer) .......................................................................... 43
  7.1.7 ToInt16: (Signed 16 Bit Integer) ............................................................................. 43
  7.1.8 ToUint16: (Unsigned 16 Bit Integer) ...................................................................... 43
  7.1.9 ToInt8: (Signed 8 Bit Integer) ............................................................................... 43
  7.1.10 ToUint8: (Unsigned 8 Bit Integer) ....................................................................... 44
  7.1.11 ToUint8Clamp: (Unsigned 8 Bit Integer, Clamped) .............................................. 44

© Ecma International 2013
12.5.3 Runtime Semantics: Evaluation ........................................................................................................ 159
12.5 Additive Operators .................................................................................................................................. 160
12.6.2 Static Semantics: IsValidSimpleAssignmentTarget ........................................................................ 161
12.6.3 The Addition operator (+) .................................................................................................................. 161
12.6.4 The Subtraction Operator (−) ............................................................................................................ 161
12.6.5 Applying the Additive Operators to Numbers .................................................................................... 162
12.7 Bitwise Shift Operators ........................................................................................................................ 162
12.7.1 Static Semantics: IsAnonymousFunctionDefinition ...................................................................... 162
12.7.2 Static Semantics: IsValidSimpleAssignmentTarget ........................................................................ 163
12.7.3 The Left Shift Operator (<<) .............................................................................................................. 163
12.7.4 The Signed Right Shift Operator (>> ) ............................................................................................... 163
12.7.5 The Unsigned Right Shift Operator (>>> ) .......................................................................................... 164
12.8 Relational Operators .............................................................................................................................. 164
12.8.1 Static Semantics: IsAnonymousFunctionDefinition ...................................................................... 164
12.8.2 Static Semantics: IsValidSimpleAssignmentTarget ........................................................................ 165
12.8.3 Runtime Semantics: Evaluation ......................................................................................................... 165
12.8.4 Runtime Semantics: InstanceofOperator(O, C) ................................................................................ 166
12.9 Equality Operators .................................................................................................................................... 166
12.9.1 Static Semantics: IsAnonymousFunctionDefinition ...................................................................... 167
12.9.2 Static Semantics: IsValidSimpleAssignmentTarget ........................................................................ 167
12.9.3 Runtime Semantics: Evaluation ......................................................................................................... 167
12.10 Binary Bitwise Operators ...................................................................................................................... 168
12.10.1 Static Semantics: IsAnonymousFunctionDefinition ...................................................................... 169
12.10.2 Static Semantics: IsValidSimpleAssignmentTarget ........................................................................ 169
12.10.3 Runtime Semantics: Evaluation ....................................................................................................... 169
12.11 Binary Logical Operators .................................................................................................................... 169
12.11.1 Static Semantics: IsAnonymousFunctionDefinition ...................................................................... 169
12.11.2 Static Semantics: IsValidSimpleAssignmentTarget ........................................................................ 170
12.11.3 Runtime Semantics: Evaluation ....................................................................................................... 170
12.12 Conditional Operator (?:) .................................................................................................................... 170
12.12.1 Static Semantics: IsAnonymousFunctionDefinition ...................................................................... 170
12.12.2 Static Semantics: IsValidSimpleAssignmentTarget ........................................................................ 171
12.12.3 Runtime Semantics: Evaluation ....................................................................................................... 171
12.13 Assignment Operators ........................................................................................................................ 171
12.13.1 Static Semantics: Early Errors ........................................................................................................... 171
12.13.2 Static Semantics: IsAnonymousFunctionDefinition ...................................................................... 172
12.13.3 Static Semantics: IsValidSimpleAssignmentTarget ........................................................................ 172
12.13.4 Runtime Semantics: Evaluation ....................................................................................................... 172
12.13.5 Destructuring Assignment ................................................................................................................ 173
12.14 Comma Operator (, ) .......................................................................................................................... 178
12.14.2 Static Semantics: IsValidSimpleAssignmentTarget ........................................................................ 178
12.14.3 Runtime Semantics: Evaluation ....................................................................................................... 178

13 ECMAScript Language: Statements and Declarations .............................................................................. 179
13.0 Statement Semantics .................................................................................................................................. 179
13.0.1 Static Semantics: VarDeclaredNames .............................................................................................. 179
13.0.2 Runtime Semantics: LabelledEvaluation ......................................................................................... 179
13.0.3 Runtime Semantics: Evaluation ......................................................................................................... 180
13.1 Block ...................................................................................................................................................... 180
13.1.1 Static Semantics: Early Errors ........................................................................................................... 180
13.1.2 Static Semantics: LexicalDeclarations .............................................................................................. 180
13.1.3 Static Semantics: LexicallyDeclaredNames ..................................................................................... 181
13.1.4 Static Semantics: TopLevelLexicallyDeclaredNames ....................................................................... 181
13.1.5 Static Semantics: TopLevelLexicallyScopedDeclarations ................................................................. 181
13.1.6 Static Semantics: TopLevelVarDeclaredNames ................................................................................ 182
13.1.7 Static Semantics: TopLevelVarScopedDeclarations ......................................................................... 182
13.1.8 Static Semantics: VarDeclaredNames .............................................................................................. 182
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.1.9</td>
<td>Static Semantics: VarScopedDeclarations</td>
</tr>
<tr>
<td>13.1.10</td>
<td>Runtime Semantics: Evaluation</td>
</tr>
<tr>
<td>13.1.11</td>
<td>Runtime Semantics: Block Declaration Instantiation</td>
</tr>
<tr>
<td>13.2</td>
<td>Declarations and the Variable Statement</td>
</tr>
<tr>
<td>13.2.1</td>
<td>Let and Const Declarations</td>
</tr>
<tr>
<td>13.2.2</td>
<td>Variable Statement</td>
</tr>
<tr>
<td>13.2.3</td>
<td>Destructuring Binding Patterns</td>
</tr>
<tr>
<td>13.3</td>
<td>Empty Statement</td>
</tr>
<tr>
<td>13.3.1</td>
<td>Runtime Semantics: Evaluation</td>
</tr>
<tr>
<td>13.4</td>
<td>Expression Statement</td>
</tr>
<tr>
<td>13.5</td>
<td>The if Statement</td>
</tr>
<tr>
<td>13.5.1</td>
<td>Static Semantics: VarDeclaredNames</td>
</tr>
<tr>
<td>13.5.2</td>
<td>Runtime Semantics: Evaluation</td>
</tr>
<tr>
<td>13.6</td>
<td>Iteration Statements</td>
</tr>
<tr>
<td>13.6.0</td>
<td>Semantics</td>
</tr>
<tr>
<td>13.6.1</td>
<td>The do-while Statement</td>
</tr>
<tr>
<td>13.6.2</td>
<td>The while Statement</td>
</tr>
<tr>
<td>13.6.3</td>
<td>The for Statement</td>
</tr>
<tr>
<td>13.6.4</td>
<td>The for-in and for-of Statements</td>
</tr>
<tr>
<td>13.7</td>
<td>The continue Statement</td>
</tr>
<tr>
<td>13.7.1</td>
<td>Static Semantics: Early Errors</td>
</tr>
<tr>
<td>13.7.2</td>
<td>Runtime Semantics: Evaluation</td>
</tr>
<tr>
<td>13.8</td>
<td>The break Statement</td>
</tr>
<tr>
<td>13.8.1</td>
<td>Static Semantics: Early Errors</td>
</tr>
<tr>
<td>13.8.2</td>
<td>Runtime Semantics: Evaluation</td>
</tr>
<tr>
<td>13.9</td>
<td>The return Statement</td>
</tr>
<tr>
<td>13.9.1</td>
<td>Runtime Semantics: Evaluation</td>
</tr>
<tr>
<td>13.10</td>
<td>The with Statement</td>
</tr>
<tr>
<td>13.10.1</td>
<td>Static Semantics: Early Errors</td>
</tr>
<tr>
<td>13.10.2</td>
<td>Static Semantics: VarDeclaredNames</td>
</tr>
<tr>
<td>13.10.3</td>
<td>Runtime Semantics: Evaluation</td>
</tr>
<tr>
<td>13.11</td>
<td>The switch Statement</td>
</tr>
<tr>
<td>13.11.1</td>
<td>Static Semantics: Early Errors</td>
</tr>
<tr>
<td>13.11.2</td>
<td>Static Semantics: LexicallyDeclaredNames</td>
</tr>
<tr>
<td>13.11.3</td>
<td>Static Semantics: VarDeclaredNames</td>
</tr>
<tr>
<td>13.11.4</td>
<td>Static Semantics: VarDeclaredNames</td>
</tr>
<tr>
<td>13.11.6</td>
<td>Runtime Semantics: CaseSelectorEvaluation</td>
</tr>
<tr>
<td>13.11.7</td>
<td>Runtime Semantics: Evaluation</td>
</tr>
<tr>
<td>13.12</td>
<td>Labelled Statements</td>
</tr>
<tr>
<td>13.12.1</td>
<td>Static Semantics: Early Errors</td>
</tr>
<tr>
<td>13.12.2</td>
<td>Static Semantics: VarDeclaredNames</td>
</tr>
<tr>
<td>13.12.3</td>
<td>Runtime Semantics: LabelledEvaluation</td>
</tr>
<tr>
<td>13.12.4</td>
<td>Runtime Semantics: Evaluation</td>
</tr>
<tr>
<td>13.13</td>
<td>The throw Statement</td>
</tr>
<tr>
<td>13.14</td>
<td>The try Statement</td>
</tr>
<tr>
<td>13.14.1</td>
<td>Static Semantics: Early Errors</td>
</tr>
<tr>
<td>13.14.2</td>
<td>Static Semantics: VarDeclaredNames</td>
</tr>
<tr>
<td>13.14.3</td>
<td>Runtime Semantics: BindingInitialisation</td>
</tr>
<tr>
<td>13.14.4</td>
<td>Runtime Semantics: CatchClauseEvaluation</td>
</tr>
<tr>
<td>13.14.5</td>
<td>Runtime Semantics: Evaluation</td>
</tr>
<tr>
<td>13.15</td>
<td>The debugger: statement</td>
</tr>
<tr>
<td>13.15.1</td>
<td>Runtime Semantics: Evaluation</td>
</tr>
<tr>
<td>14</td>
<td>ECMAScript Language: Functions and Classes</td>
</tr>
<tr>
<td>14.1</td>
<td>Function Definitions</td>
</tr>
<tr>
<td>14.1.1</td>
<td>Static Semantics: Early Errors</td>
</tr>
</tbody>
</table>
14.1.2 Static Semantics: BoundNames

14.1.3 Static Semantics: Contains

14.1.4 Static Semantics: ExpectedArgumentCount

14.1.5 Static Semantics: HasInitialiser

14.1.6 Static Semantics: IsAnonymousFunctionDefinition

14.1.7 Static Semantics: IsConstantDeclaration

14.1.8 Static Semantics: IsSimpleParameterList

14.1.9 Static Semantics: IsStrict

14.1.10 Static Semantics: LexicallyDeclaredNames

14.1.11 Static Semantics: VarDeclaredNames

14.1.12 Runtime Semantics: EvaluateBody

14.1.13 Runtime Semantics: IteratorBindingInitialisation

14.1.14 Runtime Semantics: InstantiateFunctionObject

14.1.15 Runtime Semantics: Evaluation

14.2 Arrow Function Definitions

14.2.1 Static Semantics: Early Errors

14.2.2 Static Semantics: BoundNames

14.2.3 Static Semantics: Contains

14.2.4 Static Semantics: CoveredFormalsList

14.2.5 Static Semantics: ExpectedArgumentCount

14.2.6 Static Semantics: IsSimpleParameterList

14.2.7 Static Semantics: LexicallyDeclaredNames

14.2.8 Runtime Semantics: IteratorBindingInitialisation

14.2.9 Runtime Semantics: EvaluateBody

14.2.10 Runtime Semantics: Evaluation

14.3 Method Definitions

14.3.1 Static Semantics: Early Errors

14.3.2 Static Semantics: ExpectedArgumentCount

14.3.3 Static Semantics: HasComputedPropertyKey

14.3.4 Static Semantics: IsSimpleParameterList

14.3.5 Static Semantics: PropName

14.3.6 Static Semantics: ReferencesSuper

14.3.7 Static Semantics: SpecialMethod

14.3.8 Runtime Semantics: DefineMethod

14.3.9 Runtime Semantics: PropertyDefinitionEvaluation

14.4 Generator Function Definitions

14.4.1 Static Semantics: Early Errors

14.4.2 Static Semantics: BoundNames

14.4.3 Static Semantics: Contains

14.4.4 Static Semantics: HasComputedPropertyKey

14.4.5 Static Semantics: IsAnonymousFunctionDefinition

14.4.6 Static Semantics: IsConstantDeclaration

14.4.7 Static Semantics: LexicallyDeclaredNames

14.4.8 Static Semantics: PropName

14.4.9 Static Semantics: ReferencesSuper

14.4.10 Static Semantics: VarDeclaredNames

14.4.11 Runtime Semantics: EvaluateBody

14.4.12 Runtime Semantics: InstantiateFunctionObject

14.4.13 Runtime Semantics: PropertyDefinitionEvaluation

14.4.14 Runtime Semantics: Evaluation

14.5 Class Definitions

14.5.1 Static Semantics: Early Errors

14.5.2 Static Semantics: BoundNames

14.5.3 Static Semantics: ConstructorMethod

14.5.4 Static Semantics: Contains

14.5.5 Static Semantics: IsAnonymousFunctionDefinition

14.5.6 Static Semantics: IsConstantDeclaration

14.5.7 Static Semantics: IsStatic

14.5.8 Static Semantics: LexicallyDeclaredNames
20.3 Date Objects........................................................................................................... 312
  20.3.1 Overview of Date Objects and Definitions of Abstract Operations........................................... 312
  20.3.2 The Date Constructor.......................................................................................... 317
  20.3.3 Properties of the Date Constructor ........................................................................ 319
  20.3.4 Properties of the Date Prototype Object ...................................................................... 320
  20.3.5 Properties of Date Instances ................................................................................ 329

21 Text Processing......................................................................................................... 329
  21.1 String Objects ...................................................................................................... 329
  21.1.1 The String Constructor ....................................................................................... 329
  21.1.2 Properties of the String Constructor ................................................................. 330
  21.1.3 Properties of the String Prototype Object ....................................................... 332
  21.1.4 Properties of String Instances .......................................................................... 345
  21.1.5 String Iterator Objects ..................................................................................... 345
  21.2 RegExp (Regular Expression) Objects .................................................................... 346
  21.2.1 Patterns ........................................................................................................... 347
  21.2.2 Pattern Semantics ............................................................................................ 349
  21.2.3 The RegExp Constructor .................................................................................. 362
  21.2.4 Properties of the RegExp Constructor .............................................................. 364
  21.2.5 Properties of the RegExp Prototype Object .................................................... 365
  21.2.6 Properties of RegExp Instances ...................................................................... 372

22 Indexed Collections.................................................................................................. 372
  22.1 Array Objects ...................................................................................................... 372
  22.2.1 The Array Constructor ....................................................................................... 372
  22.2.2 Properties of the Array Constructor ................................................................. 374
  22.2.3 Properties of the Array Prototype Object ....................................................... 376
  22.2.4 Properties of Array Instances .......................................................................... 401
  22.2.5 Array Iterator Objects ...................................................................................... 401
  22.2.6 TypedArray Objects ........................................................................................ 403
  22.2.7 The %TypedArray% Intrinsic Object ................................................................ 403
  22.2.8 Properties of the %TypedArray% Intrinsic Object .......................................... 407
  22.2.9 Properties of the %TypedArrayPrototype% Object ........................................ 409
  22.2.10 The TypedArray Constructor ......................................................................... 419
  22.2.11 Properties of the TypedArray Constructors ................................................... 420
  22.2.12 Properties of TypedArray Prototype Objects ................................................ 420
  22.2.13 Properties of TypedArray Instances .............................................................. 420

23 Keyed Collections.................................................................................................... 421
  23.1 Map Objects........................................................................................................ 421
  23.1.1 The Map Constructor ....................................................................................... 421
  23.1.2 Properties of the Map Constructor ..................................................................... 422
  23.1.3 Properties of the Map Prototype Object ........................................................... 422
  23.1.4 Properties of Map Instances ............................................................................ 426
  23.1.5 Map Iterator Objects ........................................................................................ 426
  23.2 Set Objects .......................................................................................................... 427
  23.2.1 The Set Constructor ........................................................................................ 428
  23.2.2 Properties of the Set Constructor ..................................................................... 429
  23.2.3 Properties of the Set Prototype Object .............................................................. 429
  23.2.4 Properties of Set Instances .............................................................................. 432
  23.2.5 Set Iterator Objects .......................................................................................... 432
  23.3 WeakMap Objects ............................................................................................... 434
  23.3.1 The WeakMap Constructor ............................................................................. 434
  23.3.2 Properties of the WeakMap Constructor .......................................................... 435
  23.3.3 Properties of the WeakMap Prototype Object .................................................. 436
  23.3.4 Properties of WeakMap Instances .................................................................... 437
  23.4 WeakSet Objects .................................................................................................. 437
  23.4.1 The WeakSet Constructor .............................................................................. 438
  23.4.2 Properties of the WeakSet Constructor ............................................................. 439
  23.4.3 Properties of the WeakSet Prototype Object .................................................... 439
  23.4.4 Properties of WeakSet Instances .................................................................... 441
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.


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 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 ECMAScript must interpret characters in conformance with the Unicode Standard, Version 5.1.0 or later and ISO/IEC 10646. If the adopted ISO/IEC 10646-I subset is not otherwise specified, it is presumed to be the Unicode set, collection 10646.

A conforming implementation of ECMAScript that provides an application programming interface that supports programs that need to adapt to the linguistic and cultural conventions used by different human languages and countries must implement the interface defined by the most recent edition of ECMA-402 that is compatible with this specification.

A conforming implementation of ECMAScript may provide additional types, values, objects, properties, and functions beyond those described in this specification. In particular, a conforming implementation of ECMAScript may 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 may support program and regular expression syntax not described in this specification. In particular, a conforming implementation of ECMAScript may support program syntax that makes use of the "future reserved words" listed in subclause 11.6.2.2 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.


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.

ECMA-402, ECMAScript Internationalization API Specification.
http://www.ecma-international.org/publications/standards/Ecma-402.htm

ECMA-404, The JSON Data Interchange Format.

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 C, Java, Self, and Scheme as described in:


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 ECMAScript 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. In ECMAScript, an **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`, `Symbol` 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 **prototype** of the object) 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.
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:

**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:

**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:

**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:

**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:

**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:

**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:

**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:

**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:

**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:

**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:

**Figure 1 — Object/Prototype Relationships**
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 6 of this specification

4.3.2 primitive value
member of one of the types Undefined, Null, Boolean, Number, Symbol, or String as defined in clause 6
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 essential internal methods that must be supported by all objects.

4.3.7 exotic object
object that has some alternative behaviour for one or more of the essential internal methods that must be
supported by all 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 slot 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 slot whose value is the String value. A String object can be coerced to a String value by calling the String constructor as a function (21.1.1).

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 slot whose value is the Number value. A Number object can be coerced to a Number value by calling the Number constructor as a function (20.1.1).

4.3.23
Infinity
number value that is the positive infinite Number value

4.3.24
NaN
number value that is an IEEE 754 "Not-a-Number" value

4.3.25
Symbol value
primitive value that represents a unique, non-String Object property key.

4.3.26
Symbol type
set of all possible Symbol values

4.3.27
Symbol object
member of the Object type that is an instance of the standard built-in symbol constructor

4.3.28
function
member of the Object type that may be invoked as a subroutine

NOTE In addition to its 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.
4.3.29

**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.

4.3.30

**property**

association between a key and a value that is a part of an object. The key be either a String value or a Symbol value.

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.

4.3.31

**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.

4.3.32

**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.

4.3.33

**attribute**

internal value that defines some characteristic of a property

4.3.34

**own property**

property that is directly contained by its object

4.3.35

**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

4.4  **Organization of This Specification**

The remainder of this specification is organized as follows:

Clause 5 defines the notational conventions used throughout the specification.

Clauses 6-9 define the execution environment within which ECMAScript programs operate.

Clauses 109.5.15-16 define the actual ECMAScript programming language including its syntactic encoding and the execution semantics of all language features.

Clauses 17-26 define the ECMAScript standard library. It includes the definitions of all of the standard objects that are available for use by ECMAScript programs as they execute.
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 11. This grammar has as its terminal symbols characters (Unicode code units) that conform to the rules for SourceCharacter defined in clause 10.1. 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 (11.9). 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 21.2.1. 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 7.1.3.1.

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 12, 13, 14 and 15 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 generalised productions that permit token sequences that are not valid ECMAScript scripts. 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 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 either exactly as written or using equivalent Unicode escape sequences (see 10.19.5.15). All terminal symbol characters specified in this way are to be understood as the appropriate Unicode code points from the Basic Latin 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 : BindingIdentifier Initialiseropt

is a convenient abbreviation for:

VariableDeclaration :

BindingIdentifier

BindingIdentifier Initialiser

and that:

IterationStatement :

for ( LexicalDeclaration ; Expressionopt ; Expressionopt ) Statement

is a convenient abbreviation for:

IterationStatement :

for ( LexicalDeclaration ; Expressionopt ) Statement

for ( LexicalDeclaration ; Expression ; Expressionopt ) Statement

which in turn is an abbreviation for:

IterationStatement :

for ( LexicalDeclaration ; ) Statement

for ( LexicalDeclaration ; Expression ) Statement

for ( LexicalDeclaration ; Expression ; ) Statement

so, in this example, the nonterminal IterationStatement actually has four alternative right-hand sides.

A production may be parameterised by a subscripted annotation of the form "parameters!", which may appear as a suffix to the nonterminal symbol defined by the production. "parameters!" may be either a single name or a comma separated list of names. A parameterised production is a shorthand for a set of productions defining all combinations of the parameter names appended to the parameterised nonterminal symbol. This means that:

StatementListopt : ReturnStatement ExpressionStatement

is a convenient abbreviation for:

StatementList :

ReturnStatement ExpressionStatement

StatementListReturn : ReturnStatement ExpressionStatement

and that:

StatementListopt , n : ReturnStatement ExpressionStatement

is abbreviation for:
References to nonterminals on the right hand side of a production can also be parameterised. For example:

StatementList : ReturnStatement ExpressionStatement

is equivalent to saying:

StatementList : ReturnStatement ExpressionStatementIn

A nonterminal reference may have both a parameter list and an "opt" suffix. For example:

VariableDeclaration : Binding Identifier Initialiser

is an abbreviation for:

VariableDeclaration : Binding Identifier InitialiserIn

Prefixing a parameter name with "?" on a right hand side nonterminal reference makes that parameter value dependent upon the occurrence of the parameter name on the reference to the current production's left hand side symbol. For example:

VariableDeclarationIn : BindingIdentifier Initialiser

is an abbreviation for:

VariableDeclaration : BindingIdentifier Initialiser

VariableDeclarationIn : BindingIdentifier InitialiserIn

If a right hand side alternative is prefixed with "[parameter]" that alternative is only available if the named parameter was used in referencing the production's nonterminal symbol. If a right hand side alternative is
prefixed with "~parameter" that alternative is only available if the named parameter was not used in referencing the production's nonterminal symbol. This means that:

```
StatementListReturn : ~return ReturnStatement
ExpressionStatement
```

is an abbreviation for:

```
StatementList :
ExpressionStatement
```

```
StatementListReturn :
ReturnStatement
ExpressionStatement
```

and that

```
StatementListReturn :
~return ReturnStatement
ExpressionStatement
```

is an abbreviation for:

```
StatementList :
ReturnStatement
ExpressionStatement
```

```
StatementListReturn :
ExpressionStatement
```

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
\[\text{DecimalDigit} :: \text{one of} \ 0 1 2 3 4 5 6 7 8 9\]

\[\text{DecimalDigits} :: \text{DecimalDigit} \ \text{DecimalDigits} \ \text{DecimalDigit}\]

The definition

\[\text{LookaheadExample} :: \]
\[\text{n} \ \text{[lookahead \ \{1, 3, 5, 7, 9\} \ DecimalDigits} \ \text{DecimalDigit} \ \text{[lookahead \ \{a \ DecimalDigit\}]}\]

matches either the letter \text{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 “\text{[no Line Terminator 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 \text{LineTerminator} occurs in the input stream at the indicated position. For example, the production:

\[\text{ThrowStatement} :: \]
\[\text{throw} \ [\text{no Line Terminator here}] \ \text{Expression} ;\]

indicates that the production may not be used if a \text{LineTerminator} occurs in the script between the \text{throw} token and the \text{Expression}.

Unless the presence of a \text{LineTerminator} is forbidden by a restricted production, any number of occurrences of \text{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 “\text{[Lexical goal LexicalGoalSymbol]}” appears on the right-hand side of a syntactic production then the next token must be lexically recognised 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 “\text{but not}” and then indicating the expansions to be excluded. For example, the production:

\[\text{Identifier} :: \]
\[\text{IdentifierName} \ \text{but not} \ \text{ReservedWord}\]

means that the nonterminal \text{Identifier} may be replaced by any sequence of characters that could replace \text{IdentifierName} provided that the same sequence of characters could not replace \text{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:

\[\text{SourceCharacter} :: \]
\[\text{any Unicode code point}\]
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 parameterised, 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 nonterminal symbols of the production alternative as if they were parameters of the algorithm. When used in this manner, nonterminal symbols refer to the actual alternative definition that is matched when parsing the script source code.

When an algorithm is associated with a production alternative, the alternative is typically shown without any ‘[]’ grammar annotations. Such annotations should only affect the syntactic recognition of the alternative and have no effect on the associated semantics for the alternative.

Unless explicitly specified otherwise, all chain productions have an implicit associated definition for every algorithm that might be applied to that production’s left-hand side nonterminal. The implicit definition 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 evaluation 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 evaluation 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 letters 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. Subsubsubstep
            i. Subsubsubsubstep

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

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

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

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

The mathematical function \( \text{sign}(x) \) produces $1$ 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 mathematical function \( \text{min}(x_1, x_2, ..., x_n) \) produces the mathematically smallest of \( x_1 \) through \( x_n \).

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

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

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

### 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 \texttt{Contains} which takes an argument named \texttt{symbol} whose value is a terminal or nonterminal of the grammar that includes the associated production. The default definition of \texttt{Contains} is:

1. For each terminal and nonterminal grammar symbol, \texttt{sym}, in the definition of this production do
   a. If \texttt{sym} is the same grammar symbol as \texttt{symbol}, return \texttt{true}.
   b. If \texttt{sym} is a nonterminal, then
      i. Let \texttt{contained} be the result of \texttt{Contains for sym} with argument \texttt{symbol}.
      ii. If \texttt{contained} is \texttt{true}, return \texttt{true}.
2. Return \texttt{false}.
The above definition is explicitly over-ridden for specific productions.

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

6 ECMAScript Data Types and Values

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.

6.1 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, Symbol, Number, and Object. An ECMAScript language value is a value that is characterized by an ECMAScript language type.

6.1.1 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.

6.1.2 The Null Type

The Null type has exactly one value, called null.

6.1.3 The Boolean Type

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

6.1.4 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 guarantee 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 \(c_1\) is in the range 0xD800 to 0xDBFF and the second code unit \(c_2\) is in the range 0xDC00 to 0xDFFF, is a surrogate pair and is interpreted as a code point with the value \((c_1 - 0xD800) \times 0x400 + (c_2 - 0xDC00) + 0x10000\).

- 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.

6.1.5 The Symbol Type

The Symbol type is the set of all non-String values that may be used as the key of an Object property (6.1.7).

Each possible Symbol values is unique and immutable.

Symbol values have an associated internal attribute called [[Description]] whose immutable value is either undefined or a String value.

6.1.5.1 Well-Known Symbols

Well-known symbols are built-in Symbol values that are explicitly referenced by algorithms of this specification. They are typically used as the keys of properties whose values serve as extension points of a specification algorithm. Unless otherwise specified, well-known symbols values are shared by all Code Realms (8.2).

Within this specification a well-known symbol is referred to by using a notation of the form @@name, where "name" is one of the values listed in Table 1.
Table 1 — Well-known Symbols

<table>
<thead>
<tr>
<th>Specification Name</th>
<th>[[Description]]</th>
<th>Value and Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>@@create</td>
<td><code>Symbol.create</code></td>
<td>A method used to allocate an object. Called from the [[Construct]] internal method.</td>
</tr>
<tr>
<td>@@hasInstance</td>
<td><code>Symbol.hasInstance</code></td>
<td>A method that determines if a constructor object recognises an object as one of the constructor’s instances. Called by the semantics of the <code>instanceof</code> operator.</td>
</tr>
<tr>
<td>@@isConcatSpreadable</td>
<td><code>Symbol.isConcatSpreadable</code></td>
<td>A Boolean value that if true indicates that an object should be flatten to its array elements by <code>Array.prototype.concat</code>.</td>
</tr>
<tr>
<td>@@isRegExp</td>
<td><code>Symbol.isRegExp</code></td>
<td>A Boolean value that if true indicates that an object may be used as a regular expression.</td>
</tr>
<tr>
<td>@@iterator</td>
<td><code>Symbol.iterator</code></td>
<td>A method that returns the default iterator for an object. Called by the semantics of the for-of statement.</td>
</tr>
<tr>
<td>@@toPrimitive</td>
<td><code>Symbol.toPrimitive</code></td>
<td>A method that converts an object to a corresponding primitive value. Called by the <code>ToPrimitive</code> abstract operation.</td>
</tr>
<tr>
<td>@@toStringTag</td>
<td><code>Symbol.toStringTag</code></td>
<td>A string value that is used in the creation of the default string description of an object. Called by the <code>Object.prototype.toString</code> method.</td>
</tr>
<tr>
<td>@@unscopables</td>
<td><code>Symbol.unscopables</code></td>
<td>An Array of string values that are property names that are excluded from the with environment bindings of the associated objects.</td>
</tr>
</tbody>
</table>

6.1.6 The Number Type

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

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

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

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

The 18437736874454810622 (that is, \(2^{64} - 2^{53} - 2\)) finite nonzero values are of two kinds:
(that is, $2^{64} - 2^{54}$) of them are normalised, having the form

$$x \times m \times 2^e$$

where $x = +1$ 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

$$x \times m \times 2^e$$

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

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

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

In this specification, the phrase "the Number value for $x$" where $x$ represents an exact nonzero real mathematical quantity (which might even be an irrational number such as $\pi$) means a Number value chosen in the following manner. Consider the set of all finite values of the Number type, with 0 removed and with two additional values added to it that are not representable in the Number type, namely $2^{1024}$ (which is $+1 \times 2^{1024} \times 2^{971}$) and $-2^{1024}$ (which is $-1 \times 2^{1024} \times 2^{971}$). 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 7.1.5 and 7.1.6, respectively.

### 6.1.7 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.
- An 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.

Properties are identified using key values. A key value is either an ECMAScript String value or a Symbol value. All String and Symbol values, including the empty string, are valid as property keys.

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 accessible 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 properties of the associated object. Each own properties of an object must each have a key value that is distinct from the key values of the other own properties of that object.

All objects are logically collections of properties, but there are multiple forms of objects that differ in their semantics for accessing and manipulating their properties. Ordinary objects 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.

### 6.1.7.1 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 2.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Value Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Value]]</td>
<td>Any ECMAScript</td>
<td>The value retrieved by a get access of the property.</td>
</tr>
<tr>
<td></td>
<td>language type</td>
<td></td>
</tr>
<tr>
<td>[[Writable]]</td>
<td>Boolean</td>
<td>If false, attempts by ECMAScript code to change the property's [[Value]] attribute using [[Set]] will not succeed.</td>
</tr>
<tr>
<td>[[Enumerable]]</td>
<td>Boolean</td>
<td>If true, the property will be enumerated by a for-in enumeration (see 13.6.4). Otherwise, the property is said to be non-enumerable.</td>
</tr>
<tr>
<td>[[Configurable]]</td>
<td>Boolean</td>
<td>If false, attempts to delete the property, change the property to be an accessor property, or change its attributes (other than [[Value]], or changing [[Writable]] to false) will fail.</td>
</tr>
</tbody>
</table>

An accessor property associates a key value with the attributes listed in Table 3.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Value Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Get]]</td>
<td>Object or Undefined</td>
<td>If the value is an Object it must be a function Object. The function’s [[Call]] internal method (Table 6) is called with an empty arguments list to retrieve the property value each time a get access of the property is performed.</td>
</tr>
<tr>
<td>[[Set]]</td>
<td>Object or Undefined</td>
<td>If the value is an Object it must be a function Object. The function’s [[Call]] internal method (Table 6) 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 [[Set]] internal method may, but is not required to, have an effect on the value returned by subsequent calls to the property's [[Get]] internal method.</td>
</tr>
<tr>
<td>[[Enumerable]]</td>
<td>Boolean</td>
<td>If true, the property is to be enumerated by a for-in enumeration (see 13.6.4). Otherwise, the property is said to be non-enumerable.</td>
</tr>
<tr>
<td>[[Configurable]]</td>
<td>Boolean</td>
<td>If false, attempts to delete the property, change the property to be a data property, or change its attributes will fail.</td>
</tr>
</tbody>
</table>

If the initial values of a property's attributes are not explicitly specified by this specification, the default value defined in Table 4 is used.
6.1.7.2 Object Internal Methods and Internal Slots

The actual semantics of objects, in ECMAScript, 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 method names are polymorphic. This means that different object values may perform different algorithms when a common internal method name is invoked 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 slots correspond to internal state that is associated with objects and used by various ECMAScript specification algorithms. Internal slots are not object properties and they are not inherited. Depending upon the specific internal slot specification, such state may consist of values of any ECMAScript language type or of specific ECMAScript specification type values. Unless explicitly specified otherwise, internal slots are allocated as part of the process of creating an object and may not be dynamically added to an object. Unless specified otherwise, the initial value of an internal slot is the value `undefined`. Various algorithms within this specification create objects that have internal slots. However, the ECMAScript language provides no direct way to associate internal slots with an object.

Internal methods and internal slots are identified within this specification using names enclosed in double square brackets `[]`.

Table 5 summarises the essential internal methods used by this specification that are applicable to all objects created or manipulated by ECMAScript code. 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 5 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 6 augmented by the following additional names: “any” means the value may be any ECMAScript language type. An internal method implicitly returns a Completion Record as described in 6.2.2. In addition to its parameters, an internal method always has access to the object upon which it is invoked as a method.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Value]]</td>
<td>undefined</td>
</tr>
<tr>
<td>[[Get]]</td>
<td>undefined</td>
</tr>
<tr>
<td>[[Set]]</td>
<td>undefined</td>
</tr>
<tr>
<td>[[Writable]]</td>
<td>false</td>
</tr>
<tr>
<td>[[Enumerable]]</td>
<td>false</td>
</tr>
<tr>
<td>[[Configurable]</td>
<td>false</td>
</tr>
</tbody>
</table>
### Table 5 — Essential Internal Methods

<table>
<thead>
<tr>
<th>Internal Method</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[GetPrototypeOf]]</td>
<td>( ) → Object or Null</td>
<td>Determine the object that provides inherited properties for this object. A null value indicates that there are no inherited properties.</td>
</tr>
<tr>
<td>[[SetPrototypeOf]]</td>
<td>(Object or Null) → Boolean</td>
<td>Associate with an object another object that provides inherited properties. Passing null indicates that there are no inherited properties. Returns true indicating that the operation was completed successfully or false indicating that the operation was not successful.</td>
</tr>
<tr>
<td>[[IsExtensible]]</td>
<td>( ) → Boolean</td>
<td>Determine whether it is permitted to add additional properties to an object.</td>
</tr>
<tr>
<td>[[PreventExtensions]]</td>
<td>( ) → Boolean</td>
<td>Control whether new properties may be added to an object. Returns true indicating that the operation was completed successfully or false indicating that the operation was not successful.</td>
</tr>
<tr>
<td>[[GetOwnProperty]]</td>
<td>(propertyKey) → The Property Descriptor</td>
<td>Returns a Property Descriptor for the own property of this object whose key is propertyKey, or undefined if no such property exists.</td>
</tr>
<tr>
<td>[[HasProperty]]</td>
<td>(propertyKey) → Boolean</td>
<td>Returns a Boolean value indicating whether the object already has either an own or inherited property whose key is propertyKey.</td>
</tr>
<tr>
<td>[[Get]]</td>
<td>(propertyKey, Receiver) → any</td>
<td>Retrieve 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.</td>
</tr>
<tr>
<td>[[Set]]</td>
<td>(propertyKey, value, Receiver) → Boolean</td>
<td>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.</td>
</tr>
<tr>
<td>[[Delete]]</td>
<td>(propertyKey) → Boolean</td>
<td>Removes the own property indentified by the propertyKey parameter from the object. Return false if the property was not deleted and is still present. Return true if the property was deleted or was not present.</td>
</tr>
<tr>
<td>[[DefineOwnProperty]]</td>
<td>(propertyKey, PropertyDescriptor) → Boolean</td>
<td>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.</td>
</tr>
<tr>
<td>[[Enumerate]]</td>
<td>( ) → Object</td>
<td>Returns an iterator object over the string values of the keys of the enumerable properties of the object.</td>
</tr>
<tr>
<td>[[OwnPropertyKeys]]</td>
<td>( ) → Object</td>
<td>Returns an Iterator object that produces all of the own property keys for the object.</td>
</tr>
</tbody>
</table>

Table 6 summarises additional essential internal methods that are supported by objects that may be called as functions.
Table 6 — Additional Essential Internal Methods of Function Objects

<table>
<thead>
<tr>
<th>Internal Method</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Call]]</td>
<td>(any, a List of any) → any</td>
<td>Executes code associated with the object. Invoked via a function call expression. The arguments to the internal method are a <code>this</code> value and a list containing the arguments passed to the function by a call expression. Objects that implement this internal method are callable.</td>
</tr>
<tr>
<td>[[Construct]]</td>
<td>(a List of any) → Object</td>
<td>Creates an object. Invoked via the <code>new</code> operator. The arguments to the internal method are the arguments passed to the <code>new</code> operator. Objects that implement this internal method are called constructors. A Function object is not necessarily a constructor and such non-constructor Function objects do not have a <code>[[Construct]]</code> internal method.</td>
</tr>
</tbody>
</table>

6.1.7.3 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 enforced.

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. `desc1` 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 `[[Set]]` internal method behaviour.

`[[GetPrototypeOf]]`

Every `[[Prototype]]` chain must have finite length (that is, starting from any object, recursively accessing the `[[Prototype]]` internal slot 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)

**hasOwnProperty**

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

get

Invariant checks:

- if `false` is returned, check if the target property is configurable

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

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)

- 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

Unless otherwise specified, the ECMAScript standard objects are ordinary objects and behave as described in 9.1 and 9.2. Some standard objects are exotic objects and have behaviour defined in 9.3 and 9.4.

Exotic objects may implement internal methods in any manner unless specified otherwise; for example, one possibility is that `[[Get]]` and `[[Set]]` for a particular exotic object indeed fetch and store property values but `[[GetProperty]]` always generates `undefined`. However, if any specified use of an exotic object's internal methods is not supported by an implementation, that usage 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 7.2.3) 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.

### 6.1.7.4 Well-Known Intrinsic Objects

Well-known intrinsics are built-in objects that are explicitly referenced by the algorithms of this specification and which usually have Realm specific identities. Unless otherwise specified each intrinsic object actually corresponds to a set of similar objects, one per Realm.

Within this specification a reference such as %name% means the intrinsic object, associated with the current Realm, corresponding to the name. Determination of the current Realm and its intrinsics is described in 8.2. The well-known intrinsics are listed in Table 7.
### Table 7 — Well-known Intrinsic Objects

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>ECMAScript Language Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Object%</td>
<td>The initial value of the global object property named &quot;Object&quot;.</td>
</tr>
<tr>
<td>%ObjectPrototype%</td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %Object%.</td>
</tr>
<tr>
<td>%ObjProto_toString%</td>
<td>The initial value of the &quot;toString&quot; data property of the intrinsic %ObjectPrototype%.</td>
</tr>
<tr>
<td>%Function%</td>
<td>The initial value of the global object property named &quot;Function&quot;.</td>
</tr>
<tr>
<td>%FunctionPrototype%</td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %Function%.</td>
</tr>
<tr>
<td>%Array%</td>
<td>The initial value of the global object property named &quot;Array&quot;.</td>
</tr>
<tr>
<td>%ArrayPrototype%</td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %Array%.</td>
</tr>
<tr>
<td>%ArrayIteratorPrototype%</td>
<td>The prototype object used for Iterator objects created by the CreateArrayIterator abstract operation.</td>
</tr>
<tr>
<td>%String%</td>
<td>The initial value of the global object property named &quot;String&quot;.</td>
</tr>
<tr>
<td>%StringPrototype%</td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %String%.</td>
</tr>
<tr>
<td>%StringIteratorPrototype%</td>
<td>The prototype object used for Iterator objects created by the CreateStringIterator abstract operation</td>
</tr>
<tr>
<td>%Boolean%</td>
<td>The initial value of the global object property named &quot;Boolean&quot;.</td>
</tr>
<tr>
<td>%BooleanPrototype%</td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %Boolean%.</td>
</tr>
<tr>
<td>%Number%</td>
<td>The initial value of the global object property named &quot;Number&quot;.</td>
</tr>
<tr>
<td>%NumberPrototype%</td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %Number%.</td>
</tr>
<tr>
<td>%Date%</td>
<td>The initial value of the global object property named &quot;Date&quot;.</td>
</tr>
<tr>
<td>%DatePrototype%</td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %Date%.</td>
</tr>
<tr>
<td>%RegExp%</td>
<td>The initial value of the global object property named &quot;RegExp&quot;.</td>
</tr>
<tr>
<td>%RegExpPrototype%</td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %RegExp%.</td>
</tr>
<tr>
<td>%Map%</td>
<td>The initial value of the global object property named &quot;Map&quot;.</td>
</tr>
<tr>
<td>%MapPrototype%</td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %Map%.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>%MapIteratorPrototype%</td>
<td>The prototype object used for Iterator objects created by the CreateMapIterator abstract operation</td>
</tr>
<tr>
<td>%WeakMap%</td>
<td>The initial value of the global object property named &quot;WeakMap&quot;.</td>
</tr>
<tr>
<td>%WeakMapPrototype%</td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %WeakMap%.</td>
</tr>
<tr>
<td>%Set%</td>
<td>The initial value of the global object property named &quot;Set&quot;.</td>
</tr>
<tr>
<td>%SetPrototype%</td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %Set%.</td>
</tr>
<tr>
<td>%WeakSet%</td>
<td>The initial value of the global object property named &quot;WeakSet&quot;.</td>
</tr>
<tr>
<td>%WeakSetPrototype%</td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %WeakSet%.</td>
</tr>
<tr>
<td>%SetIteratorPrototype%</td>
<td>The prototype object used for Iterator objects created by the CreateSetIterator abstract operation</td>
</tr>
<tr>
<td>%GeneratorFunction%</td>
<td>The initial value of the name &quot;GeneratorFunction&quot; exported from the built-in module &quot;std:iteration&quot;</td>
</tr>
<tr>
<td>%Generator%</td>
<td>The initial value of the name &quot;Generator&quot; exported from the built-in module &quot;std:iteration&quot;</td>
</tr>
<tr>
<td>%GeneratorPrototype%</td>
<td>The initial value of the &quot;prototype&quot; property of the %Generator% intrinsic</td>
</tr>
<tr>
<td>%Error%</td>
<td></td>
</tr>
<tr>
<td>%EvalError%</td>
<td></td>
</tr>
<tr>
<td>%RangeError%</td>
<td></td>
</tr>
<tr>
<td>%ReferenceError%</td>
<td></td>
</tr>
<tr>
<td>%SyntaxError%</td>
<td></td>
</tr>
<tr>
<td>%TypeError%</td>
<td></td>
</tr>
<tr>
<td>%URIError%</td>
<td></td>
</tr>
<tr>
<td>%ErrorPrototype%</td>
<td></td>
</tr>
<tr>
<td>%EvalErrorPrototype%</td>
<td></td>
</tr>
<tr>
<td>%RangeErrorPrototype%</td>
<td></td>
</tr>
<tr>
<td>%ReferenceErrorPrototype%</td>
<td></td>
</tr>
<tr>
<td>%SyntaxErrorPrototype%</td>
<td></td>
</tr>
<tr>
<td>%TypeErrorPrototype%</td>
<td></td>
</tr>
<tr>
<td>%URIErrorPrototype%</td>
<td></td>
</tr>
<tr>
<td>%ArrayBuffer%</td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %ArrayBuffer%</td>
</tr>
<tr>
<td>%TypedArray%</td>
<td></td>
</tr>
<tr>
<td>%TypedArrayPrototype%</td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %TypedArray%</td>
</tr>
<tr>
<td>%Int8Array%</td>
<td></td>
</tr>
<tr>
<td>%Int8ArrayPrototype%</td>
<td></td>
</tr>
<tr>
<td>%DataView%</td>
<td></td>
</tr>
</tbody>
</table>

Commented [AWB164]: TODO add all the other TypedArray view intrinsics
<table>
<thead>
<tr>
<th><strong>%DataViewPrototype%</strong></th>
<th><strong>%ThrowTypeError%</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>A function object that unconditionally throws a new instance of %TypeError%.</td>
<td></td>
</tr>
</tbody>
</table>

**Commented [AWB125]:** TODO more to come TODO
6.2 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, 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.

6.2.1 The List and Record Specification Type

The List type is used to explain the evaluation of argument lists (see 12.2.6) in `new` expressions, in function calls, and in other algorithms where a simple ordered list of values is needed. Values of the List type are simply ordered sequences of list elements containing the individual values. These sequences may be of any length. The elements of a list may be randomly accessed using 0-origin indices. For notational convenience an array-like syntax can be used to access List elements. For example, `arguments[2]` is shorthand for saying the 3rd element of the List arguments.

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 initialised 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: `PropertyDescriptor{[[Value]]: 42, [[Writable]]: false, [[Configurable]]: true}`.

6.2.2 The Completion Record Specification Type

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

Values of the Completion type are Record values whose fields are defined as by Table 8.

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

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

© Ecma International 2009 – All rights reserved
6.2.2.1 NormalCompletion

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

1. Return NormalCompletion(argument).

Is a shorthand that is defined as follows:

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

6.2.2.2 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:

```
return "Infinity".
```

generally means the same thing as:

1. Return NormalCompletion("Infinity").

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

1. Return NormalCompletion(undefined).

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.

6.2.2.3 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}.

6.2.2.4 ReturnIfAbrupt

Algorithms steps that say

```
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]].

6.2.3 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 language features. For example, the left-hand operand of an assignment is expected to produce a reference.
A Reference is a resolved name or property 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 Symbol, a Number, or an environment record (8.1.1). A base value of undefined indicates that the Reference could not be resolved to a binding. The referenced name is a String or Symbol value.

A Super Reference is a Reference that is used to represent 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 flag component of the reference V.
- HasPrimitiveBase(V). Returns true if Type(base) is a Boolean, String, Symbol, 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:

6.2.3.1 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. Assert: In this case, base will never be null or undefined.
   b. Return the result of calling the [[Get]] 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 8.1.1) 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 abstract operation and the ordinary object [[Get]] internal method. An implementation might choose to avoid the actual creation of the object.

6.2.3.2 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 abstract 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. Assert: 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 [[Set]] 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 (6.1.1) 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 and the ordinary object [[Set]] internal method. An implementation might choose to avoid the actual creation of that object.

6.2.3.3 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).

6.2.4 The Property Descriptor Specification Type

The Property Descriptor type is used to explain the manipulation and reification of Object 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 6.1.7.1. In addition, any field may be present or absent. The schema name used within this specification to tag literal descriptions of Property Descriptor records is "PropertyDescriptor".

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 Table 2 or Table 3.

A Property Descriptor may be derived from an 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:

6.2.4.1 IsAccessorDescriptor ( Desc )

When the abstract operation IsAccessorDescriptor is called with Property Descriptor Desc, the following steps are taken:
1. If Desc is undefined, then return false.
2. If both Desc.[[Get]] and Desc.[[Set]] are absent, then return false.
3. Return true.
6.2.4.2 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.

6.2.4.3 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.

6.2.4.4 FromPropertyDescriptor (Desc)

When the abstract operation FromPropertyDescriptor is called with Property Descriptor Desc, the following steps are taken:
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 with the intrinsic object %ObjectPrototype% as its argument.
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 PropertyDescriptor{[[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 PropertyDescriptor{[[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 PropertyDescriptor{[[Value]]: Desc.[[Get]], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}.
8. If Desc has a [[Set]] field, then
   a. Call OrdinaryDefineOwnProperty with arguments obj, "set", and PropertyDescriptor{[[Value]]: Desc.[[Set]], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}.
9. If Desc has an [[Enumerable]] field, then
   a. Call OrdinaryDefineOwnProperty with arguments obj, "enumerable", and PropertyDescriptor{[[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 PropertyDescriptor{[[Value]]: Desc.[[Configurable]], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}.
11. Return obj.

6.2.4.5 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.

6.2.4.6 CompletePropertyDescriptor (Desc, LikeDesc)

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

1. Assert: LikeDesc is either a Property Descriptor or undefined.
2. ReturnIfAbrupt(Desc).
3. Assert: Desc is a Property Descriptor.
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 an [[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.

6.2.5 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 8.1.
6.2.6 Data Blocks

The Data Block specification type is used to describe a distinct and mutable sequence of byte-sized (8 bit) numeric values. A Data Block value is created with a fixed number of bytes that each have the initial value 0.

For notational convenience within this specification, an array-like syntax can be used to express to the individual bytes of a Data Block value. This notation presents a Data Block value as a 0-origined integer indexed sequence of bytes. For example, if \( db \) is a 5 byte Data Block value then \( db[2] \) can be used to express access to its 3rd byte.

The following abstract operations are used in this specification to operate upon Data Block values:

6.2.6.1 CreateByteDataBlock(size)

When the abstract operation CreateByteDataBlock is called with integer argument \( size \), the following steps are taken:

1. Assert: \( size \geq 0 \).
2. Let \( db \) be a new Data Block value consisting of \( size \) bytes. If it is impossible to create such a Data Block, then throw a RangeError exception.
3. Set all of the bytes of \( db \) to 0.
4. Return \( db \).

6.2.6.2 CopyDataBlockBytes(toBlock, toIndex, fromBlock, fromIndex, count)

When the abstract operation CopyDataBlockBytes is called the following steps are taken:

1. Assert: \( fromBlock \) and \( toBlock \) are distinct Data Block values.
2. Assert: \( fromIndex \), \( toIndex \), and \( count \) are positive integer values.
3. Let \( fromSize \) be the number of bytes in \( fromBlock \).
4. Assert: \( fromIndex + count \leq fromSize \).
5. Let \( toSize \) be the number of bytes in \( toBlock \).
6. Assert: \( toIndex + count \leq toSize \).
7. Repeat, while \( count > 0 \)
   a. Set \( toBlock[toIndex] \) to the value of \( fromBlock[fromIndex] \).
   b. Increment \( toIndex \) and \( fromIndex \) each by 1.
   c. Decrement \( count \) by 1.
8. Return NormalCompletion(empty).

7 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.

7.1 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; they can accept a value of any ECMAScript language type or of a Completion Record value. But no other specification types are used with these operations.

7.1.1 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 9:

Table 9 — ToPrimitive Conversions

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completion Record</td>
<td>If argument is an abrupt completion, return argument. Otherwise return ToPrimitive(argument[[value]]) also passing the optional hint PreferredType.</td>
</tr>
<tr>
<td>Undefined</td>
<td>Return argument (no conversion).</td>
</tr>
<tr>
<td>Null</td>
<td>Return argument (no conversion).</td>
</tr>
<tr>
<td>Boolean</td>
<td>Return argument (no conversion).</td>
</tr>
<tr>
<td>String</td>
<td>Return argument (no conversion).</td>
</tr>
<tr>
<td>Symbol</td>
<td>Return argument (no conversion).</td>
</tr>
<tr>
<td>Object</td>
<td>Perform the steps following this table.</td>
</tr>
</tbody>
</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 String, let hint be "string".
3. Else PreferredType is hint Number, let hint be "number".
4. Let exoticToPrim be the result of Get(argument, @@toPrimitive).
5. ReturnIfAbrupt(exoticToPrim).
6. If exoticToPrim is not undefined, then
   a. If IsCallable(exoticToPrim) is false, then throw a TypeError exception.
   b. Let result be the result of calling the [[Call]] internal method of exoticToPrim, with argument as thisArgument and a List containing hint as argumentsList.
   c. ReturnIfAbrupt(result).
   d. If result is an ECMAScript language value and Type(result) is not Object, then return result.
   e. Else, throw a TypeError exception.
7. If hint is "default" then, let hint be "number".
8. Return the result of OrdinaryToPrimitive(argument, 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 value is either "string" or "number".
3. If hint is "string", then
   a. Let methodName be the List ( "toString", "valueOf" ).
4. Else,
   a. Let methodName be the List ( "valueOf", "toString" ).
5. For each name in methodName in List order, do
   a. Let method be the result of Get(O, name).
   b. If IsCallable(method) is true then,
      i. Let result be the result of calling the [[Call]] internal method of method, with O as thisArgument and an empty List as argumentsList.
      ii. ReturnIfAbrupt(result).
   c. If Type(result) is not Object, then return result.
6. 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 @@toPrimitive method. Of the objects defined in this specification only Date objects (see 20.3) and Symbol objects (see 19.4.3.4) over-ride the default ToPrimitive behaviour. Date objects treat no hint as if the hint were String.

7.1.2 ToBoolean

The abstract operation ToBoolean converts its argument to a value of type Boolean according to Table 10:

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

7.1.3 ToNumber

The abstract operation ToNumber converts its argument to a value of type Number according to Table 11:

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

7.1.3.1 ToNumber Applied to the String Type

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

Syntax

StringNumericLiteral :::
  StrWhiteSpaceopt StrNumericLiteral StrWhiteSpaceopt
  StrWhiteSpaceopt

© Ecma International 2009 – All rights reserved
StrWhiteSpace ::: StrWhiteSpaceChar StrWhiteSpaceOpt

StrWhiteSpaceChar ::: Whitespace LineTerminator

StrNumericLiteral ::: StrDecimalLiteral HexIntegerLiteral

StrDecimalLiteral ::: StrUnsignedDecimalLiteral
  + StrUnsignedDecimalLiteral
  - StrUnsignedDecimalLiteral

StrUnsignedDecimalLiteral :::
  Infinity
  DecimalDigits . DecimalDigits [optional ExponentPart]
  . DecimalDigits [optional ExponentPart]
  DecimalDigits [optional ExponentPart]

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 11.8.3):
  • 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 recognised as a StringNumericLiteral but not as a NumericLiteral.
7.1.3.1.1 Runtime Semantics: MV's

The conversion of a String to a Number value is similar overall to the determination of the Number value for a numeric literal (see 11.8.3), 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 StrNumericLiteral StrWhiteSpace 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 ** MV (a value so large that it will round to Infinity).
- 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 ** n, where n is the number of characters in the second DecimalDigits and e is the MV of ExponentPart.
- The MV of StrUnsignedDecimalLiteral ::: DecimalDigits . ExponentPart is 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 ** n, where n is the number of characters in DecimalDigits and e is the MV of ExponentPart.
- The MV of StrUnsignedDecimalLiteral :: DecimalDigits . ExponentPart is the MV of DecimalDigits times 10 ** n, where e is the MV of ExponentPart.
- The MV of DecimalDigits :: DecimalDigit is the MV of DecimalDigit.
- The MV of DecimalDigits :: DecimalDigit DecimalDigit is the MV of DecimalDigit 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 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 6.1.6, 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;
- there is a nonzero digit to its left and there is a nonzero digit, not in the ExponentPart, to its right.

### 7.1.4 ToInteger

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

1. Let \(\text{number}\) be the result of calling ToNumber on the input argument.
2. \(\text{ReturnIfAbrupt(number)}\).
3. If \(\text{number}\) is NaN, return +0.
4. If \(\text{number}\) is +0, −0, +∞, or −∞, return \(\text{number}\).
5. Return the result of computing \(\text{sign(number)} \times \text{floor(abs(number))}\).

### 7.1.5 ToInt32: (Signed 32 Bit Integer)

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

1. Let \(\text{number}\) be the result of calling ToNumber on the input argument.
2. \(\text{ReturnIfAbrupt(number)}\).
3. If \(\text{number}\) is NaN, +0, −0, +∞, or −∞, return +0.
4. Let \(\text{int}\) be \(\text{sign(number)} \times \text{floor(abs(number))}\).
5. Let \(\text{int32bit}\) be \(\text{int} \mod 2^{32}\).
6. If \(\text{int32bit} \geq 2^{31}\), return \(\text{int32bit} - 2^{32}\), otherwise return \(\text{int32bit}\).

NOTE Given the above definition of ToInt32:

- The ToInt32 abstract operation is idempotent: if applied to a result that it produced, the second application leaves that value unchanged.
- \(\text{ToInt32(ToInt32(x))}\) is equal to \(\text{ToInt32(x)}\) for all values of \(x\). (It is to preserve this latter property that +∞ and −∞ are mapped to +0.)
- ToInt32 maps −0 to +0.
7.1.6 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 $2^{32}$.

NOTE Given the above definition of ToUint32:
• Step 6 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.

7.1.7 ToInt16: (Signed 16 Bit Integer)

The abstract operation ToInt16 converts its argument to one of $2^{16}$ integer values in the range −32768 through 32767, 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 int modulo $2^{16}$.
6. If int16bit ≥ $2^{15}$, return int16bit − $2^{16}$, otherwise return int16bit.

7.1.8 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 int modulo $2^{16}$.
6. Return int16bit.

NOTE Given the above definition of ToUint16:
• The substitution of $2^{16}$ for $2^{32}$ in step 5 is the only difference between ToUint32 and ToUint16.
• ToUint16 maps −0 to +0.

7.1.9 ToInt8: (Signed 8 Bit Integer)

The abstract operation ToInt8 converts its argument to one of $2^{8}$ integer values in the range −128 through 127, 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 int8bit be int modulo $2^{8}$.
6. If \(\text{int8bit} \geq 2^8\), return \(\text{int8bit} - 2^8\); otherwise return \(\text{int8bit}\).

### 7.1.10 ToUint8: (Unsigned 8 Bit Integer)

The abstract operation ToUint8 converts its argument to one of \(2^8\) integer values in the range 0 through 255, inclusive. This abstract operation functions as follows:

1. Let \(\text{number}\) be the result of calling ToNumber on the input argument.
2. ReturnIfAbrupt(\(\text{number}\)).
3. If \(\text{number}\) is NaN, +0, -0, +\(\infty\), or -\(\infty\), return +0.
4. Let \(\text{int}\) be \(\text{sign(\(\text{number}\))} \times \text{floor(abs(\(\text{number}\ )))}\).
5. Let \(\text{int8bit}\) be \(\text{int}\) modulo \(2^8\).
6. Return \(\text{int8bit}\).

### 7.1.11 ToUint8Clamp: (Unsigned 8 Bit Integer, Clamped)

The abstract operation ToUint8Clamp converts its argument to one of \(2^8\) integer values in the range 0 through 255, inclusive. This abstract operation functions as follows:

1. Let \(\text{number}\) be the result of calling ToNumber on the input argument.
2. ReturnIfAbrupt(\(\text{number}\)).
3. If \(\text{number}\) is NaN, return +0.
4. If \(\text{number} \leq 0\), return +0.
5. If \(\text{number} > 255\), return 255.
6. Let \(f\) be \(\text{floor(\(\text{number}\))}\).
7. If \(f + 0.5 \leq \text{number}\), then return \(f + 1\).
8. Return \(f\).

**NOTE**  Note that unlike the other integer conversion abstract operation, ToUint8Clamp rounds rather than truncates non-integer values.

### 7.1.12 ToString

The abstract operation ToString converts its argument to a value of type String according to Table 12:

#### Table 12 — ToString Conversions

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completion Record</td>
<td>If (\text{argument}) is an abrupt completion, return (\text{argument}). Otherwise return ToPrimitive((\text{argument}[[\text{value}]]))</td>
</tr>
<tr>
<td>Undefined</td>
<td>&quot;undefined&quot;</td>
</tr>
<tr>
<td>Null</td>
<td>&quot;null&quot;</td>
</tr>
<tr>
<td>Boolean</td>
<td>If (\text{argument}) is \text{true}, then return &quot;true&quot;. If (\text{argument}) is \text{false}, then return &quot;false&quot;.</td>
</tr>
<tr>
<td>Number</td>
<td>See 7.1.12.1.</td>
</tr>
<tr>
<td>String</td>
<td>Return (\text{argument}) (no conversion)</td>
</tr>
<tr>
<td>Symbol</td>
<td>Throw a TypeError exception.</td>
</tr>
</tbody>
</table>
| Object        | Apply the following steps:  
  1. Let \(\text{primValue}\) be ToPrimitive(\(\text{argument}\), hint String).  
  2. Return ToString(\(\text{primValue}\)). |

#### 7.1.12.1 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 +\( \infty \), return the String "Infinity".
5. Otherwise, let \( n \), \( k \), and \( x \) be integers such that \( k \geq 1 \), \( 10^{n-1} \leq s < 10^n \), the Number value for \( s \times 10^n \) 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 \( n-k \) is positive or negative, followed by the decimal representation of the integer \( \text{abs}(n-1) \) (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 \( n-k \) is positive or negative, followed by the decimal representation of the integer \( \text{abs}(n-1) \) (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 \( x \) be integers such that \( k \geq 1 \), \( 10^{n-1} \leq s < 10^n \), the Number value for \( s \times 10^n \) is \( m \), and \( k \) is as small as possible. If there are multiple possibilities for \( s \), choose the value of \( s \) for which \( s 	imes 10^n \) 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:

7.1.13 ToObject

The abstract operation ToObject converts its argument to a value of type Object according to Table 13:
Table 13 — ToObject Conversions

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completion Record</td>
<td>If argument is an abrupt completion, return argument. Otherwise return ToObject(argument.[[value]])</td>
</tr>
<tr>
<td>Undefined</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Null</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Boolean</td>
<td>Return a new Boolean object whose [[BooleanData]] internal slot is set to the value of argument. See 19.3 for a description of Boolean objects.</td>
</tr>
<tr>
<td>Number</td>
<td>Return a new Number object whose [[NumberData]] internal slot is set to the value of argument. See 20.1 for a description of Number objects.</td>
</tr>
<tr>
<td>String</td>
<td>Return a new String object whose [[StringData]] internal slot is set to the value of argument. See 21.1 for a description of String objects.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Return a new Symbol object whose [[SymbolData]] internal slot is set to the value of argument. See 19.4 for a description of Symbol objects.</td>
</tr>
<tr>
<td>Object</td>
<td>Return argument (no conversion).</td>
</tr>
</tbody>
</table>

7.1.14 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 Symbol, then
   a. Return argument.
3. Return ToString(argument).

7.1.15 ToLength

The abstract operation ToLength converts its argument to an integer suitable for use as the length of an array-like object. It performs the following steps:

1. Let len be ToInteger(argument).
2. ReturnIfAbrupt(len).
3. If len ≤ +0, then return +0.
4. Return min(len, 2**53 - 1).

7.2 Testing and Comparison Operations

7.2.1 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 14:
Table 14 — CheckObjectCoercible Results

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completion Record</td>
<td>If argument is an abrupt completion, return argument. Otherwise return CheckObjectCoercible(argument.[value])</td>
</tr>
<tr>
<td>Undefined</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Null</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Boolean</td>
<td>Return argument</td>
</tr>
<tr>
<td>Number</td>
<td>Return argument</td>
</tr>
<tr>
<td>String</td>
<td>Return argument</td>
</tr>
<tr>
<td>Symbol</td>
<td>Return argument</td>
</tr>
<tr>
<td>Object</td>
<td>Return argument</td>
</tr>
</tbody>
</table>

7.2.2 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 15:

Table 15 — IsCallable Results

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

7.2.3 SameValue(x, y)

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 code units (same length and same code units 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. If Type(x) is Symbol, then
   a. If x and y are both the same Symbol value, then return true; otherwise, return false.
10. Return true if x and y are the same Object value. Otherwise, return false.
7.2.4 SameValueZero(x, y)

The internal comparison abstract operation SameValueZero(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 true.
   c. If x is -0 and y is +0, return true.
   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 code units (same length and same code units 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. If Type(x) is Symbol, then
   a. If x and y are both the same Symbol value, then return true; otherwise, return false.
10. Return true if x and y are the same Object value. Otherwise, return false.

NOTE SameValueZero differs from SameValue only in its treatment of +0 and -0.

7.2.5 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.

7.2.6 IsPropertyKey

The abstract operation IsPropertyKey determines if its argument, which must be an ECMAScript language value or a Completion Record, is a value that may be used as a property key.

1. ReturnIfAbrupt(argument).
2. If Type(argument) is String, return true.
3. If Type(argument) is Symbol, return true.
4. Return false.

7.2.7 IsExtensible(O)

The abstract operation IsExtensible is used to determine whether additional properties can be added to the object that is O. A Boolean value is returned. This abstract operation performs the following steps:

1. Assert: Type(O) is Object.
2. Return the result of calling the [[IsExtensible]] internal method of O.
7.2.8 Abstract Relational Comparison

The comparison \( x < y \), where \( x \) and \( y \) are values, produces \texttt{true}, \texttt{false}, or \texttt{undefined} (which indicates that at least one operand is \texttt{NaN}). In addition to \( x \) and \( y \) the algorithm takes a Boolean flag named \texttt{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 \texttt{LeftFirst} is \texttt{true} and indicates that the \( x \) parameter corresponds to an expression that occurs to the left of the \( y \) parameter's corresponding expression. If \texttt{LeftFirst} is \texttt{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 \texttt{LeftFirst} flag is \texttt{true}, then
   a. Let \( px \) be the result of calling \texttt{ToPrimitive}(\( x \), hint \texttt{Number}).
   b. ReturnIfAbrupt(\( px \)).
   c. Let \( py \) be the result of calling \texttt{ToPrimitive}(\( y \), hint \texttt{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 \texttt{ToPrimitive}(\( y \), hint \texttt{Number}).
   b. ReturnIfAbrupt(\( py \)).
   c. Let \( px \) be the result of calling \texttt{ToPrimitive}(\( x \), hint \texttt{Number}).
   d. ReturnIfAbrupt(\( px \)).
5. If both \( px \) and \( py \) are Strings, then
   a. If \( py \) is a prefix of \( px \), return \texttt{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 \texttt{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 \texttt{true}. Otherwise, return \texttt{false}.
6. Else,
   a. Let \( nx \) be the result of calling \texttt{ToNumber}(\( px \)). Because \( px \) and \( py \) are primitive values evaluation order is not important.
   b. Let \( ny \) be the result of calling \texttt{ToNumber}(\( py \)).
   c. If \( nx \) is \texttt{NaN}, return \texttt{undefined}.
   d. If \( ny \) is \texttt{NaN}, return \texttt{undefined}.
   e. If \( nx \) and \( ny \) are the same Number value, return \texttt{false}.
   f. If \( nx \) is \(+0\) and \( ny \) is \(-0\), return \texttt{false}.
   g. If \( nx \) is \(+0\) and \( ny \) is \(+0\), return \texttt{false}.
   h. If \( nx \) is \(+\infty\), return \texttt{false}.
   i. If \( ny \) is \(+\infty\), return \texttt{true}.
   j. If \( ny \) is \(-\infty\), return \texttt{false}.
   k. If \( nx \) is \(-\infty\), return \texttt{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 \texttt{true}. Otherwise, return \texttt{false}.

NOTE 1 Step 5 differs from step 11 in the algorithm for the addition operator \(+\) (12.6.3) 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.
7.2.9 Abstract Equality Comparison

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 \(x === y\).
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 ToNumber\(x\) == \(y\).
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.

7.2.10 Strict Equality Comparison

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 Symbol value, return true.
8. If \(x\) and \(y\) are the same Object value, return true.
9. Return false.

NOTE This algorithm differs from the SameValue Algorithm (7.2.3) in its treatment of signed zeroes and NaNs.

7.3 Operations on Objects

7.3.1 Get \((O, P)\)

The abstract operation Get is used to retrieve the value of a 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. Assert: Type(O) is Object.
2. Assert: IsPropertyKey(P) is true.
3. Return the result of calling the [[Get]] internal method of O passing P and O as the arguments.

### 7.3.2 Put (O, P, V, Throw)

The abstract operation Put is used to set the value of a 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. Assert: Type(O) is Object.
2. Assert: IsPropertyKey(P) is true.
3. Assert: Type(Throw) is Boolean.
4. Let success be the result of calling the [[Set]] 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.

### 7.3.3 CreateDataProperty (O, P, V)

The abstract operation CreateDataProperty 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 value for the property. This abstract operation performs the following steps:

1. Assert: Type(O) is Object.
2. Assert: IsPropertyKey(P) is true.
3. Let newDesc be the PropertyDescriptor{ [[Value]]: V, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true }.  
4. Return the result of calling the [[DefineOwnProperty]] internal method of O passing P and newDesc as arguments.

NOTE This abstract operation creates a property whose attributes are set to the same defaults used for properties created by the ECMAScript language assignment operator. Normally, the property will not already exist. If it does exist and is not configurable or O is not extensible [[DefineOwnProperty]] will return false.

### 7.3.4 CreateDataPropertyOrThrow (O, P, V)

The abstract operation CreateDataPropertyOrThrow is used to create a new own property of an object. It throws a TypeError exception if the requested property update cannot be performed. The operation is called with arguments O, P, and V where O is the object, P is the property key, and V is the value for the property. This abstract operation performs the following steps:

1. Assert: Type(O) is Object.
2. Assert: IsPropertyKey(P) is true.
3. Let success be the result of CreateDataProperty(O, P, V).
4. ReturnIfAbrupt(success).
5. If success is false, then throw a TypeError exception.

NOTE This abstract operation creates a property whose attributes are set to the same defaults used for properties created by the ECMAScript language assignment operator. Normally, the property will not already exist. If it does exist and is not configurable or O is not extensible [[DefineOwnProperty]] will return false causing this operation to throw a TypeError exception.

### 7.3.5 DefinePropertyOrThrow (O, P, desc)

The abstract operation DefinePropertyOrThrow is used to call the [[DefineOwnProperty]] internal method of an object in a manner that will throw a TypeError exception if the requested property update cannot 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 the Property Descriptor for the property. This abstract operation perform, the following steps:
1. Assert: Type(O) is Object.
2. Assert: IsPropertyKey(P) is true.
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.

7.3.6 DeletePropertyOrThrow (O, P)

The abstract operation DeletePropertyOrThrow is used to remove a specific own property of an object. It throws an exception if 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. Assert: Type(O) is Object.
2. Assert: IsPropertyKey(P) is true.
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.

7.3.7 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 returned. 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. Assert: Type(O) is Object.
2. Assert: IsPropertyKey(P) is true.
3. Return the result of calling the [[HasProperty]] internal method of O with argument P.

7.3.8 HasOwnProperty (O, P)

The abstract operation HasOwnProperty is used to determine whether an object has an own property with the specified property key. A Boolean value is returned. 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. Assert: Type(O) is Object.
2. Assert: IsPropertyKey(P) is true.
3. Let desc be the result of calling the [[GetOwnProperty]] internal method of O passing P as the argument.
4. ReturnIfAbrupt(desc).
5. If desc is undefined, return false.
6. Return true.

7.3.9 GetMethod (O, P)

The abstract operation GetMethod is used to get the value of a 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. Assert: Type(O) is Object.
2. Assert: IsPropertyKey(P) is true.
3. Let func be the result of calling the [[Get]] internal method of O passing P and O 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.

7.3.10 SetIntegrityLevel (O, level)

The abstract operation SetIntegrityLevel is used to fix the set of own properties of an object. This abstract operation performs the following steps:

1. Assert: Type(O) is Object.
2. Assert: level is either "sealed" or "frozen".
3. Let keys be the result of calling the [[OwnPropertyKeys]] internal method of O.
4. ReturnIfAbrupt(keys).
5. Let pendingException be undefined.
6. If level is "sealed", 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 level is "frozen", then
      a. Repeat for each element k of keys,
         i. Let status be the result of calling the [[GetProperty]] 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 the PropertyDescriptor([Configurable]: false).
               b. Else,
                  i. Let desc be the 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.

7.3.11 TestIntegrityLevel (O, level)

The abstract operation TestIntegrityLevel is used to determine if the set of own properties of an object are fixed. This abstract operation performs the following steps:

1. Assert: Type(O) is Object.
2. Assert: level is either "sealed" or "frozen".
3. Let status be the result of IsExtensible(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 the [[OwnPropertyKeys]] internal method of O.
8. ReturnIfAbrupt(keys).
9. Let pendingException be undefined.
10. Let configurable be false.
11. Let writable be false.
12. Repeat for each element k of keys,
    a. Let status be the result of calling the [[GetProperty]] 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]].
   13. If pendingException is not undefined, then return pendingException.
   14. If level is "frozen" and writable is true, then return false.
   15. If configurable is true, then return false.
   16. Return true.

7.3.12 CreateArrayFromList (elements)

The abstract operation CreateArrayFromList is used to create an Array object whose elements are provided by a 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 each element e of elements
   a. Let status be the result of CreateDataProperty(array, ToString(n), e).
   b. Assert: status is true.
   c. Increment n by 1.
5. Return array.

7.3.13 CreateListFromArrayLike (obj)

The abstract operation CreateListFromArrayLike is used to create a List value whose elements are provided by the indexed properties of an array-like object. This abstract operation performs the following steps:

1. If Type(obj) is not Object, then throw a TypeError exception.
2. Let len be the result of Get(obj, "length").
3. Let n be ToLength(len).
4. ReturnIfAbrupt(n).
5. Let list be an empty List.
6. Let index be 0.
7. Repeat while index < n
   a. Let indexName be ToString(index).
   b. Let next be the result of Get(obj, indexName).
   c. ReturnIfAbrupt(next).
   d. Append next as the last element of list.
   e. Set index to index + 1.
8. Return list.

7.3.14 OrdinaryHasInstance (C, O)

The abstract operation OrdinaryHasInstance implements the default algorithm for determining if an object O inherits from the instance object inheritance path provided 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 slot, then
   a. Let BC be the value of C’s [[BoundTargetFunction]] internal slot.
   b. Return the result of InstanceofOperator(O, BC) (see 12.8.4).
3. If Type(O) is not 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 [[GetPrototypeOf]] 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.

7.3.15 GetPrototypeOf (constructor, intrinsicDefaultProto)

The abstract operation GetPrototypeOf determines the [[Prototype]] value that should be used to create an object corresponding to a specific constructor. The value is retrieved from the constructor's prototype property, if it exists. Otherwise the supplied default is used for [[Prototype]]. This abstract operation performs the following steps:

1. Assert: intrinsicDefaultProto is a string value that is this specification's name of an intrinsic object. The corresponding object must be an intrinsic that is intended to be used as the [[Prototype]] value of an object.
2. If IsConstructor (constructor) is false, then throw a TypeError exception.
3. Let proto be the result of Get(constructor, "prototype").
4. ReturnIfAbrupt(proto).
5. If Type(proto) is not Object, then
   a. If constructor has a [[Realm]] internal slot, let realm be constructor's [[Realm]].
   b. Else,
      i. Let ctx be the running execution context.
      ii. Let realm be ctx's Realm.
   c. Let proto be realm's intrinsic object named intrinsicDefaultProto.
6. Return proto.

NOTE If constructor does not supply a [[Prototype]] value, the default value that is used is obtained from the Code Realm of the constructor function rather than from the running execution context. This accounts for the possibility that a built-in @@create method from a different Code Realm might be installed on constructor.

7.3.16 OrdinaryCreateFromConstructor (constructor, intrinsicDefaultProto, internalDataList)

The abstract operation OrdinaryCreateFromConstructor creates an ordinary object whose [[Prototype]] value is retrieved from a constructor's prototype property, if it exists. Otherwise the supplied default is used for [[Prototype]]. The optional internalDataList is a List of the names of internal slot names that should be defined as part of the object. If the list is not provided, an empty List is used. This abstract operation performs the following steps:

1. Assert: intrinsicDefaultProto is a string value that is this specification's name of an intrinsic object. The corresponding object must be an intrinsic that is intended to be used as the [[Prototype]] value of an object.
2. Let proto be the result of GetPrototypeOfFromConstructor(constructor, intrinsicDefaultProto).
3. ReturnIfAbrupt(proto).
4. Return the result of the abstract operation ObjectCreate(proto, internalDataList).

7.4 Operations on Iterator Objects

See Common Iteration Interfaces (25.1).

7.4.1 GetIterator (obj)

The abstract operation GetIterator with argument obj performs the following steps:

1. Let iterator be the result of performing Invoke with arguments obj, @@iterator and an empty List.
2. ReturnIfAbrupt(iterator).
3. If Type(iterator) is not Object, then throw a TypeError exception.
4. Return iterator.
7.4.2 IteratorNext (iterator, value)

The abstract operation IteratorNext with argument iterator and optional argument value performs the following steps:

1. If value was not passed, let value be undefined.
2. Let result be the result of Invoke(iterator, "next", (value)).
3. ReturnIfAbrupt(result).
4. If Type(result) is not Object, then throw a TypeError exception.
5. Return result.

7.4.3 IteratorComplete (iterResult)

The abstract operation IteratorComplete with argument iterResult performs the following steps:

1. Assert: Type(iterResult) is Object.
2. Let done be the result of Get(iterResult, "done").
3. Return ToBoolean(done).

7.4.4 IteratorValue (iterResult)

The abstract operation IteratorValue with argument iterResult performs the following steps:

1. Assert: Type(iterResult) is Object.
2. Return the result of Get(iterResult, "value").

7.4.5 IteratorStep (iterator, value)

The abstract operation IteratorStep with argument iterator and optional argument value requests the next value from iterator and returns either false indicating that the iterator has reached its end or the IteratorResult object if a next value is available. IteratorStep performs the following steps:

1. If value was not passed, let value be undefined.
2. Let result be the result of IteratorNext(iterator, value).
3. ReturnIfAbrupt(result).
4. Let done be the result of IteratorComplete(result).
5. ReturnIfAbrupt(done).
6. If done is true, then return false.
7. Return result.

7.4.6 CreateIterResultObject (value, done)

The abstract operation CreateIterResultObject with arguments value and done creates an object that supports the IteratorResult interface by performing the following steps:

1. Assert: Type(done) is Boolean.
2. Let obj be the result of performing ObjectCreate(%ObjectPrototype%).
3. Perform CreateDataProperty(obj, "value", value).
4. Perform CreateDataProperty(obj, "done", done).
5. Return obj.

7.4.7 CreateListIterator (list)

The abstract operation CreateListIterator with argument list creates an Iterator (25.1.2) object whose next method returns the successive elements of list. It performs the following steps:
1. Let iterator be the result of ObjectCreate(%ObjectPrototype%, ([%IteratedList%], [%ListIteratorNextIndex%])).
2. Set iterator’s [%IteratedList%] internal slot to list.
3. Set iterator’s [%ListIteratorNextIndex%] internal slot to 0.
4. Define ListIterator next (7.4.7.1) as an own property of iterator.
5. Return iterator.

7.4.7.1 ListIterator next()

The ListIterator next method is a standard built-in function object (clause 17) that performs the following steps:

1. Let O be the this value.
2. If O does not have a [%IteratedList%] internal slot, then throw a TypeError exception.
3. Let list be the List that is value of the [%IteratedList%] internal slot of O.
4. Let index be the value of the [%ListIteratorNextIndex%] internal slot of O.
5. Let lenValue be the number of elements of list.
6. If index ≥ len, then
   a. Return CreateIterResultObject(undefined, true).
7. Set the value of the [%ListIteratorNextIndex%] internal slot of O to index + 1.
8. Return CreateIterResultObject(list[index], false).

7.4.8 CreateEmptyIterator()

The abstract operation CreateEmptyIterator with no arguments creates an Iterator object whose next method always reports that the iterator is done. It performs the following steps:

1. Let empty be a List with no elements.
2. Return the result of CreateListIterator(empty).

8 Executable Code and Execution Contexts

8.1 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.

8.1.1 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 functions.

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 16. These abstract methods have distinct concrete algorithms for each of the concrete subclasses.
### Table 16 — Abstract Methods of Environment Records

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasBinding(N)</td>
<td>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.</td>
</tr>
<tr>
<td>CreateMutableBinding(N, D)</td>
<td>Create a new but uninitialised mutable binding in an environment record. The String value V is the text of the bound name. If the optional Boolean argument D is true the binding may be subsequently deleted.</td>
</tr>
<tr>
<td>CreateImmutableBinding(N)</td>
<td>Create a new but uninitialised immutable binding in an environment record. The String value N is the text of the bound name.</td>
</tr>
<tr>
<td>InitialiseBinding(N, V)</td>
<td>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.</td>
</tr>
<tr>
<td>SetMutableBinding(N, V, S)</td>
<td>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.</td>
</tr>
<tr>
<td>GetBindingValue(N, S)</td>
<td>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.</td>
</tr>
<tr>
<td>DeleteBinding(N)</td>
<td>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.</td>
</tr>
<tr>
<td>HasThisBinding()</td>
<td>Determine if an environment record establishes a this binding. Return true if it does and false if it does not.</td>
</tr>
<tr>
<td>HasSuperBinding()</td>
<td>Determine if an environment record establishes a super method binding. Return true if it does and false if it does not.</td>
</tr>
<tr>
<td>WithBaseObject()</td>
<td>If this environment record is associated with a with statement, return the with object. Otherwise, return undefined.</td>
</tr>
</tbody>
</table>

#### 8.1.1.1 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.

#### 8.1.1.1.1 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. Return false.

8.1.1.1.2 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 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(\( \emptyset \)).

8.1.1.1.3 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.

8.1.1.1.4 InitialiseBinding (N,V)

The concrete Environment Record method InitialiseBinding 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.

8.1.1.1.5 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 \( envRec \) has not yet been initialised throw a ReferenceError exception.
4. Else if the binding for \( N \) in \( envRec \) is a mutable binding, change its bound value to \( V \).
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(\( \emptyset \)).

8.1.1.1.6 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`.

8.1.1.1.7 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`.

8.1.1.1.8 HasThisBinding ()

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

1. Return `false`.

8.1.1.1.9 HasSuperBinding ()

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

1. Return `false`.

8.1.1.1.10 WithBaseObject()

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

1. Return `undefined`.

8.1.1.2 Object Environment Records

Each object environment record is associated with an object called its `binding object`. An object environment record binds the set of string identifier names that directly correspond to the property names of its binding object. Property keys that are not strings in the form of 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 also have a possibly empty List of strings called `unscopables`. The strings in this List are excluded from the environment records set of bound names, regardless of whether or not they exist as property keys of its binding object.

Object environment records created for `with` statements (13.10) 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.
8.1.1.2.1 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 \(\text{envRec}\) be the object environment record for which the method was invoked.
2. If \(N\) is an element of \(\text{envRec}'s\text{ unscopables}\), then return false.
3. Let \(\text{bindings}\) be the binding object for \(\text{envRec}\).
4. Return the result of HasProperty\((\text{bindings}, N)\).

8.1.1.2.2 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. 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 \(\text{envRec}\) be the object environment record for which the method was invoked.
2. Let \(\text{bindings}\) be the binding object for \(\text{envRec}\).
3. If \(D\) is true then let \(\text{configValue}\) be true otherwise let \(\text{configValue}\) be false.
4. Return the result of DefinePropertyOrThrow\((\text{bindings}, N, \text{PropertyDescriptor}\{[[\text{Value}]]: \text{undefined}, [[\text{Writable}]]: \text{true}, [[\text{Enumerable}]]: \text{true}, [[\text{Configurable}]]: \text{configValue}\})\).

NOTE Normally \(\text{envRec}\) will not have a binding for \(N\) but if it does, the semantics of DefinePropertyOrThrow may result in an existing binding being replaced or shadowed or cause an abrupt completion to be returned.

8.1.1.2.3 CreateImmutableBinding \((N)\)

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

8.1.1.2.4 InitialiseBinding \((N,V)\)

The concrete Environment Record method InitialiseBinding 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 \(\text{envRec}\) be the object environment record for which the method was invoked.
2. Assert: \(\text{envRec}\) must have an uninitialised binding for \(N\).
3. Record that the binding for \(N\) in \(\text{envRec}\) has been initialised.
4. Return the result of calling the SetMutableBinding concrete method of \(\text{envRec}\) with \(N, V, \text{and false}\) as arguments.

8.1.1.2.5 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 \(\text{envRec}\) be the object environment record for which the method was invoked.
2. Let \(\text{bindings}\) be the binding object for \(\text{envRec}\).
3. Return the result of Put\((\text{bindings}, N, V, \text{and } S)\).
8.1.1.2.6 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(\( bindings, N \)).

8.1.1.2.7 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.

8.1.1.2.8 HasThisBinding ()

Regular Object Environment Records do not provide a this binding.

1. Return false.

8.1.1.2.9 HasSuperBinding ()

Regular Object Environment Records do not provide a super binding.

1. Return false.

8.1.1.2.10 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.

8.1.1.3 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 16 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 17:
Table 17 — Additional Methods of Function Environment Records

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetThisBinding()</td>
<td>Return the value of this environment record’s this binding.</td>
</tr>
<tr>
<td>GetSuperBase()</td>
<td>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.</td>
</tr>
<tr>
<td>GetMethodName()</td>
<td>Return the value of this environment record’s MethodName binding.</td>
</tr>
</tbody>
</table>

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

8.1.1.3.1 HasThisBinding ()

Function Environment Records always provide a this binding.
1. Return true.

8.1.1.3.2 HasSuperBinding ()
1. If this environment record’s HomeObject has the value Empty, then return false. Otherwise, return true.

8.1.1.3.3 GetThisBinding ()
1. Return the value of this environment record’s thisValue.

8.1.1.3.4 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 [[GetPrototypeOf]] internal method.

8.1.1.3.5 GetMethodName ()
1. Return the value of this environment record’s MethodName.

8.1.1.4 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 (8.2). A global environment provides the bindings for built-in globals (clause 18), 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 (clause 18) 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 18 and the additional methods listed in Table 19.

**Table 18 -- Components of Global Environment Records**

<table>
<thead>
<tr>
<th>Component</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ObjectEnvironment</td>
<td>An 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.</td>
</tr>
<tr>
<td>DeclarativeEnvironment</td>
<td>A Declarative Environment Record that contains bindings for all declarations in global for the associated Realm code except for FunctionDeclaration and VariableStatement declarations.</td>
</tr>
<tr>
<td>VarNames</td>
<td>A List containing the string names bound by FunctionDeclaration or VariableStatement declarations in global code for the associated Realm.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetThisBinding()</td>
<td>Return the value of this environment record’s this binding.</td>
</tr>
<tr>
<td>HasVarDeclaration (N)</td>
<td>Determines if the argument identifier has a binding in this environment record that was created using a VariableStatement or a FunctionDeclaration.</td>
</tr>
<tr>
<td>HasLexicalDeclaration (N)</td>
<td>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.</td>
</tr>
<tr>
<td>CanDeclareGlobalVar (N)</td>
<td>Determines if a corresponding CreateGlobalVarBinding call would succeed if called for the same argument N.</td>
</tr>
<tr>
<td>CanDeclareGlobalFunction (N)</td>
<td>Determines if a corresponding CreateGlobalFunctionBinding call would succeed if called for the same argument N.</td>
</tr>
<tr>
<td>CreateGlobalVarBinding(N, D)</td>
<td>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 appropriate 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.</td>
</tr>
<tr>
<td>CreateGlobalFunctionBinding(N, V, D)</td>
<td>Used to create and initialise 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 appropriate 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.</td>
</tr>
</tbody>
</table>
The behaviour of the concrete specification methods for Global Environment Records is defined by the following algorithms.

8.1.1.4.1 \textbf{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 \texttt{true}, return \texttt{true}.
4. Let ObjRec be envRec’s ObjectEnvironment.
5. Return the result of calling ObjRec’s HasBinding concrete method with argument \( N \).

8.1.1.4.2 \textbf{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 \texttt{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. Return the result of calling the CreateMutableBinding concrete method of DclRec with arguments \( N \) and \( D \).

8.1.1.4.3 \textbf{CreateImmutableBinding (N)}

The concrete Environment Record method CreateImmutableBinding for global 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. Let DclRec be envRec’s DeclarativeEnvironment.
3. Assert: DclRec does not already have a binding for \( N \).
4. Return the result of calling the CreateImmutableBinding concrete method of DclRec with argument \( N \).

8.1.1.4.4 \textbf{InitialiseBinding (N,V)}

The concrete Environment Record method InitialiseBinding 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 \texttt{true}, then
   a. Return the result of calling DclRec’s InitialiseBinding concrete method with arguments \( N \) and \( V \).
4. Assert: If the binding exists it must be in the object environment record.
5. Let ObjRec be envRec’s ObjectEnvironment.
6. Return the result of calling ObjRec’s InitialiseBinding concrete method with arguments \( N \) and \( V \).

8.1.1.4.5 \textbf{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 \texttt{TypeError} is thrown if \( S \) is \texttt{true}. A property
8.1.1.4.4 GetValue (N, S)

The concrete Environment Record method GetValue 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 GetMutableBinding concrete method of DclRec with arguments N, V, and S.
4. Let ObjRec be envRec’s ObjectEnvironment.
5. Return the result of calling the GetMutableBinding concrete method of ObjRec with arguments N, V, and S.

8.1.1.4.6 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.

8.1.1.4.7 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 ObjRec 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.

8.1.1.4.8 HasThisBinding ()

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

1. Return true.

8.1.1.4.9 HasSuperBinding ()

1. Return false.

8.1.1.4.10 WithBaseObject()

Global Environment Records always return undefined as their WithBaseObject.

1. Return undefined.
8.1.1.4.11 GetThisBinding ()

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

8.1.1.4.12 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.

8.1.1.4.13 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.

8.1.1.4.14 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. Redundant 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. Let extensible be the result of IsExtensible(bindings).
6. Return extensible.

8.1.1.4.15 CanDeclareGlobalFunction (N)

The concrete environment record method CanDeclareGlobalFunction 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 IsExtensible(globalObject).
5. ReturnIfAbrupt(extensible).
6. If the result of calling ObjRec's HasBinding concrete method with argument N is false, then return extensible.
7. Let existingProp be the result of calling the [[GetOwnProperty]] internal method of globalObject with argument N.
8. If existingProp is undefined, then return extensible.
9. If existingProp.[[Configurable]] is true, then return true.
10. If IsDataDescriptor(existingProp) is true and existingProp has attribute values {[[Writable]]: true, [[Enumerable]]: true}, then return true.
11. Return false.

8.1.1.4.16 CreateGlobalVarBinding (N, D)
The concrete Environment Record method CreateGlobalVarBinding 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. If the result of calling ObjRec’s HasBinding concrete method with argument N is false, then
   a. Let status be the result of calling the CreateMutableBinding concrete method of ObjRec with arguments N and D.
   b. ReturnIfAbrupt(status).
4. Let varDeclaredNames be envRec’s VarNames List.
5. If varDeclaredNames does not contain the value of N, then
   a. Append N to varDeclaredNames.
6. Return NormalCompletion(empty).

8.1.1.4.17 CreateGlobalFunctionBinding (N, V, D)
The concrete Environment Record method CreateGlobalFunctionBinding 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. Let globalObject be the binding object for ObjRec.
4. Let existingProp be the result of calling the [[GetOwnProperty]] internal method of globalObject with argument N.
5. If existingProp is undefined or existingProp.[[Configurable]] is true, then
   a. Let desc be the PropertyDescriptor{[[Value]]: V, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: D}.
6. Else, a. Let desc be the PropertyDescriptor{[[Value]]: V}.
7. Let status be the result of DefinePropertyOrThrow(globalObject, N, desc).
8. ReturnIfAbrupt(status).
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 NormalCompletion(empty).

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

8.1.2 Lexical Environment Operations
The following abstract operations are used in this specification to operate upon lexical environments:

8.1.2.1 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. ReturnIfAbrupt(`exists`).
5. 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`.
6. 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.

8.1.2.2 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`.

8.1.2.3 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 `envRec`’s `unscopables` to an empty List.
4. Set `env`’s environment record to be `envRec`.
5. Set the outer lexical environment reference of `env` to `E`.
6. Return `env`.

8.1.2.4 NewFunctionEnvironment (F, T)
When the abstract operation NewFunctionEnvironment is called with an ECMAScript function Object `F` and an ECMAScript value `T` as arguments, the following steps are performed:

1. Assert: The value of `F`’s `[ThisMode]` internal slot is not lexical.
2. Let `env` be a new Lexical Environment.
3. Let `envRec` be a new Function environment record containing no bindings.
4. Set `envRec`’s `thisValue` to `T`.
5. If `F` has a `[HomeObject]` internal slot, then
   a. Set `envRec`’s `HomeObject` to the value of `F`’s `[HomeObject]` internal slot.
   b. Set `envRec`’s `MethodName` to the value of `F`’s `[MethodName]` internal slot.
6. Else,
   a. Set `envRec`’s `HomeObject` to `Empty`.
7. Set `env`’s environment record to be `envRec`.
8. Set the outer lexical environment reference of `env` to the value of `F`’s `[Scope]` internal slot.
8.2 Code Realms

Before it is evaluated, all ECMAScript code must be associated with a Realm. Conceptually, a realm consists of a 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 20:

Table 20 — Realm Record Fields

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

8.3 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 21.

Table 21 — State Components for All Execution Contexts

<table>
<thead>
<tr>
<th>Component</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>code evaluation state</td>
<td>Any state needed to perform, suspend, and resume evaluation of the code associated with this execution context.</td>
</tr>
<tr>
<td>Realm</td>
<td>The Realm from which associated code accesses ECMAScript resources.</td>
</tr>
</tbody>
</table>

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.

The value is the Realm component of the running execution context is also called the current Realm.

Execution contexts for ECMAScript code have the additional state components listed in Table 22.

Commented [AWB810]: Other possible terms that have been discussed are “Home” and “Island”. We still need to get final agreement on terminology.
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.

Execution contexts representing the evaluation of generator objects have the additional state components listed in Table 23.

<table>
<thead>
<tr>
<th>Component</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generator</td>
<td>The GeneratorObject that this execution context is evaluating.</td>
</tr>
</tbody>
</table>

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.

8.3.1 ResolveBinding(name)

The ResolveBinding abstract operation is used to determine the binding of name passed as a string value using the LexicalEnvironment of the running execution context. During execution of ECMAScript code, ResolveBinding 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, name, and strict as arguments.

The result of resolving a name is always a Reference value with its referenced name component equal to the name argument.

8.3.2 GetThisEnvironment

The abstract operation GetThisEnvironment finds the lexical environment that currently supplies the binding of the keyword this. GetThisEnvironment 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 2 will always terminate because the list of environments always ends with the global environment which has a this binding.

8.3.3 ResolveThisBinding

The abstract operation ResolveThisBinding is the process of determining the binding of the keyword this using the LexicalEnvironment of the running execution context. ResolveThisBinding 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.

8.3.4 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]].

9 Ordinary and Exotic Objects Behaviours

9.1 Ordinary Object Internal Methods and Internal Slots

All ordinary objects have an internal slot called [[Prototype]]. The value of this internal slot 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 object has a Boolean-valued [[Extensible]] internal slot that controls whether or not properties may be added to the object. If the value of the [[Extensible]] internal slot is false then additional properties may not be added to the object. In addition, if [[Extensible]] is false the value of [[Prototype]] internal slot of the object may not be modified. Once the value of an object’s [[Extensible]] internal slot has been set to false it may not be subsequently changed.

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

9.1.1 [[GetPrototypeOf]] ( )

When the [[GetPrototypeOf]] internal method of O is called the following steps are taken:
1. Return the value of the [[Prototype]] internal slot of O.

9.1.2 [[SetPrototypeOf]] ( V)

When the [[SetPrototypeOf]] 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 slot of O.
3. Let current be the value of the [[Prototype]] internal slot of O.
4. If SameValue(V, current), then return true.
5. If extensible is false, then return false.
6. If \( V \) is not \texttt{null}, then
   a. Let \( p = V \).
   b. Repeat, while \( p \) is not \texttt{null}
      i. If \( \text{SameValue}(p, O) \) is \texttt{true}, then return \texttt{false}.
      ii. Let \( \text{nextp} \) be the result of calling the \text{[[GetPrototypeOf]]} internal method of \( p \) with no arguments.
      iii. Return \text{[[HasAbrupt]]}(\text{nextp}).
      iv. Let \( p = \text{nextp} \).
    7. Set the value of the \text{[[Prototype]]} internal slot of \( O \) to \( V \).
   8. Return \texttt{true}.

\[9.1.3 \text{[[IsExtensible]]}()\]
When the \text{[[IsExtensible]]} internal method of \( O \) is called, the following steps are taken:
1. Return the value of the \text{[[Extensible]]} internal slot of \( O \).

\[9.1.4 \text{[[PreventExtensions]]}()\]
When the \text{[[PreventExtensions]]} internal method of \( O \) is called, the following steps are taken:
1. Set the value of the \text{[[Extensible]]} internal slot of \( O \) to \texttt{false}.
2. Return \texttt{true}.

\[9.1.5 \text{[[GetOwnProperty]]}(P)\]
When the \text{[[GetOwnProperty]]} internal method of \( O \) is called with property key \( P \), the following steps are taken:

Return the result of \text{OrdinaryGetOwnProperty} with arguments \( O \) and \( P \).

\[9.1.5.1 \text{OrdinaryGetOwnProperty}(O, P)\]
When the abstract operation \text{OrdinaryGetOwnProperty} is called with Object \( O \) and with property key \( P \), the following steps are taken:

1. Assert: \text{IsPropertyKey}(P) \text{ is } \texttt{true}.
2. If \( O \) does not have an own property with key \( P \), return \texttt{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 \( D.\text{[[Value]]} \) to the value of \( X \)'s \text{[[Value]]} attribute.
   b. Set \( D.\text{[[Writable]]} \) to the value of \( X \)'s \text{[[Writable]]} attribute.
6. Else \( X \) is an accessor property, so
   a. Set \( D.\text{[[Get]]} \) to the value of \( X \)'s \text{[[Get]]} attribute.
   b. Set \( D.\text{[[Set]]} \) to the value of \( X \)'s \text{[[Set]]} attribute.
7. Set \( D.\text{[[Enumerable]]} \) to the value of \( X \)'s \text{[[Enumerable]]} attribute.
8. Set \( D.\text{[[Configurable]]} \) to the value of \( X \)'s \text{[[Configurable]]} attribute.
9. Return \( D \).

\[9.1.6 \text{[[DefineOwnProperty]]}(P, Desc)\]
When the \text{[[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 \text{OrdinaryDefineOwnProperty} with arguments \( O \), \( P \), and \( Desc \).
9.1.6.1 OrdinaryDefineOwnProperty (O, P, Desc)

When the abstract operation OrdinaryDefineOwnProperty is called with Object O, property key P, and Property Descriptor 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 slot of O.
3. Return the result of ValidateAndApplyPropertyDescriptor with arguments O, P, extensible, Desc, and current.

9.1.6.2 IsCompatiblePropertyDescriptor (Extensible, Desc, Current)

When the abstract operation IsCompatiblePropertyDescriptor 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.

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

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 no object updates are performed.

1. Assert: If O is not undefined then P is a valid property key.
2. If current is undefined, then
   a. If extensible is false, then return false.
   b. Assert: extensible is true.
   c. 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.
   d. Else Desc must be an accessor Property Descriptor,
      i. If O is not undefined, then create an 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.
   e. Return true.
3. Return true, if every field in Desc is absent.
4. 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.
5. 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.
6. If IsGenericDescriptor(Desc) is true, then no further validation is required.
7. 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.

8. Else if IsDataDescriptor(\( current \)) and IsAccessorDescriptor(\( Desc \)) 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`.
   b. Else if IsDataDescriptor(\( parent \)) and IsAccessorDescriptor(\( Desc \)) are both `true`, then
      i. Return `false`, if the [[Writable]] field of \( current \) is `false`.

9. Else IsAccessorDescriptor(\( current \)) and IsAccessorDescriptor(\( Desc \)) 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`.
   b. Else if IsDataDescriptor(\( parent \)) and IsAccessorDescriptor(\( Desc \)) are both `true`, then
      i. Return `false`, if the [[Writable]] field of \( current \) is `false`.

10. 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.

11. Return `true`.

NOTE Step 8.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`.

9.1.7 `[[HasProperty]](P)`

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

1. `Assert: IsPropertyKey(P) is true.`
2. Let hasOwn be the result of calling the `[[GetOwnProperty]]` internal method of \( O \) with argument \( P \).
3. ReturnIfAbrupt(hasOwn).
4. If hasOwn is `undefined`, then
   a. Let `parent` be the result of calling the `[[GetPrototypeOf]]` internal method of \( O \).
   b. ReturnIfAbrupt(parent).
   c. If `parent` is not `null`, then
      i. Return the result of calling the `[[HasProperty]]` internal method of `parent` with argument \( P \).
5. Return hasOwn.

9.1.8 `[[Get]](P, Receiver)`

When the `[[Get]]` internal method of \( O \) is called with property key \( P \) and ECMAScript language value `Receiver` the following steps are taken:

1. `Assert: IsPropertyKey(P) is true.`
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 `[[GetPrototypeOf]]` internal method of \( O \).
   b. ReturnIfAbrupt(parent).
   c. If `parent` is `null`, then return `undefined`.
   d. Return the result of calling the `[[Get]]` internal method 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]]`.  

© Ecma International 2013
7. If \( \text{getter} \) is undefined, return undefined.
8. Return the result of calling the \([[\text{Call}] \text{internal method of } \text{getter} \text{ with } \text{Receiver} \text{ as the } \text{thisArgument} \text{ and an empty List as argumentsList}.\]

9.1.9 \([[\text{Set}] \ P, V, \text{Receiver}])

When the \([[\text{Set}] \text{internal method of } \text{O} \text{ is called with property key } P, \text{ value } V, \text{ and ECMAScript language value Receiver} \), the following steps are taken:

1. Assert: IsPropertyKey\((P)\) is \text{true}.
2. Let \(\text{ownDesc}\) be the result of calling the \([[\text{GetOwnProperty}] \text{internal method of } \text{O} \text{ with argument } P).\]
3. ReturnIfAbrupt\((\text{ownDesc}).\]
4. If \(\text{ownDesc}\) is undefined, then
   a. Let \(\text{parent}\) be the result of calling the \([[\text{GetPrototypeOf}] \text{internal method of } \text{O}).\]
   b. ReturnIfAbrupt\((\text{parent}).\]
   c. If \(\text{parent}\) is not null, then
      i. Return the result of calling the \([[\text{Set}] \text{internal method of } \text{parent} \text{ with arguments } P, V, \text{ and Receiver}).\]
   Else,
      i. Let \(\text{ownDesc}\) be the PropertyDescriptor{\[[\text{Value}]: \text{undefined}, \text{[Writable]}: \text{true},
      \text{[Enumerable]}: \text{true}, \text{[Configurable]}: \text{true}.\]
5. If IsPropertyDescriptor\((\text{ownDesc})\) is \text{true}, then
   a. If \(\text{ownDesc}.\text{[Writable]}\) is \text{false}, return \text{false}.
   b. If Type\((\text{Receiver})\) is not Object, return \text{false}.
   c. Let \(\text{existingDesc}\) be the result of calling the \([[\text{GetOwnProperty}] \text{internal method of } \text{Receiver} \text{ with argument } P).\]
   d. ReturnIfAbrupt\((\text{existingDesc}).\]
   e. If \(\text{existingDesc}\) is not undefined, then
      i. Let \(\text{valueDesc}\) be the PropertyDescriptor{\[[\text{Value}]: \text{V}.\]
      ii. Return the result of calling the \([[\text{DefineOwnProperty}] \text{internal method of } \text{Receiver} \text{ with arguments } P \text{ and valueDesc}).\]
      f. Else \(\text{Receiver}\) does not currently have a property \(P).\]
         i. Return the result of performing CreateDataProperty\((\text{Receiver}, P, V).\)
6. If IsAccessorDescriptor\((\text{ownDesc})\) is \text{true}, then
   a. Let \(\text{setter}\) be ownDesc.\text{[[Set]]}.\]
   b. If \(\text{setter}\) is undefined, return \text{false}.
   c. Let \(\text{setterResult}\) be the result of calling the \([[\text{Call}] \text{internal method of } \text{setter} \text{ providing } \text{Receiver} \text{ as } \text{thisArgument} \text{ and a new List containing } V \text{ as argumentsList}).\]
   d. ReturnIfAbrupt\((\text{setterResult}).\]
   e. Return \text{true}.

9.1.10 \([[\text{Delete}] \ P)\]

When the \([[\text{Delete}] \text{internal method of } \text{O} \text{ is called with property key } P \text{ the following steps are taken:}

1. Assert: IsPropertyKey\((P)\) is \text{true}.
2. Let \(\text{desc}\) be the result of calling the \([[\text{GetOwnProperty}] \text{internal method of } \text{O} \text{ with argument } P).\]
3. If \(\text{desc}\) is undefined, then return \text{true}.
4. If desc.\text{[Configurable]} is \text{true}, then
5. Remove the own property with name \(P\) from \text{O}.
6. Return \text{true}.
7. Return \text{false}.

9.1.11 \([[\text{Enumerate}])\]

When the \([[\text{Enumerate}] \text{internal method of } \text{O} \text{ is called the following steps are taken:}

1. Return an Iterator object (25.1.2) whose next method iterates over all the String valued keys of enumerable property keys of \text{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 [[GetPrototypeOf]] 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.
   b. 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.
   b. Order the elements of propList in an implementation defined order.
8. Return propList.

9.1.12 [[OwnPropertyKeys]] ( )

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

1. Let keys be a new empty List.
2. For each own property key P of O
   a. Add P as the last element of keys.
3. Return CreateListIterator(keys).

9.1.13 ObjectCreate(proto, internalDataList) Abstract Operation

The abstract operation ObjectCreate with argument proto (an object or null) is used to specify the runtime creation of new ordinary objects. The optional argument internalDataList is a List of the names of internal slot names that should be defined as part of the object. If the list is not provided, an empty List is used. If no arguments are provided, %ObjectPrototype% is used as the value of proto. This abstract operation performs the following steps:

1. If internalDataList was not provided, let internalDataList be an empty List.
2. Let obj be a newly created object with an internal slot for each name in internalDataList.
3. Set obj’s essential internal methods to the default ordinary object definitions specified in 9.1.
4. Set the [[Prototype]] internal slot of obj to proto.
5. Set the [[Extensible]] internal slot of obj to true.
6. Return obj.
9.2 ECMAScript Function Objects

ECMAScript function objects encapsulate parameterised ECMAScript code closed over a lexical environment and support the dynamic evaluation of that code. An ECMAScript function object is an ordinary object and has the same internal slots and (except as noted below) the same internal methods as other ordinary objects. The code of an ECMAScript function object may be either strict mode code (10.2.1) or non-strict mode code.

ECMAScript function objects have the additional internal slots listed in Table 24.

ECMAScript function objects whose code is not strict mode code (10.2.1) provide alternative definitions for the [[GetOwnProperty]] internal method. This alternatives prevents the value of strict mode function from being revealed as the value of a function object property named "caller". The alternative definition exist solely 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 ECMAScript function objects. ECMAScript function objects are considered to be ordinary objects even though they may use the alternative definition of [[GetOwnProperty]].

Table 24 -- Internal Slots of ECMAScript Function Objects

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Scope]]</td>
<td>Lexical Environment</td>
<td>The Lexical Environment that the function was closed over. Used as the outer environment when evaluating the code of the function.</td>
</tr>
<tr>
<td>[[FormalParameters]]</td>
<td>Parse Node</td>
<td>The root parse node of the source code that defines the function's formal parameter list.</td>
</tr>
<tr>
<td>[[FunctionKind]]</td>
<td>String</td>
<td>Either &quot;normal&quot; or &quot;generator&quot;.</td>
</tr>
<tr>
<td>[[Code]]</td>
<td>Parse Node</td>
<td>The root parse node of the source code that defines the function's body.</td>
</tr>
<tr>
<td>[[Realm]]</td>
<td>Realm Record</td>
<td>The Code Realm in which the function was created and which provides any intrinsic objects that are accessed when evaluating the function.</td>
</tr>
<tr>
<td>[[ThisMode]]</td>
<td>(lexical, strict, global)</td>
<td>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 the this value of undefined is interpreted as a reference to the global object.</td>
</tr>
<tr>
<td>[[Strict]]</td>
<td>Boolean</td>
<td>true if this is a strict mode function, false if this is not a strict mode function.</td>
</tr>
<tr>
<td>[[HomeObject]]</td>
<td>Object</td>
<td>If the function uses super, this is the object whose [[GetPrototypeOf]] provides the object where super property lookups begin. Not present for functions that don't reference super.</td>
</tr>
<tr>
<td>[[MethodName]]</td>
<td>String or Symbol</td>
<td>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.</td>
</tr>
</tbody>
</table>

All ECMAScript function objects have the [[Call]] internal method defined here. ECMAScript functions that are also constructors in addition have the [[Construct]] internal method. ECMAScript function objects whose code is not strict mode code have the [[Get]] and [[GetOwnProperty]] internal methods defined here.

9.2.1 [[Call]] (thisArgument, argumentsList)

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

1. If F's [[Code]] internal slot has the value undefined, then throw a TypeError exception.
2. Let callerContext be the running execution context.
3. If callerContext is not already suspended, then Suspend callerContext.
4. Let callerContext be a new ECMAScript Code execution context.
5. Set calleeRealm be the value of F’s [[Realm]] internal slot.
6. Set calleeContext’s Realm to calleeRealm.
7. Let thisMode be the value of F’s [[ThisMode]] internal slot.
8. If thisMode is lexical, then
   a. Let localEnv be the result of calling NewDeclarativeEnvironment passing the value of the [[Scope]] internal slot of F as the argument.
9. Else, then
   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.
10. Set the LexicalEnvironment of calleeContext to localEnv.
11. Set the VariableEnvironment of calleeContext to localEnv.
12. Push calleeContext onto the execution context stack; calleeContext is now the running execution context.
13. Let status be the result of performing Function Declaration Instantiation using the function F, argumentsList, and localEnv as described in 9.2.13.
14. 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.
15. Let result be the result of EvaluateBody of the production that is the value of F’s [[Code]] internal slot passing F as the argument.
16. Remove calleeContext from the execution context stack and restore callerContext as the running execution context.
17. Return result.

NOTE 1 Most ECMAScript functions use a Function Environment Record as their LexicalEnvironment. ECMAScript functions that are arrow functions use a Declarative Environment Record as their LexicalEnvironment.

NOTE 2 When calleeContext is removed from the execution context stack it must not be destroyed because it may have been suspended and retained by a generator object for later resumption.

9.2.2 [[Construct]] ( argumentsList)

The [[Construct]] internal method for an ECMAScript 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. Return the result of OrdinaryConstruct(F, argumentsList).

9.2.2.1 OrdinaryConstruct (F, argumentsList)

When the abstract operation OrdinaryConstruct is called with Object F and List argumentsList the following steps are taken:

1. Let creator be the result of Get(F, @0create).
2. ReturnIfAbrupt(creator).
3. If creator is not undefined, then
   a. If IsCallable(creator) is false, then throw a TypeError exception.
   b. Let obj be the result of calling the [[Call]] internal method of creator with arguments F and an empty List.
4. Else creator is undefined so fall back to object creation defaults.
9.2.3 [[GetOwnProperty]] (P)

When the [[GetOwnProperty]] internal method of non-strict ECMAScript 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 (9.1.5) 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 null.
4. Return v.

If an implementation does not provide a built-in caller property for non-strict ECMAScript function objects then it must not use this definition. Instead the ordinary object [[GetOwnProperty]] internal method is used.

9.2.4 FunctionAllocate Abstract Operation

The abstract operation FunctionAllocate requires the two arguments functionPrototype and strict. It also accepts one optional argument, functionKind. FunctionAllocate performs the following steps:

1. Assert: Type(functionPrototype) is Object.
2. Assert: If functionKind is present, its value is either "normal" or "generator".
3. If functionKind is not present, then let functionKind be "normal".
4. Let F be a newly created ECMAScript function object with the internal slots listed in Table 24. All of those internal slots are initialised to undefined.
5. Set F’s essential internal methods except for [[GetOwnProperty]] to the default ordinary object definitions specified in 9.1.
6. If strict is true, set F’s [[GetOwnProperty]] internal methods to the default ordinary object definitions specified in 9.1.
7. Else, set F’s [[GetOwnProperty]] internal methods to the definitions specified in 9.2.3.
8. Set F’s [[Call]] internal method to the definition specified in 9.2.1.
9. Set the [[Strict]] internal slot of F to strict.
10. Set the [[FunctionKind]] internal slot of F to functionKind.
11. Set the [[Prototype]] internal slot of F to functionPrototype.
12. Set the [[Extensible]] internal slot of F to true.
13. Set the [[Realm]] internal slot of F to the running execution context’s Realm.
14. Return F.

9.2.5 FunctionInitialise Abstract Operation

The abstract operation FunctionInitialise requires the arguments: a function object F, kind which is one of (Normal, Method, Arrow), a parameter list production specified by ParameterList, a body production specified by Body, a Lexical Environment specified by Scope, and optionally, a property key methodName and an object homeObject. FunctionInitialise performs the following steps:

1. Let len be the ExpectedArgumentCount of ParameterList.
2. Let strict be the value of F’s [[Strict]] internal slot.
3. Let status be the result of DefinePropertyOrThrow(F, "length", PropertyDescriptor{[[Value]]: len, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true}).
4. ReturnIfAbrupt(status).
5. If strict is true, then
   a. Let status be the result of the AddRestrictedFunctionProperties abstract operation with argument F.
   b. ReturnIfAbrupt(status).
6. Set the [[Scope]] internal slot of F to the value of Scope.
7. Set the [[NormalParameters]] internal slot of F to ParameterList.
8. Set the [[Code]] internal slot of F to Body.
9. If the homeObject argument was provided, then
   a. Set the [[HomeObject]] internal slot of F to homeObject.
   b. Set the [[MethodName]] internal slot of F to methodName.
10. If kind is Arrow, then set the [[ThisMode]] internal slot of F to lexical.
11. Else if strict is true, then set the [[ThisMode]] internal slot of F to strict.
12. Else set the [[ThisMode]] internal slot of F to global.
13. Return F.

9.2.6 FunctionCreate Abstract Operation

The abstract operation FunctionCreate requires the arguments: kind which is one of (Normal, Method, Arrow), a parameter list production specified by ParameterList, a body production specified by Body, a Lexical Environment specified by Scope, a Boolean flag Strict, and optionally, an object functionPrototype, a property key methodName and an object homeObject. FunctionCreate performs the following steps:

1. If the functionPrototype argument was not passed, then
   a. Let functionPrototype be the intrinsic object %FunctionPrototype%.
2. Let F be the result of performing FunctionAllocate with arguments functionPrototype and Strict.
3. Return the result of performing FunctionInitialise with passing F, kind, ParameterList, Body, and Scope. Also pass methodName and homeObject if they are present.

9.2.7 GeneratorFunctionCreate Abstract Operation

The abstract operation GeneratorFunctionCreate requires the arguments: kind which is one of (Normal, Method, Arrow), a parameter list production specified by ParameterList, a body production specified by Body, a Lexical Environment specified by Scope, a Boolean flag Strict, and optionally, an object functionPrototype, a property key methodName and an object homeObject. GeneratorFunctionCreate performs the following steps:

1. If the functionPrototype argument was not passed, then
   a. Let functionPrototype be the intrinsic object %Generator%.
2. Let F be the result of performing FunctionAllocate with arguments functionPrototype, Strict, and "generator".
3. Return the result of performing FunctionInitialise with passing F, kind, ParameterList, Body, and Scope. Also pass methodName and homeObject if they are present.

9.2.8 AddRestrictedFunctionProperties Abstract Operation

The abstract operation AddRestrictedFunctionProperties is called with a function object F as its argument. It performs the following steps:

1. Let thrower be the %TypeError% intrinsic function Object.
2. Let status be the result of DefinePropertyOrThrow(F, "caller", PropertyDescriptor {
   [[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: false}).
3. ReturnIfAbrupt(status).
4. Return the result of DefinePropertyOrThrow(F, "arguments", PropertyDescriptor {
   [[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: false}).

The %TypeError% object is a unique function object that is defined once for each Realm as follows:

1. Assert: %FunctionPrototype% for the current Realm has already been initialised.
2. Let functionPrototype be the intrinsic object %FunctionPrototype%.
3. Let scope be the Global Environment.
4. Let formalParameters be the syntactic production: FormalParameters: (empty).
5. Let body be the syntactic production: FunctionBody: ; ThrowTypeError.
6. Let F be the result of performing FunctionAllocate with arguments functionPrototype and true.
7. Let %ThrowTypeError% be F.
8. Perform the abstract operation FunctionInitialize with arguments F, Normal, formalParameters, body, and scope.
9. Call the [[PreventExtensions]] internal method of F.
10. Return F.

9.2.9 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 assumed to already contain, if needed, a "constructor" property whose value is F. This operation converts F into a constructor by performing the following steps:

1. Assert: F is an ECMAScript function object.
2. Let installNeeded be false.
3. If the prototype argument was not provided, then
   Let installNeeded be true.
   Let prototype be the result of ObjectCreate(%ObjectPrototype%).
4. If the writablePrototype argument was not provided, then
   a. Let writablePrototype be true.
5. Set F's essential internal method [[Construct]] to the definition specified in 9.2.2.
6. If installNeeded, then
   a. Let status be the result of DefinePropertyOrThrow prototype, "constructor", PropertyDescriptor{[[Value]]: F, [[Writable]]: writablePrototype, [[Enumerable]]: false, [[Configurable]]: writablePrototype}.
   b. ReturnIfAbrupt(status).
7. Let status be the result of DefinePropertyOrThrow F, "prototype", and PropertyDescriptor{[[Value]]: prototype , [[Writable]]: writablePrototype, [[Enumerable]]: false, [[Configurable]]: false}.
8. ReturnIfAbrupt(status).
9. Return NormalCompletion(undefined).

9.2.10 SetFunctionName Abstract Operation

The abstract operation SetFunctionName requires a Function argument F, a String or Symbol argument name and optionally a String argument prefix. This operation adds a name property to F by performing the following steps:

1. Assert: F is an extensible ECMAScript function object, that does not have a name own property.
2. Assert: Type(name) is either Symbol or String.
3. If Type(name) is Symbol, then
   a. Let description be the values of name's [[Description]].
   b. If description is undefined, then let name be the empty String.
   c. Else, let name be the concatenation of " [", description, and "] ".
4. If name is not the empty string and prefix was passed, then let name be the concatenation of prefix, Unicode code point U+0020 (Space), and name.
5. Call the [[DefineOwnProperty]] internal method of F with arguments "name" and PropertyDescriptor{[[Value]]: name, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true}.
6. Assert: Defining the name property will always succeed.
7. Return NormalCompletion(undefined).

9.2.11 GetSuperBinding(obj) Abstract Operation

The abstract operation GetSuperBinding is called with obj as its argument. It performs the following steps:

1. If Type(obj) is not Object, then return undefined.

© Ecma International 2009 – All rights reserved

Commented [AWB1913]: Same as ES5
2. If `obj` does not have a [[HomeObject]] internal slot, then return `undefined`.
3. Return the value of `obj`’s [[HomeObject]] internal slot.

### 9.2.12 RebindSuper(function, newHome) Abstract Operation

The abstract operation RebindSuper is called with a function object `function` and an object `newHome` as its argument. It performs the following steps:

1. Assert: `function` is an ECMAScript function object that has a [[HomeObject]] internal slot.
2. Assert: Type(`newHome`) is Object.
3. Let `new` be a new ECMAScript function object that has all of the same internal methods and internal slots as `function`.
4. Set the value of each of `new`’s internal slots, except for `[[HomeObject]]` to the value of `function`’s corresponding internal slot.
5. Set the value of `new`’s [[HomeObject]] internal slot to `newHome`.
6. Return `new`.

**NOTE** The purpose of this abstract operation is to create a new function object that is identical to the argument object in all ways except for its identity and the value of its [[HomeObject]] internal slot.

### 9.2.13 Function Declaration Instantiation

This version reflects the concensus as of the Sept. 2012 TC39 meeting. A concensus for a new semantics was reach at the Sept. 2013 meeting. This specification has not yet been update.

**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 initialised as part of this process. All other bindings are initialised during execution of the function code.

**Function Declaration Instantiation**

1. Let `code` be the value of the [[Code]] internal slot of `func`.
2. Let `strict` be the value of the [[Strict]] internal slot of `func`.
3. Let `formals` be the value of the [[FormalParameters]] internal slot of `func`.
4. Let `parameterNames` be the BoundNames of `formals`.
5. Let `varDeclarations` be the VarScopedDeclarations of `code`.
6. Let `functionsToInitialise` be an empty List.
7. If the value of the [[ThisMode]] internal slot of `func` is [`lexical`], then
   a. Let `argumentsObjectNeeded` be `false`.
8. Else, let `argumentsObjectNeeded` be `true`.
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 “`arguments`”, then let `argumentsObjectNeeded` be `false`.
      iv. If `alreadyDeclared` is the result of calling `env`’s HasBinding concrete method passing `fn` as the argument.
      v. If `alreadyDeclared` is `false`, then
1. Let \( \text{status} \) be the result of calling \( env \)'s CreateMutableBinding concrete method passing \( fn \) as the argument.
2. Assert: \( \text{status} \) is never an abrupt completion.
3. Append \( d \) to functionsToInitialise.

10. For each String \( \text{paramName} \) in \( \text{parameterNames} \), do
    a. Let \( \text{alreadyDeclared} \) be the result of calling \( env \)'s HasBinding concrete method passing \( \text{paramName} \) 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 initialised to undefined.
    c. If \( \text{alreadyDeclared} \) is false, then
        i. If \( \text{paramName} \) is "arguments", then let \( \text{argumentsObjectNeeded} \) be false.
        ii. Let \( \text{status} \) be the result of calling \( env \)'s CreateMutableBinding concrete method passing \( \text{paramName} \) as the argument.
        iii. Assert: \( \text{status} \) is never an abrupt completion
        iv. Call \( env \)'s InitialiseBinding concrete method passing \( \text{paramName} \), and undefined as the arguments.

11. NOTE If there is a function declaration or formal parameter with the name "arguments" then an argument object is not created.

12. If argumentsObjectNeeded is true, then
    a. If \( \text{strict} \) is true, then
        i. Call \( env \)'s CreateImmutableBinding concrete method passing the String "arguments" as the argument.
    b. Else,
        i. Let \( \text{status} \) be the result of calling \( env \)'s CreateMutableBinding concrete method passing the String "arguments" as the argument.
        ii. Assert: \( \text{status} \) is never an abrupt completion.

13. Let \( \text{varNames} \) be the VarDeclaredNames of \( \text{code} \).

14. For each String \( \text{varName} \) in \( \text{varNames} \), in list order do
    a. Let \( \text{alreadyDeclared} \) be the result of calling \( env \)'s HasBinding concrete method passing \( \text{varName} \) as the argument.
    b. NOTE A VarDeclaredNames is only instantiated and initialised here if it is not also the name of a formal parameter or a FunctionDeclarations.
    c. If \( \text{alreadyDeclared} \) is false, then
        i. Let \( \text{status} \) be the result of calling \( env \)'s CreateMutableBinding concrete method passing \( \text{varName} \) as the argument.
        ii. Assert: \( \text{status} \) is never an abrupt completion.

15. Let \( \text{lexDeclarations} \) be the LexicalDeclarations of \( \text{code} \).

16. For each element \( d \) in \( \text{lexDeclarations} \) do
    a. NOTE A lexically declared name cannot be the same as a function declaration, formal parameter, or a var name. Lexically declared names are only instantiated here but not initialised.
    b. For each element \( d \) of the BoundNames of \( d \) do
        i. If \( \text{IsConstantDeclaration} \) of \( d \) is true, then
            1. Call \( env \)'s CreateImmutableBinding concrete method passing \( dn \) as the argument.
            2. Assert: \( \text{status} \) is never an abrupt completion.
        ii. Else, 
            1. Let \( \text{status} \) be the result of calling \( env \)'s CreateMutableBinding concrete method passing \( dn \) and false as the arguments.
            2. Assert: \( \text{status} \) is never an abrupt completion.
    c. If \( d \) is a GeneratorDeclaration production, then
        i. Append \( d \) to functionsToInitialise.

17. For each production \( f \) in \( \text{functionsToInitialise} \), 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. Let \( status \) be the result of calling \( env \)'s SetMutableBinding concrete method passing \( fn, fo \), and false as the arguments.
    d. Assert: \( \text{status} \) is never an abrupt completion.

18. NOTE Function declaration are initialised prior to parameter initiation so that default value expressions may reference them. "arguments" is not initialised until after parameter initialisation.

19. Let formalStatus be the result of performing IteratorBindingInitialisation for \( \text{formals} \) with CreateListIterator(\( \text{argumentsList} \)) and undefined as arguments.
20. ReturnIfAbrupt(formalStatus).
21. If `argumentsObjectNeeded` is `true`, then
   a. Let `ao` be the result of `InstantiateArgumentsObject` with argument `argumentsList`.
   b. If `strict` is `true`, then
      i. Perform the abstract operation `CompleteStrictArgumentsObject` with argument `ao`.
   c. Else,
      i. Perform the abstract operation `CompleteMappedArgumentsObject` with arguments `ao`, `func`, `formals`, and `env`.
   d. Call `env`’s `InitialiseBinding` concrete method passing `"arguments"` and `ao` as arguments.
22. Return `NormalCompletion(empty)`.

9.3 Built-in Function Objects

The built-in function objects defined in this specification may be implemented as either ECMAScript function objects (9.2) whose behaviour is provided using ECMAScript code or as implementation provided exotic function objects whose behaviour is provided in some other manner. In either case, the effect of calling such functions must conform to their specifications.

If a built-in function object is implemented as an exotic object it must have the ordinary object behaviour specified in 9.1 except `[[GetOwnProperty]]` which must be as specified in 9.2. All such exotic function objects also have `[[Prototype]]` and `[[Extensible]]` internal slots.

Unless otherwise specified every built-in function object initially has `%FunctionPrototype%` object (19.2.3) as the initial value of its `[[Prototype]]` internal slot.

The behaviour specified for each built-in function via algorithm steps or other means is the specification of the `[[Call]]` behaviour for that function with the `[[Call]]` `thisArgument` providing the `this` value and the `[[Call]]` `argumentsList` providing the named parameters for each built-in function. If the built-in function is implemented as an ECMAScript function object then this specified behaviour must be implemented by the ECMAScript code that is the body of the function. Built-in functions that are ECMAScript function objects must be strict mode functions.

Built-in function objects that are not identified as constructors do not implement the `[[Construct]]` internal method unless otherwise specified in the description of a particular function. When a built-in constructor is called as part of a `new` expression the `argumentsList` parameter of the invoked `[[Construct]]` internal method provides the values for the built-in constructor’s named parameters.

Built-in functions that are not constructors do not have a `prototype` property unless otherwise specified in the description of a particular function.

If a built-in function object is not implemented as an ECMAScript function it must have a `[[Realm]]` internal slot and a `[[Call]]` internal method that conforms to the following definition:

9.3.1 `[[Call]]` ( `thisArgument`, argumentsList)

The `[[Call]]` internal method for a built-in 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 execution context.
4. Let `calleeRealm` be the value of `F`’s `[[Realm]]` internal slot.
5. Set `calleeContext`’s `Realm` to `calleeRealm`.
6. Perform any necessary implementation defined initialisation of `calleeContext`.
7. Push `calleeContext` onto the execution context stack; `calleeContext` is now the running execution context.
8. Let result be the Completion Record that is the result of evaluating \( F \) in an implementation defined manner that conforms to this specification of \( F \).

9. Remove calleeContext from the execution context stack and restore callerContext as the running execution context.

10. Return result.

**NOTE 1** When calleeContext is removed from the execution context stack it must not be destroyed because it may have been suspended and retained by a generator object for later resumption.

### 9.3.2 CreateBuiltinFunction Abstract Operation

The abstract operation CreateBuiltinFunction takes a single argument, \( \text{steps} \), that is a list of algorithm steps. It returns a built-in function object created by following steps:

1. Let \( \text{func} \) be a new built-in function object, in the current Realm, that when called performs the action described by \( \text{steps} \).
2. Perform the AddRestrictedFunctionProperties (9.2.8) abstract operation with argument \( \text{func} \).
3. Return \( \text{func} \).

### 9.4 Built-In Exotic Object Internal Methods and Data Fields

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

#### 9.4.1 Bound Function Exotic Objects

A *bound function* is an exotic object that wraps another function object. A bound function is callable (it has a \([\text{Call}]\) internal method and may have a \([\text{Construct}]\) internal method). Calling a bound function generally results in a call of its wrappered function.

Bound function objects do not have the internal slots of ECMAScript function objects defined in Table 24. Instead they have the internal slots defined in Table 25.

**Table 25 – Internal Slots of Exotic Bound Function Objects**

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{BoundTargetFunction}])</td>
<td>Callable Object</td>
<td>The wrappered function object.</td>
</tr>
<tr>
<td>([\text{BoundThis}])</td>
<td>Any</td>
<td>The value that is always passed as the \text{this} value when calling the wrappered function.</td>
</tr>
<tr>
<td>([\text{BoundArguments}])</td>
<td>List of Any</td>
<td>A list of values whose elements are used as the first arguments to any call to the wrappered function.</td>
</tr>
</tbody>
</table>

Unlike ECMAScript function objects, bound function objects do not use alternative definitions of the \([\text{Get}]\) and \([\text{GetOwnProperty}]\) internal methods. Bound function objects provide all of the essential internal methods as specified in 9.1. However, they use the following definitions for the essential internal methods of function objects.

#### 9.4.1.1 \([\text{Call}]\)

When the \([\text{Call}]\) internal method of an exotic bound function object, \( F \), which was created using the bind function is called with parameters \text{this}Argument and \text{argumentsList}, a List of ECMAScript language values, the following steps are taken:

1. Let \( \text{boundArgs} \) be the value of \( F \)'s \([\text{BoundArguments}]\) internal slot.
2. Let \( \text{boundThis} \) be the value of \( F \)'s \([\text{BoundThis}]\) internal slot.
3. Let \( \text{target} \) be the value of \( F \)'s \([\text{BoundTargetFunction}]\) internal slot.
4. Let \( \text{args} \) be a new list containing the same values as the list \( \text{boundArgs} \) in the same order followed by the same values as the list \( \text{argumentsList} \) in the same order.

Commented [AWB1616]: Perhaps this needs to be elaborated.
5. Return the result of calling the [[Call]] internal method of target providing boundThis as thisArgument and providing args as argumentsList.

9.4.1.2 [[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 slot.
2. Assert: $target$ has a [[Construct]] internal method.
3. Let $boundArgs$ be the value of $F$'s [[BoundArguments]] internal slot.
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.

9.4.1.3 BoundFunctionCreate Abstract Operation

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

1. Let $proto$ be the intrinsic %FunctionPrototype%.
2. Let $obj$ be a newly created object.
3. Set $obj$'s essential internal methods to the default ordinary object definitions specified in 9.1.
4. Set the [[Call]] internal method of $obj$ as described in 9.4.1.1.
5. If $targetFunction$ has a [[Construct]] internal method, then
   a. Set the [[Construct]] internal method of $obj$ as described in 9.4.1.2.
6. Set the [[Prototype]] internal slot of $obj$ to $proto$.
7. Set the [[Extensible]] internal slot of $obj$ to true.
8. Set the [[BoundTargetFunction]] internal slot of $obj$ to $targetFunction$.
9. Set the [[BoundThis]] internal slot of $obj$ to the value of $boundThis$.
10. Set the [[BoundArguments]] internal slot of $obj$ to $boundArgs$.
11. Return $obj$.

9.4.2 Array Exotic Objects

An Array object is an exotic object that gives 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 ToUint32(ToStr($P$)) 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 have the same internal slots as ordinary objects. They also have an [[ArrayInitialisationState]] internal slot.

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

Exotic Array objects provide an alternative definition for the [[DefineOwnProperty]] internal method. Except for that internal method, exotic Array objects provide all of the other essential internal methods as specified in 9.1.
9.4.2.1 [[DefineOwnProperty]] (P, Desc)

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

1. Assert: IsPropertyKey(P) is true.
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
   b. 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.
   c. Let oldLen be oldLenDesc.([[Value]].
   d. Let index be ToUint32(P).
   e. Assert: index will never be an abrupt completion.
   f. If index ≥ oldLen and oldLenDesc.([[Writable]]) is false, then return false.
   g. Let succeeded be the result of calling OrdinaryDefineOwnProperty passing A, P, and Desc as arguments.
   h. ReturnIfAbrupt(succeeded).
   i. If succeeded is false, then return false.
   j. 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).
   k. Return true.
4. Return the result of calling OrdinaryDefineOwnProperty passing A, P, and Desc as arguments.

9.4.2.2 ArrayCreate(length) Abstract Operation

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

2. If the proto argument was not passed, then let proto be the intrinsic object %ArrayPrototype%.
3. Let A be a newly created Array exotic object.
4. Set A’s essential internal methods except for [[DefineOwnProperty]] to the default ordinary object definitions specified in 9.1.
5. Set the [[DefineOwnProperty]] internal method of A as specified in 9.4.2.1.
6. Set the [[Prototype]] internal slot of A to proto.
7. Set the [[Extensible]] internal slot of A to true.
8. If length is not undefined, then
   a. Set the [[ArrayInitialisationState]] internal slot of A to true.
   b. Let length be 0.
10. If length+232-1, then throw a RangeError exception.
11. Call OrdinaryDefineOwnProperty with arguments A, "length", and PropertyDescriptor{[[Value]]: length, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false}.
12. Return A.

9.4.2.3 ArraySetLength(A, Desc) Abstract Operation

When the abstract operation ArraySetLength is called with an exotic Array object A, and Property Descriptor Desc the following steps are taken:

2. If the [[Value]] field of Desc is absent, then
   a. Return the result of calling OrdinaryDefineOwnProperty passing A, "length", and Desc as arguments.
3. Let newLenDesc be a copy of Desc.
4. Let `newLen` be `ToUint32Desc([Value]])`.
5. If `newLen` is not equal to `ToNumber(Desc.[Value])`, throw a `RangeError` exception.
7. 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.
9. If `newLen` ≥ `oldLen`, then
   a. Return the result of calling `OrdinaryDefineOwnProperty passing A, "length", and `newLenDesc` as arguments.
   b. If `oldLenDesc.[Writable]` is `false`, then return `false`.
   c. Otherwise, set `newWritable` to `true`.
10. If `oldLenDesc.[Writable]` is `false`, then set `newLenDesc.[Writable]` to `false`.
12. If `newWritable` is `false`, then set `newLenDesc.[Writable]` to `false`.
13. Let `succeeded` be the result of calling `OrdinaryDefineOwnProperty passing A, "length", and `newLenDesc` as arguments.
14. ReturnIfAbrupt(`succeeded`).
15. If `succeeded` is `false`, return `false`.
16. 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(`deleteSucceeded`).
   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`.
   e. Otherwise, set `newLenDesc.[Value]` to `oldLen+1`.
   f. Return `false`.
17. If `newWritable` is `false`, then
   a. Call `OrdinaryDefineOwnProperty` passing `A, "length", and PropertyDescriptor{[Writable]: false}` as arguments. This call will always return `true`.
18. Return `true`.

### 9.4.3 String Exotic Objects

A `String` object is an exotic object that encapsulates a `String` value and exposes virtual integer indexed 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 slots as ordinary objects. They also have a `[StringData]` internal slot.

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 9.1.

#### 9.4.3.1 `[[GetOwnProperty]] (P)`

When the `[[GetOwnProperty]]` internal method of an exotic `String` object `S` is called with property key `P` the following steps are taken:
1. Assert: IsPropertyKey(P) is true.
2. Let desc be the result of OrdinaryGetOwnProperty(S, P).
3. ReturnIfAbrupt(desc).
4. If desc is not undefined return desc.
5. If Type(P) is not String, then return undefined.
6. Let index be ToInteger(P).
7. Assert: index is not an abrupt completion.
8. Let absIntIndex be ToString(abs(index)).
9. If SameValue(absIntIndex, P) is false return undefined.
10. Let str be the String value of the [[StringData]] internal slot of S, if the value of [[StringData]] is undefined the empty string is used as its value.
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 PropertyDescriptor{ [[Value]]: resultStr, [[Enumerable]]: true, [[Writable]]: false, [[Configurable]]: false }.

9.4.3.2 [[DefineOwnProperty]] ( P, Desc)

When the [[DefineOwnProperty]] internal method of an exotic String object O is called with property key 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 value of the [[Extensible]] internal slot 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.

9.4.3.3 [[Enumerate]] ()

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

9.4.3.4 [[OwnPropertyKeys]] ()

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

9.4.3.5 StringCreate Abstract Operation

The abstract operation StringCreate with argument prototype is used to specify the creation of new exotic String objects. It performs the following steps:

1. Let A be a newly created String exotic object.
2. Set A’s essential internal methods to the default ordinary object definitions specified in 9.1.
3. Set the [[DefineOwnProperty]] internal method of A as specified in 9.4.3.1.
4. Set the [[DefineOwnProperty]] internal method of A as specified in 9.4.3.2.
5. Set the [[Enumerate]] internal method of A as specified in 9.4.3.3.
6. Set the [[OwnPropertyKeys]] internal method of A as specified in 9.4.3.4.
7. Set the [[Prototype]] internal slot of A to prototype.
8. Set the [[Extensible]] internal slot of A to true.
9. Return A.

9.4.4 Arguments Exotic Objects

Most of the text in this section is copied from the ES5 spec. and still needs to be updated to work within the context of the ES6 specification.
An arguments object is an exotic object whose array index properties map to the formal parameters bindings of an invocation of a non-strict function.

Exotic arguments objects have the same internal slots as ordinary objects. They also have a [[ParameterMap]] internal slot.

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 9.1.

9.4.4.1 Arguments Object

When function code is evaluated, an arguments object is created unless (as specified in 9.2.13) the identifier arguments occurs as an Identifier in the function’s FormalParameters 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 with the intrinsic object %ObjectPrototype% as its argument.
3. Call the [[DefineOwnProperty]] internal method on obj passing "length" and the PropertyDescriptor{[[Value]]: len, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true} as arguments.
4. Let indx = len - 1.
5. Repeat while indx ≥ 0,
   a. Let val be the element of args at 0-originated list position indx.
   b. Call the [[DefineOwnProperty]] internal method on obj passing ToString(indx) and the PropertyDescriptor{[[Value]]: val, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true} as arguments.
   c. Let indx = indx - 1
6. Return obj

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

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

The abstract operation CompleteMappedArgumentsObject is called with object obj, object func, grammar production formals, and environment record env. obj must have been previously created by the abstract 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 with the intrinsic object %ObjectPrototype% as its argument.
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(indices) and the PropertyDescriptor([[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 slot of obj to map.
   b. Set the [[Get]], [[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 PropertyDescriptor([[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 FormalParameters : [empty] production.

4. Return the result of calling the abstract operation FunctionCreate with arguments Normal, parameters, body, env, and true.

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 "=".

2. Let parameters be the result of parsing paramText using FormalParameters 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 with arguments Normal, parameters, body for FunctionBody, env, and true.

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 slot 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 [[Get]] internal method (9.1.8) on args passing P and args as the arguments.

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 for ordinary objects (9.1.5) 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 slot 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
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 \( \text{Desc} \) performs the following steps:

1. Let \( \text{map} \) be the value of the [[ParameterMap]] internal slot of the arguments object.
2. Let \( \text{isMapped} \) be the result of calling the [[GetOwnProperty]] internal method of \( \text{map} \) passing \( P \) as the argument.
3. Let \( \text{allowed} \) be the result of calling the default [[DefineOwnProperty]] internal method for ordinary objects (9.1.6) on the arguments object passing \( P \) and \( \text{Desc} \) as the arguments.
4. ReturnIfAbrupt(\( \text{allowed} \)).
5. If \( \text{allowed} \) is \( \text{false} \), then return \( \text{false} \).
6. If the value of \( \text{isMapped} \) is not \( \text{undefined} \), then
   a. If IsAccessorDescriptor(\( \text{Desc} \)) is \( \text{true} \), then
      i. Call the [[Delete]] internal method of \( \text{map} \) passing \( P \) as the argument.
   b. Else
      i. If \( \text{Desc}.[[\text{Value}]] \) is present, then
         1. Let \( \text{putStatus} \) be the result of \( \text{Put}(\text{map}, P, \text{Desc}.[[\text{Value}]], \text{false}) \).
         2. Assert: \( \text{putStatus} \) is \( \text{true} \) because formal parameters mapped by argument objects are always writable.
      ii. If \( \text{Desc}.[[\text{Writable}]] \) is present and its value is \( \text{false} \), then
         1. Call the [[Delete]] internal method of \( \text{map} \) passing \( P \) as the argument.
6. Return \( \text{true} \).

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 \( \text{map} \) be the value of the [[ParameterMap]] internal slot of the arguments object.
2. Let \( \text{isMapped} \) be the result of calling the [[GetOwnProperty]] internal method of \( \text{map} \) passing \( P \) as the argument.
3. Let \( \text{result} \) be the result of calling the default [[Delete]] internal method for ordinary objects (9.1.10) on the arguments object passing \( P \) as the argument.
4. If \( \text{result} \) is \( \text{true} \) and the value of \( \text{isMapped} \) is not \( \text{undefined} \), then
   a. Call the [[Delete]] internal method of \( \text{map} \) passing \( P \) as the argument.
5. Return \( \text{result} \).

NOTE 1 For non-strict mode functions the integer indexed 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 “callee” 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.
9.4.5 Integer Indexed Exotic Objects

An Integer Indexed object is an exotic object that performs special handling of integer property keys. Integer Indexed exotic objects have the same internal slots as ordinary objects additionally [[ViewedArrayBuffer]], [[ArrayLength]], [[ByteOffset]], and [[TypedArrayName]] internal slots.

Integer Indexed exotic objects provide alternative definitions for the following internal methods. All of the other Integer Indexed exotic object essential internal methods that are not defined below are as specified in 9.1.

9.4.5.1 [[GetOwnProperty]] (P)

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

1. Assert: IsPropertyKey(P) is true.
2. Assert: O is an Object that has a [[ViewedArrayBuffer]] internal slot.
3. If Type(P) is String, then
   a. Let index be ToInteger(P).
   b. Assert: index is not an abrupt completion.
   c. If SameValue(ToString(index), P) is true, then
      i. Let value be the result of IntegerIndexedElementGet (O, index).
      ii. ReturnIfAbrupt(value).
      iii. If value is undefined, then return undefined.
      iv. Let writable be true if the integer indexed properties of O are writable and false if they are not.
      v. Return a PropertyDescriptor { [[Value]]: value, [[Enumerable]]: true, [[Writable]]: writable, [[Configurable]]: false }.
4. Return the result of OrdinaryGetOwnProperty(O, P).

9.4.5.2 [[DefineOwnProperty]] (P, Desc)

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

1. Assert: IsPropertyKey(P) is true.
2. Assert: O is an Object that has a [[ViewedArrayBuffer]] internal slot.
3. If Type(P) is String, then
   a. Let index be ToInteger(P).
   b. Assert: index is not an abrupt completion.
   c. If SameValue(ToString(index), P) is true, then
      i. If index < 0, then return false.
      ii. Let length be the value of O’s [[ArrayLength]] internal slot.
      iii. If length is undefined, then throw a TypeError exception.
      iv. If index ≥ length, then return false.
      v. If IsAccessorDescriptor(Desc) is true, then return false.
      vi. If Desc has a [[Configurable]] field and Desc.[[Configurable]] is true, then return false.
      vii. If Desc has an [[Enumerable]] field and Desc.[[Enumerable]] is false, then return false.
      viii. Let writable be true if the integer indexed properties of O are writable and false if they are not.
      ix. Let makeReadOnly be false.
      x. If Desc has a [[Writable]] field, then
         1. If Desc.[[Writable]] is true and writable is false, then return false.
         2. If Desc.[[Writable]] is false and writable is true, then let makeReadOnly be true.
      xi. If Desc has a [[Value]] field, then
          1. Let value be Desc.[[Value]].
          2. If writable is false, then
             a. Let oldValue be the result of IntegerIndexedElementGet (O, index).
             b. ReturnIfAbrupt(oldValue).
c. If oldValue is undefined, then return false.
d. If SameValue(value, oldValue) is false, then return false.
3. Else
   a. Let status be the result of IntegerIndexedElementSet(O, intIndex, value).
   b. ReturnIfAbrupt(status).
   xii. If makeReadOnly is true, then mark the integer indexed properties of O as non-writable.
   xiii. Return true.
4. Return the result of OrdinaryGetOwnProperty(O, P).

9.4.5.3 [[Get]] (P, Receiver)

When the [[Get]] internal method of an Integer Indexed exotic object O is called with property key P and ECMAScript language value Receiver the following steps are taken:

1. Assert: IsPropertyKey(P) is true.
2. If Type(P) is String and if SameValue(O, Receiver) is true, then
   a. Let intIndex be ToInteger(P).
   b. Assert: intIndex is not an abrupt completion.
   c. If SameValue(ToString(intIndex), P) is true, then
      i. Return the result of ToBoolean(IntegerIndexedElementGet(O, intIndex)).
3. Return the result of calling the default ordinary object [[Get]] internal method (9.1.8) on O passing P and Receiver as arguments.

9.4.5.4 [[Set]] (P, V, Receiver)

When the [[Set]] internal method of an Integer Indexed exotic object O is called with property key P, value V, and ECMAScript language value Receiver, the following steps are taken:

1. Assert: IsPropertyKey(P) is true.
2. If Type(P) is String and if SameValue(O, Receiver) is true, then
   Let intIndex be ToInteger(P).
   Assert: intIndex is not an abrupt completion.
   If SameValue(ToString(intIndex), P) is true, then
   Return the result of ToBoolean(IntegerIndexedElementSet(O, intIndex, V)).
3. Return the result of calling the default ordinary object [[Set]] internal method (9.1.8) on O passing P, V, and Receiver as arguments.

9.4.5.5 [[Enumerate]] ()

9.4.5.6 [[OwnPropertyKeys]] ()

9.4.5.7 IntegerIndexedObjectCreate Abstract Operation

The abstract operation IntegerIndexedObjectCreate with argument prototype is used to specify the creation of new Integer Indexed exotic objects. It performs the following steps:

1. Let A be a newly created object.
2. Set A’s essential internal methods to the default ordinary object definitions specified in 9.1.
3. Set the [[GetOwnProperty]] internal method of A as specified in 9.4.5.1.
4. Set the [[DefineOwnProperty]] internal method of A as specified in 9.4.5.2.
5. Set the [[Get]] internal method of A as specified in 9.4.5.3.
6. Set the [[Set]] internal method of A as specified in 9.4.5.4.
7. Set the [[Enumerate]] internal method of A as specified in 9.4.5.5.
8. Set the [[OwnPropertyKeys]] internal method of A as specified in 9.4.5.6.
9. Set the [[Prototype]] internal slot of A to prototype.
10. Set the [[Extensible]] internal slot of A to true.
11. Return A.

9.4.5.8 IntegerIndexedElementGet ( O, index ) Abstract Operation

1. Assert: Type(index) is Number and index is an integer.
2. Assert: O is an Object that has [[ViewedArrayBuffer]], [[ArrayLength]], [[ByteOffset]], and [[TypedArrayName]] internal slots.
3. Let buffer be the value of O’s [[ViewedArrayBuffer]] internal slot.
4. If buffer is undefined, then throw a TypeError exception.
5. Let length be the value of O’s [[ArrayLength]] internal slot.
6. If index < 0 or index ≥ length, then return undefined.
7. Let offset be the value of O’s [[ByteOffset]] internal slot.
8. Let arrayTypeName be the string value O’s [[TypedArrayName]] internal slot.
9. Let elementSize be the Number value of the Element Size value specified in Table 36 for arrayTypeName.
10. Let indexedPosition = (index × elementSize) + offset.
11. Let elementType be the string value of the Element Type value in Table 36 for arrayTypeName.
12. Return the result of GetValueFromBuffer(buffer, indexedPosition, elementType).

9.4.5.9 IntegerIndexedElementSet ( O, index, value ) Abstract Operation

1. Assert: Type(index) is Number and index is an integer.
2. Assert: O is an Object that has [[ViewedArrayBuffer]], [[ArrayLength]], [[ByteOffset]], and [[TypedArrayName]] internal slots.
3. Let buffer be the value of O’s [[ViewedArrayBuffer]] internal slot.
4. If buffer is undefined, then throw a TypeError exception.
5. Let length be the value of O’s [[ArrayLength]] internal slot.
6. Let numValue be ToNumber(value).
7. ReturnIfAbrupt(numValue).
8. If index < 0 or index ≥ length, then return false.
9. Let offset be the value of O’s [[ByteOffset]] internal slot.
10. Let arrayTypeName be the string value O’s [[TypedArrayName]] internal slot.
11. Let elementSize be the Number value of the Element Size value specified in Table 36 for arrayTypeName.
12. Let indexedPosition = (index × elementSize) + offset.
13. Let elementType be the string value of the Element Type value in Table 36 for arrayTypeName.
14. Let status be the result of SetValueInBuffer(buffer, indexedPosition, elementType, numValue).
15. ReturnIfAbrupt(status).
16. Return true.

9.5 Proxy Object Internal Methods and Internal Slots

A proxy object is an exotic object whose essential internal methods are partially implemented using ECMAScript code. Every proxy object has an internal slot 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 slot called [[ProxyTarget]] whose value is either an object or the null value. 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 if the handler object does not have a method corresponding to the internal trap.

The [[ProxyHandler]] and [[ProxyTarget]] internal slots of a proxy object are always initialised when the object is created and typically may not be modified. Some proxy objects are created in a manner that permits them to be subsequently revoked. When a proxy is revoked, its [[ProxyHandler]] and [[ProxyTarget]] internal slots are set to null causing subsequent invocations of internal methods on that proxy object to throw a TypeError exception.
Because proxy permit arbitrary ECMAScript code to be used to in the implementation of internal methods, it is possible to define a proxy object whose handler methods violates the invariants defined in 6.1.7.3. Some of the internal method invariants defined in 6.1.7.3 are essential integrity invariants. These invariants are explicitly enforced by the proxy internal methods specified in this section. An ECMAScript implementation must be robust in the presence of all possible invariant violations.

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

9.5.1 \([[\text{GetPrototypeOf}]()]\)

When the \([[\text{GetPrototypeOf}]()]\) internal method of an exotic Proxy object \( O \) is called the following steps are taken:

1. Let \( \text{handler} \) be the value of the \([[\text{ProxyHandler}]] \) internal slot of \( O \).
2. If \( \text{handler} \) is \( \text{null} \), then throw a \( \text{TypeError} \) exception.
3. Let \( \text{target} \) be the value of the \([[\text{ProxyTarget}]] \) internal slot of \( O \).
4. Let \( \text{trap} \) be the result of GetMethod(\( \text{handler} \), "\[\text{getPrototypeOf}\]").
5. ReturnIfAbrupt(\( \text{trap} \)).
6. If \( \text{trap} \) is \( \text{undefined} \), then
   a. Return the result of calling the \([[\text{GetPrototypeOf}]] \) internal method of \( \text{target} \).
7. Let \( \text{handlerProto} \) be the result of calling the \([[\text{Call}]] \) internal method of \( \text{trap} \) with \( \text{handler} \) as the \( \text{this} \) value and a new List containing \( \text{target} \).
8. ReturnIfAbrupt(\( \text{handlerProto} \)).
9. If Type(\( \text{handlerProto} \)) is neither Object nor \( \text{null} \), then throw a \( \text{TypeError} \) exception.
10. Let \( \text{extensibleTarget} \) be the result of IsExtensible(\( \text{target} \)).
11. ReturnIfAbrupt(\( \text{extensibleTarget} \)).
12. If \( \text{extensibleTarget} \) is \( \text{true} \), then return \( \text{handlerProto} \).
13. Let \( \text{targetProto} \) be the result of calling the \([[\text{GetPrototypeOf}]] \) internal method of \( \text{target} \).
14. ReturnIfAbrupt(\( \text{targetProto} \)).
15. If SameValue(\( \text{handlerProto} \), \( \text{targetProto} \)) is \( \text{false} \), then throw a \( \text{TypeError} \) exception.
16. Return \( \text{handlerProto} \).

NOTE 
\([[\text{GetPrototypeOf}]] \) for proxy objects enforces the following invariant:
- The result of \([[\text{GetPrototypeOf}]] \) must be either an Object or \( \text{null} \).
- If the target object is not extensible, \([[\text{GetPrototypeOf}]] \) applied to the proxy object must return the same value as \([[\text{GetPrototypeOf}]] \) applied to the proxy object's target object.

9.5.2 \([[\text{SetPrototypeOf}]()](V)\)

When the \([[\text{SetPrototypeOf}]()](V)\) internal method of an exotic Proxy object \( O \) is called with argument \( V \) the following steps are taken:

1. Assert: Either Type(\( V \)) is Object or Type(\( V \)) is \( \text{null} \).
2. Let \( \text{handler} \) be the value of the \([[\text{ProxyHandler}]] \) internal slot of \( O \).
3. If \( \text{handler} \) is \( \text{null} \), then throw a \( \text{TypeError} \) exception.
4. Let \( \text{target} \) be the value of the \([[\text{ProxyTarget}]] \) internal slot of \( O \).
5. Let \( \text{trap} \) be the result of GetMethod(\( \text{handler} \), "\[\text{setPrototypeOf}\]").
6. ReturnIfAbrupt(\( \text{trap} \)).
7. If \( \text{trap} \) is \( \text{undefined} \), then
   a. Return the result of calling the \([[\text{SetPrototypeOf}]] \) internal method of \( \text{target} \) with argument \( V \).
8. Let \( \text{trapResult} \) be the result of calling the \([[\text{Call}]] \) internal method of \( \text{trap} \) with \( \text{handler} \) as the \( \text{this} \) value and a new List containing \( \text{target} \) and \( V \).
9. Let booleanTrapResult be ToBoolean(\( \text{trapResult} \)).
10. ReturnIfAbrupt(\( \text{booleanTrapResult} \)).
11. Let \( \text{extensibleTarget} \) be the result of IsExtensible(\( \text{target} \)).
12. ReturnIfAbrupt(extendibleTarget).
13. If extendibleTarget is true, then return trapResult.
14. Let targetProto be the result of calling the [[GetPrototypeOf]] internal method of target.
15. ReturnIfAbrupt(targetProto).
16. If booleanTrapResult is true and SameValue(V, targetProto) is false, then throw a TypeError exception.
17. Return booleanTrapResult.

NOTE [[SetPrototypeOf]] for proxy objects enforces the following invariant:
- If the target object is not extensible, the argument value must be the same as the result of [[GetPrototypeOf]] applied to target object.

9.5.3 [[IsExtensible]] ()
When the [[IsExtensible]] internal method of an exotic Proxy object O is called, the following steps are taken:
1. Let handler be the value of the [[ProxyHandler]] internal slot of O.
2. If handler is null, then throw a TypeError exception.
3. Let target be the value of the [[ProxyTarget]] internal slot of O.
4. Let trap be the result of GetMethod(handler, "isExtensible").
5. ReturnIfAbrupt(trap).
6. If trap is undefined, then
   a. Return the result of calling the [[IsExtensible]] internal method of target.
7. Let trapResult be the result of calling the [[Call]] internal method of trap with handler as the this value and a new List containing target.
8. Let booleanTrapResult be ToBoolean(trapResult).
9. ReturnIfAbrupt(booleanTrapResult).
10. Let targetResult be the result of calling the [[IsExtensible]] internal method of target.
11. ReturnIfAbrupt(targetResult).
12. If SameValue(booleanTrapResult, targetResult) is false, then throw a TypeError exception.
13. Return booleanTrapResult.

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 target object with the same argument.

9.5.4 [[PreventExtensions]] ()
When the [[PreventExtensions]] internal method of an exotic Proxy object O is called, the following steps are taken:
1. Let handler be the value of the [[ProxyHandler]] internal slot of O.
2. If handler is undefined, then throw a TypeError exception.
3. Let target be the value of the [[ProxyTarget]] internal slot of O.
4. Let trap be the result of GetMethod(handler, "preventExtensions").
5. ReturnIfAbrupt(trap).
6. If trap is undefined, then
   a. Return the result of calling the [[PreventExtensions]] internal method of target.
7. Let trapResult be the result of calling the [[Call]] internal method of trap with handler as the this value and a new List containing target.
8. Let booleanTrapResult be ToBoolean(trapResult).
9. ReturnIfAbrupt(booleanTrapResult).
10. Let targetIsExtensible be the result of calling the [[IsExtensible]] internal method of target.
11. ReturnIfAbrupt(targetIsExtensible).
12. If booleanTrapResult is true and targetIsExtensible is true, then throw a TypeError exception.
13. Return booleanTrapResult.

NOTE [[PreventExtensions]] for proxy objects enforces the following invariant:
- [[PreventExtensions]] applied to the proxy object only returns true if [[IsExtensible]] applied to the proxy object’s target object is false.
9.5.5 [[GetOwnProperty]] (P)

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

1. **Assert:** IsPropertyKey($P$) is `true`.
2. Let $handler$ be the value of the `[[ProxyHandler]]` internal slot of $O$.
3. If $handler$ is `null`, then throw a `TypeError` exception.
4. Let $target$ be the value of the `[[ProxyTarget]]` internal slot of $O$.
5. Let $trap$ be the result of GetMethod($handler$, "getOwnPropertyDescriptor").
6. ReturnIfAbrupt($trap$).
7. If $trap$ is `undefined`, then
   a. Return the result of calling the `[[GetOwnProperty]]` internal method of $target$ with argument $P$.
8. Let $trapResultObj$ be the result of calling the `[[Call]]` internal method of $trap$ with $handler$ as the this value and a new List containing $target$ and $P$.
9. ReturnIfAbrupt($trapResultObj$).
10. If Type($trapResultObj$) is neither Object nor Undefined, then throw a `TypeError` exception.
11. Let $targetDesc$ be the result of calling the `[[GetOwnProperty]]` internal method of $target$ with argument $P$.
12. ReturnIfAbrupt($targetDesc$).
13. If $trapResultObj$ 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 IsExtensible($target$).
    d. ReturnIfAbrupt($extensibleTarget$).
    e. If ToBoolean($extensibleTarget$) is `false`, then throw a `TypeError` exception.
   f. Return `undefined`.
14. Let $extensibleTarget$ be the result of IsExtensible($target$).
15. ReturnIfAbrupt($extensibleTarget$).
16. Set $extensibleTarget$ to ToBoolean($extensibleTarget$).
17. Let $resultDesc$ be ToPropertyDescriptor($trapResultObj$).
18. ReturnIfAbrupt($resultDesc$).
19. Call CompletePropertyDescriptor($resultDesc$, $targetDesc$).
20. Let $valid$ be the result of IsCompatiblePropertyDescriptor($extensibleTarget$, $resultDesc$, $targetDesc$).
21. If $valid$ is `false`, then throw a `TypeError` exception.
22. If $resultDesc$.[[Configurable]] is `false`, then
    a. If $targetDesc$.isUndefined or $targetDesc$.[[Configurable]] is `true`, then
       i. Throw a `TypeError` exception.
23. Return $resultDesc$.

NOTE: [[GetOwnProperty]] 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 an own property of the target object and the target object is not extensible.
- A property cannot be reported as existent, if it does not exist as an own property of the target object and the target object is extensible.
- A property cannot be reported as non-configurable, if it does not exist as an 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 [[DefineOwnProperty]] and will not throw an exception.

9.5.6 [[DefineOwnProperty]] (P, Desc)

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

1. **Assert:** IsPropertyKey($P$) is `true`. 

Commented [AWB1228]: Note the result descriptor defaults are set to the values in the targetDesc (if there is one) rather than the normal defaults. This is a change from the wiki spec.

Commented [AWB1229]: The resultDesc carries a reference to the original descriptor returned by the trap. A copy is not made and missing attribute properties are not added to it. This is a change from the wiki spec.
2. Let handler be the value of the [[ProxyHandler]] internal slot of O.
3. If handler is null, then throw a TypeError exception.
4. Let target be the value of the [[ProxyTarget]] internal slot of O.
5. Let trap be the result of GetMethod(handler, "defineProperty").
6. ReturnIfAbrupt(trap).
7. If trap is undefined, then
   a. Return the result of calling the [[DefineOwnProperty]] internal method of target with arguments P and Desc.
8. Let descObj be FromPropertyDescriptor(Desc).
9. NOTE If Desc was originally generated from an object using ToPropertyDescriptor, then descObj will be that original object.
10. Let trapResult be the result of calling the [[Call]] internal method of trap with handler as the this value and a new List containing target, P, and descObj.
11. Let booleanTrapResult be ToBoolean(trapResult).
12. ReturnIfAbrupt(booleanTrapResult).
13. If booleanTrapResult is false, then return false.
14. Let targetDesc be the result of calling the [[GetOwnProperty]] internal method of target with argument P.
15. ReturnIfAbrupt(targetDesc).
16. Let extensibleTarget be the result of IsExtensible(target).
17. ReturnIfAbrupt(extensibleTarget).
18. Set extensibleTarget to ToBoolean(extensibleTarget),
19. If Desc has a [[Configurable]] field and if Desc.[[Configurable]] is false, then
   a. Let settingConfigFalse be true.
20. Else let settingConfigFalse be false.
21. If targetDesc is undefined, then
   a. If extensibleTarget is false, then throw a TypeError exception.
   b. If settingConfigFalse is true, then throw a TypeError exception.
22. Else targetDesc is not undefined, then
   a. If IsCompatiblePropertyDescriptor(extensibleTarget, Desc, targetDesc) is false, then throw a TypeError exception.
   b. If settingConfigFalse is true and targetDesc.[[Configurable]] is true, then throw a TypeError exception.
23. Return true.

NOTE [[DefineOwnProperty]] 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 exist as a non-configurable own property of the target object.
- A property may not be non-configurable, if it 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 [[DefineOwnProperty]] will not throw an exception.

9.5.7 [[HasProperty]](P)

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

1. Assert: IsPropertyKey(P) is true.
2. Let handler be the value of the [[ProxyHandler]] internal slot of O.
3. If handler is null, then throw a TypeError exception.
4. Let target be the value of the [[ProxyTarget]] internal slot of O.
5. Let trap be the result of GetMethod(handler, "has").
6. ReturnIfAbrupt(trap).
7. If trap is undefined, then
   a. Return the result of calling the [[HasProperty]] internal method of target with argument P.
8. Let trapResult be the result of calling the [[Call]] internal method of trap with handler as the this value and a new List containing target and P.
9. Let booleanTrapResult be ToBoolean(trapResult).
10. ReturnIfAbrupt(booleanTrapResult).
11. If booleanTrapResult 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 `IsExtensible(target)`.
      iii. ReturnIfAbrupt(`extensibleTarget`).
      iv. If `ToBoolean(extensibleTarget)` is `false`, then throw a `TypeError` exception.
12. Return `booleanTrapResult`.

NOTE `[[HasProperty]]` 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 an own property of the target object and the target object is not extensible.

9.5.8 `[[Get]] (P, Receiver)`

When the `[[Get]]` internal method of an exotic Proxy object `O` is called with property key `P` and ECMAScript language value `Receiver` the following steps are taken:

1. Assert: `IsPropertyKey(P)` is `true`.
2. Let `handler` be the value of the `[[ProxyHandler]] internal slot of `O`.
3. If `handler` is `null`, then throw a `TypeError` exception.
4. Let `target` be the value of the `[[ProxyTarget]] internal slot of `O`.
5. Let `trap` be the result of `GetMethod(handler, "get")`.
6. ReturnIfAbrupt(`trap`).
7. If `trap` is `undefined`, then
   a. Return the result of calling the `[[Get]] internal method of `target` with arguments `P` and `Receiver`.
8. Let `trapResult` be the result of calling the `[[Call]]` internal method of `trap` with `handler` as the `this` value and a new List containing `target`, `P`, and `Receiver`.
9. ReturnIfAbrupt(`trapResult`).
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(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.
13. Return `trapResult`.

NOTE `[[Get]]` 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.

9.5.9 `[[Set]] (P, V, Receiver)`

When the `[[Set]]` internal method of an exotic Proxy object `O` is called with property key `P`, value `V`, and ECMAScript language value `Receiver`, the following steps are taken:

1. Assert: `IsPropertyKey(P)` is `true`.
2. Let `handler` be the value of the `[[ProxyHandler]] internal slot of `O`.
3. If `handler` is `undefined`, then throw a `TypeError` exception.
4. Let \( \text{target} \) be the value of the \([\text{ProxyTarget}]\) internal slot of \( O \).
5. Let \( \text{trap} \) be the result of GetMethod(\( \text{handler} \), "set").
6. ReturnIfAbrupt(\( \text{trap} \)).
7. If \( \text{trap} \) is \text{undefined}, then
   a. Return the result of calling the \([\text{Set}]\) internal method of \( \text{target} \) with arguments \( P \), \( V \), and \( \text{Receiver} \).
8. Let \( \text{trapResult} \) be the result of calling the \([\text{Call}]\) internal method of \( \text{trap} \) with \( \text{handler} \) as the \text{this} value and a new List containing \( \text{target} \), \( P \), and \( \text{Receiver} \).
9. Let \( \text{booleanTrapResult} \) be ToBoolean(\( \text{trapResult} \)).
10. ReturnIfAbrupt(\( \text{booleanTrapResult} \)).
11. If \( \text{booleanTrapResult} \) is \text{false}, then return \text{false}.
12. Let \( \text{targetDesc} \) be the result of calling the \([\text{GetOwnProperty}]\) internal method of \( \text{target} \) with argument \( P \).
13. ReturnIfAbrupt(\( \text{targetDesc} \)).
14. If \( \text{targetDesc} \) is not \text{undefined}, then
   a. If \( \text{IsDataDescriptor}(\text{targetDesc}) \) and \( \text{targetDesc}.[[\text{Configurable}]] \) is \text{false} and \( \text{targetDesc}.[[\text{Writable}]] \) is \text{false}, then
      i. If SameValue(V, \( \text{targetDesc}.[[\text{Value}]] \)) is \text{false}, then throw a \text{TypeError} exception.
   b. If \( \text{IsAccessorDescriptor}(\text{targetDesc}) \) and \( \text{targetDesc}.[[\text{Configurable}]] \) is \text{false}, then
      i. If \( \text{targetDesc}.[[\text{Set}]] \) is \text{undefined}, then throw a \text{TypeError} exception.
15. Return \text{true}.

NOTE [[Set]] for proxy objects enforces the following invariants:
- Cannot change the value of a property to be different from the value of the corresponding 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 \text{undefined} as its [[Set]] attribute.

9.5.10 [[Delete]] (P)

When the [[Delete]] internal method of an exotic Proxy object \( O \) is called with property name \( P \) the following steps are taken:

1. Assert: IsPropertyKey(\( P \)) is \text{true}.
2. Let \( \text{handler} \) be the value of the \([\text{ProxyHandler}]\) internal slot of \( O \).
3. If \( \text{handler} \) is \text{null}, then throw a \text{TypeError} exception.
4. Let \( \text{target} \) be the value of the \([\text{ProxyTarget}]\) internal slot of \( O \).
5. Let \( \text{trap} \) be the result of GetMethod(\( \text{handler} \), "deleteProperty").
6. ReturnIfAbrupt(\( \text{trap} \)).
7. If \( \text{trap} \) is \text{undefined}, then
   a. Return the result of calling the [[Delete]] internal method of \( \text{target} \) with argument \( P \).
8. Let \( \text{trapResult} \) be the result of calling the [[Call]] internal method of \( \text{trap} \) with \( \text{handler} \) as the \text{this} value and a new List containing \( \text{target} \) and \( P \).
9. Let \( \text{booleanTrapResult} \) be ToBoolean(\( \text{trapResult} \)).
10. ReturnIfAbrupt(\( \text{booleanTrapResult} \)).
11. If \( \text{booleanTrapResult} \) is \text{false}, then return \text{false}.
12. Let \( \text{targetDesc} \) be the result of calling the [[GetOwnProperty]] internal method of \( \text{target} \) with argument \( P \).
13. ReturnIfAbrupt(\( \text{targetDesc} \)).
14. If \( \text{targetDesc} \) is not \text{undefined}, then return \text{true}.
15. If \( \text{targetDesc}.[[\text{Configurable}]] \) is \text{false}, then throw a \text{TypeError} exception.
16. Return \text{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.

9.5.11 [[Enumerate]] ()

When the [[Enumerate]] internal method of an exotic Proxy object \( O \) is called the following steps are taken:

1. Let \( \text{handler} \) be the value of the \([\text{ProxyHandler}]\) internal slot of \( O \).
2. If \( \text{handler} \) is \text{null}, then throw a \text{TypeError} exception.
3. Let \( \text{target} \) be the value of the \([\text{ProxyTarget}]\) internal slot of \( O \).

© Ecma International 2009 – All rights reserved
4. Let trap be the result of GetMethod(handler, "enumerate").
5. ReturnIfAbrupt(trap).
6. If trap is undefined, then
   a. Return the result of calling the [[Enumerate]] internal method of target.
7. Let trapResult be the result of calling the [[Call]] internal method of trap with handler as the this value and a new List containing target.
8. ReturnIfAbrupt(trapResult).
9. If Type(trapResult) is not Object, then throw a TypeError exception.
10. Return trapResult.

**NOTE**

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

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

9.5.12 [[OwnPropertyKeys]]

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

1. Let handler be the value of the [[ProxyHandler]] internal slot of O.
2. If handler is undefined, then throw a TypeError exception.
3. Let target be the value of the [[ProxyTarget]] internal slot of O.
4. Let trap be the result of GetMethod(handler, "ownKeys").
5. ReturnIfAbrupt(trap).
6. If trap is undefined, then
   a. Return the result of calling the [[OwnPropertyKeys]] internal method of target.
7. Let trapResult be the result of calling the [[Call]] internal method of trap with handler as the this value and a new List containing target.
8. ReturnIfAbrupt(trapResult).
9. If Type(trapResult) is not Object, then throw a TypeError exception.
10. 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.
11. Return trapResult.

**NOTE**

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

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

9.5.13 [[Call]]

The [[Call]] internal method of an exotic Proxy object O is called with parameters thisArgument and argumentsList, a List of ECMAScript language values. The following steps are taken:

1. Let handler be the value of the [[ProxyHandler]] internal slot of O.
2. If handler is null, then throw a TypeError exception.
3. Let target be the value of the [[ProxyTarget]] internal slot of O.
4. Let trap be the result of GetMethod(handler, "apply").
5. ReturnIfAbrupt(trap).
6. If trap is undefined, then
   a. Return the result of calling the [[Call]] internal method of target with arguments thisArgument and argumentsList.
7. Let argArray be the result of CreateArrayFromList(argumentsList).
8. Return the result of calling the [[Call]] internal method of trap with handler as the this value and a new List containing target, thisArgument, and argArray.

**NOTE**

Proxy exotic object only has a [[Call]] internal method if the initial value of its [[ProxyTarget]] internal slot is an object that has a [[Call]] internal method.
9.5.14 [[Construct]] Internal Method

The [[Construct]] internal method of an exotic Proxy object O is called with a single parameter `argumentsList` which is a possibly empty List of ECMAScript language values. The following steps are taken:

1. Let `handler` be the value of the [[ProxyHandler]] internal slot of O.
2. If `handler` is `null`, then throw a `TypeError` exception.
3. Let `target` be the value of the [[ProxyTarget]] internal slot of O.
4. Let `trap` be the result of `GetMethod(handler, "construct")`.
5. ReturnIfAbrupt(`trap`).
6. If `trap` is `undefined`, then
   a. If `target` does not have a [[Construct]] internal method, then throw a `TypeError` exception.
   b. Return the result of calling the [[Construct]] internal method of `target` with argument `argumentsList`.
7. Let `argArray` be the result of `CreateArrayFromList(argumentsList)`.  
8. Let `newObj` be the result of calling `trap` with `handler` as the `this` value and a new List containing `target` and `argArray`.
9. ReturnIfAbrupt(`newObj`).
10. If Type(`newObj`) is not Object, then throw a `TypeError` exception.
11. Return `newObj`.

NOTE 1 A Proxy exotic object only has a [[Construct]] internal method if the initial value of its [[ProxyTarget]] internal slot is an object that has a [[Construct]] internal method.

NOTE 2 [[Construct]] for proxy objects enforces the following invariants:
   - The result of [[Construct]] must be an Object.

9.5.15 ProxyCreate(target, handler) Abstract Operation

The abstract operation ProxyCreate with arguments `target` and `handler` is used to specify the creation of new Proxy exotic objects. It performs the following steps:

1. If Type(`target`) is not Object, throw a `TypeError` Exception.
2. If Type(`handler`) is not Object, throw a `TypeError` Exception.
3. Let `P` be a newly created object.
4. Set `P`’s essential internal methods to the definitions specified in 9.5.
5. If IsCallable(`target`) is `true`, then
   a. Set the [[Call]] internal method of `P` as specified in 9.5.13.
   b. If `target` has a [[Construct]] internal method, then
      i. Set the [[Construct]] internal method of `P` as specified in 9.5.14.
6. Set the [[ProxyTarget]] internal slot of `P` to `target`.
7. Set the [[ProxyHandler]] internal slot of `P` to `handler`.
8. Return `P`.

10 ECMAScript Language: Source Code

10.1 Source Text

Syntax

SourceCharacter ::= any Unicode code point

The ECMAScript code is expressed using Unicode, version 5.1 or later. ECMAScript source text is a sequence of code points. All Unicode code point values from U+0000 to U+10FFFF, including surrogate code points, may occur in source text where permitted by the ECMAScript grammars. The actual encodings used to store and interchange ECMAScript source text is not relevant to this specification. 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 a Unicode code point. Conforming ECMAScript implementations are not required to perform any normalisation of text, or behave as though they were performing normalisation of text.
The components of a combining character sequence are treated as individual Unicode code points even though a user might think of the whole sequence as a single character.

NOTE In string literals, regular expression literals, template literals and identifiers, any Unicode code point may also be expressed using Unicode escape sequences that explicitly express a code point’s numeric value. Within a comment, such an escape sequence is effectively ignored as part of the comment.

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 \u000A is line feed) and therefore the next character is not part of the comment. Similarly, if the Unicode escape sequence \u000A occurs within a string literal in a Java program, it is likewise interpreted as a line terminator, which is not allowed within a string literal—one must write \n instead of \u000A to cause a line feed to be part of the string value of a string literal. In an ECMAScript program, a Unicode escape sequence occurring within a comment is never interpreted and therefore cannot contribute to termination of the comment. Similarly, a Unicode escape sequence occurring within a string literal in an ECMAScript program always contributes a Unicode code unit or code point (depending upon the first of the escape) to the literal and is never interpreted as a line terminator or as a quote mark that might terminate the string literal.

10.1.1 Static Semantics: UTF-16 Encoding

The UTF-16 Encoding of a numeric code point value, \( cp \), is determined 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 \( cu2 \).

10.1.2 Static Semantics: UTF16Decode(lead, trail)

Two code units, \( lead \) and \( trail \), that form a UTF-16 surrogate pair are converted to a code point by performing the following steps:

1. Assert: \( 0xD800 \leq lead \leq 0xDBFF \) and \( 0xDC00 \leq trail \leq 0xDFFF \).
2. If \( cp \leq 65535 \), then return \( cp \).
3. Let \( cp \) be ((\( lead – 55296 \))×1024)+\( trail – 56320 \)). NOTE 55296 is 0xD800 and 56320 is 0xDC00.
4. Return the code point \( cp \).

10.2 Types of Source Code

There are four types of ECMAScript 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 slot (see 9.2) 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, GeneratorBody, ConciseBody, or ClassBody.
• Module code is source text that is code that is provided as a ModuleBody. It is the code that is
directly evaluated when a module is initialised. The module code of a particular module does not
include any source text that is parsed as part of a nested FunctionBody, GeneratorBody, ConciseBody,
ClassBody, or ModuleBody.

NOTE  Function code is generally provided as the bodies of Function Definitions (14.1), Arrow Function Definitions
(14.2), Method Definitions (14.3) and Generator Definitions (14.4). Function code is also derived from the last argument to
the Function constructor (19.2.1.1) and the GeneratorFunction constructor (25.2.1.1).

10.2.1  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 four types of ECMAScript code are referred to as module
code, 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 15.3).

• Module code is always strict code.

• A ClassDeclaration or a ClassExpression 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 18.2.1.1) 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.

10.2.2  Non-ECMAScript Functions

An ECMAScript implementation may support the evaluation of exotic 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.

11  ECMAScript Language: Lexical Grammar

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 11.9); in examples such as the
following:

Commented [AWB1732]: See
07/july-25.md#consensus-resolution

Commented [AWB933]: May need to also say something
about TemplateSubstitution tail. Also need to consider with
there are any ASI issues concerning it.
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:

```javascript
a = b / hi / g.exec(c).map(d);
```
Table 26 — Format-Control Character Usage

<table>
<thead>
<tr>
<th>Code Point</th>
<th>Name</th>
<th>Formal Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+200C</td>
<td>Zero width non-joiner</td>
<td>&lt;ZWNJ&gt;</td>
<td>IdentifierPart</td>
</tr>
<tr>
<td>U+200D</td>
<td>Zero width joiner</td>
<td>&lt;ZWJ&gt;</td>
<td>IdentifierPart</td>
</tr>
<tr>
<td>U+FEFF</td>
<td>Byte Order Mark</td>
<td>&lt;BOM&gt;</td>
<td>Whitespace</td>
</tr>
</tbody>
</table>

11.2 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 27.

Table 27 — Whitespace Characters

<table>
<thead>
<tr>
<th>Code Point</th>
<th>Name</th>
<th>Formal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+0009</td>
<td>Tab</td>
<td>&lt;TAB&gt;</td>
</tr>
<tr>
<td>U+000B</td>
<td>Vertical Tab</td>
<td>&lt;VT&gt;</td>
</tr>
<tr>
<td>U+000C</td>
<td>Form Feed</td>
<td>&lt;FF&gt;</td>
</tr>
<tr>
<td>U+0020</td>
<td>Space</td>
<td>&lt;SP&gt;</td>
</tr>
<tr>
<td>U+00A0</td>
<td>No-break space</td>
<td>&lt;NBSP&gt;</td>
</tr>
<tr>
<td>U+FEFF</td>
<td>Byte Order Mark</td>
<td>&lt;BOM&gt;</td>
</tr>
<tr>
<td>Other category “Zs”</td>
<td>Any other Unicode “space separator”</td>
<td>&lt;USP&gt;</td>
</tr>
</tbody>
</table>

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>  

11.3 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 (11.9). 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 (11.4) 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 28.

<table>
<thead>
<tr>
<th>Code Point</th>
<th>Name</th>
<th>Formal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+000A</td>
<td>Line Feed</td>
<td>&lt;LF&gt;</td>
</tr>
<tr>
<td>U+000D</td>
<td>Carriage Return</td>
<td>&lt;CR&gt;</td>
</tr>
<tr>
<td>U+2028</td>
<td>Line separator</td>
<td>&lt;LS&gt;</td>
</tr>
<tr>
<td>U+2029</td>
<td>Paragraph separator</td>
<td>&lt;PS&gt;</td>
</tr>
</tbody>
</table>

Only the Unicode code points in Table 28 are treated as line terminators. Other new line or line breaking Unicode code points 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 \s <LF>] ]
- <LS>
- <PS>
- <CR> <LF>

### 11.4 Comments

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

Because a single-line comment can contain any Unicode code point 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 11.9).

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 ::

/\* MultilineCommentCharacters */
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

SingleLineCommentChars ::
  SingleLineCommentChar SingleLineCommentChars opt

SingleLineCommentChar ::
  SourceCharacter but not LineTerminator

11.5 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.

11.6 Names and Keywords

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 an enumerated subset of IdentifierName and Identifier is an IdentifierName that is not a ReservedWord (see 11.6.2). The Unicode identifier grammar is based on character properties specified by the Unicode Standard. The Unicode code points 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 recognize 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 code point to the IdentifierName. The code point is expressed by the HexDigits of the UnicodeEscapeSequence (see 11.8.4). The \ preceding the UnicodeEscapeSequence and the w and { } characters, if they appear, do not contribute code points to the IdentifierName. A UnicodeEscapeSequence cannot be used to put a code point into an IdentifierName that would otherwise be illegal. In other words, if a \ UnicodeEscapeSequence sequence were replaced by the SourceCharacter it contributes, the result must still be a valid IdentifierName that has the exact same sequence of SourceCharacter elements as the original IdentifierName. All interpretations of IdentifierName
within this specification are based upon their actual code points regardless of whether or not an escape sequence was used to contribute any particular characters.

Two \textit{IdentifierName} that are canonically equivalent according to the Unicode standard are \textit{not} equal unless they are represented by the exact same sequence of code points (in other words, conforming ECMAScript implementations are only required to do bitwise comparison on IdentifierName values).

\textbf{Syntax}

\begin{verbatim}
Identifier :: IdentifierName \textbf{but not} ReservedWord

IdentifierName :: IdentifierStart IdentifierName IdentifierPart

IdentifierStart :: UnicodeIDStart
                   $ UnicodeEscapeSequence

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

UnicodeIDStart :: any Unicode code point with the Unicode property "ID_Start".

UnicodeIDContinue :: any Unicode code point with the Unicode property "ID_Continue"
\end{verbatim}

The definitions of the nonterminal \textit{UnicodeEscapeSequence} is given in 11.8.4.

\section*{11.6.1 Identifiers and Identifier Names}

\subsection*{11.6.1.1 Static Semantics: StringValue}

See also: 12.1.2.2, 13.2.1.4.

Identifier :: IdentifierName \textbf{but not} ReservedWord

1. Return the StringValue of IdentifierName.

IdentifierName ::

1. Return the String value consisting of the sequence of code units corresponding to IdentifierName. In determining the sequence any occurrences of \textit{UnicodeEscapeSequence} are first replaced with the code point represented by the \textit{UnicodeEscapeSequence} and then the code points of the entire IdentifierName are converted to code units by UTF-16 Encoding (clause 10.1.1) each code point.
11.6.2 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 specific SourceCharacter elements. However, any code point in a ReservedWord can also be expressed by a UnicodeEscapeSequence that expresses that same Unicode code point. Use of such escape sequences does not change the meaning of the ReservedWord.

11.6.2.1 Keywords

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

Syntax
Keyword :: one of
  break  do   in   typeof
  case   else  instanceof  var
  catch  export  new  void
  class  extends  return  while
  const  finally  super  with
  continue  for  switch  yield
  debugger  function  this
  default  if  throw
  delete  import  try

NOTE In some contexts yield is given the semantics of an Identifier. See 12.1.2, 12.13, 13.2.1

11.6.2.2 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 ::
  enum

NOTE Use of the following tokens within strict mode code (see 10.2.1) is also reserved. That usage is restricted using static semantic restrictions (see 0 and 13.2.1.1) rather than the lexical grammar:

  implements    let    private    public
  interface     package     protected    static
11.7 Punctuators

Syntax

Punctuator : one of

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

DivPunctuator : one of
   / /=

RightBracePunctuator :}

11.8 Literals

11.8.1 Null Literals

Syntax

NullLiteral : null

11.8.2 Boolean Literals

Syntax

BooleanLiteral : true false

11.8.3 Numeric Literals

Syntax

NumericLiteral : DecimallLiteral BinaryIntegerLiteral OctalIntegerLiteral HexIntegerLiteral

DecimallLiteral :

DecimalIntegerLiteral , DecimalDigits , ExponentPart

DecimalIntegerLiteral . DecimalDigits

DecimalIntegerLiteral ExponentPart

DecimalIntegerLiteral ::

0 NonZeroDigit DecimalDigits

Commented [AWB735]: From March 29 meeting notes:
Hex floating point literals:
Waldemar: Other languages include these things. They're rarely used
but when you want one, you really want one. Use cases are similar to
that of hex literals.
Will explore adding them.
MarkM: 0x3.p1 currently evaluates to undefined. This would be a
breaking change.
Waldemar: Not clear anyone would notice. How did other
languages deal with this?
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 BinaryDigits
  0B BinaryDigits

BinaryDigits ::
  BinaryDigit
  BinaryDigits BinaryDigit

BinaryDigit :: one of
  0 1

OctalIntegerLiteral ::
  0o OctalDigits
  0O OctalDigits

OctalDigits ::
  OctalDigit
  OctalDigits 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.

Commented [AWB736]: The various Digit productions could be refactored to have less redundancy
A conforming implementation, when processing strict mode code (see 10.2.1), must not extend the syntax of
NumericLiteral to include LegacyOctalIntegerLiteral as described in 8.1.1.

11.8.3.1 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 . ExponentPart is the MV of DecimalIntegerLiteral \(\times 10^e\),
  where \(e\) is the MV of ExponentPart.
- 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 :: 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 \(\times 10^n\), where \(n\) is the
  number of characters in DecimalDigits and \(e\) is the MV of ExponentPart.
- 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 :: 4 DecimalDigits is the MV of DecimalDigits.
- The MV of SignedInteger :: – DecimalDigit 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 `HexString :: a` or `HexString :: A` is 10.
The MV of `HexString :: b` or `HexString :: B` is 11.
The MV of `HexString :: c` or `HexString :: C` is 12.
The MV of `HexString :: d` or `HexString :: D` is 13.
The MV of `HexString :: e` or `HexString :: E` is 14.
The MV of `HexString :: f` or `HexString :: F` is 15.
The MV of `BinaryIntegerLiteral :: 0b BinaryDigits` is the MV of `BinaryDigits`.
The MV of `BinaryIntegerLiteral :: 0B BinaryDigits` is the MV of `BinaryDigits`.
The MV of `BinaryIntegerLiteral :: 0b BinaryDigit` is the MV of `BinaryDigit`.
The MV of `OctalIntegerLiteral :: 0x OctalDigits` is the MV of `OctalDigits`.
The MV of `OctalIntegerLiteral :: 0X OctalDigits` is the MV of `OctalDigits`.
The MV of `OctalIntegerLiteral :: 0B OctalDigits` is the MV of `OctalDigits`.
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 `HexIntegerLiteral :: 0B HexDigits` is the MV of `HexDigits`.
The MV of `HexIntegerLiteral :: 0b HexDigits` is the MV of `HexDigits`.

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 6.1.6), 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;
- there is a nonzero digit to its left and there is a nonzero digit, not in the `ExponentPart`, to its right.

### 11.8.4 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 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 ECMAScript String values. When generating these string values Unicode code points are UTF-16 encoded as defined in 10.1.1. 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 ::
  " DoubleStringCharactersopt "
  ' SingleStringCharactersopt '

DoubleStringCharacters ::
  DoubleStringCharacter DoubleStringCharactersopt

SingleStringCharacters ::
  SingleStringCharacter SingleStringCharactersopt

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

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

LineContinuation ::
  \ LineTerminatorSequence

EscapeSequence ::
  CharacterEscapeSequence
  0 [lookahead a DecimalDigit]
  HexEscapeSequence
  UnicodeEscapeSequence

A conforming implementation, when processing strict mode code (see 10.2.1), must not extend the syntax of
EscapeSequence to include LegacyOctalEscapeSequence as described in 3.1.2.

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

HexEscapeSequence ::
  x HexDigit HexDigit

UnicodeEscapeSequence ::
  u HexDigits
  u{ HexDigits }

HexDigits ::
  HexDigit HexDigit HexDigit HexDigit

The definition of the nonterminal HexDigit is given in 11.8.3. SourceCharacter is defined in 10.1.

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.

11.8.4.1 Static Semantics: Early Errors

UnicodeEscapeSequence :: u{ HexDigits }
  It is a Syntax Error if the MV of HexDigits > 1114111.
11.8.4.2 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 code points within the string literal are interpreted as having a mathematical value (MV), as described below or in 11.6.3.

- 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 CV of DoubleStringCharacter :: SourceCharacter but not one of " or \ or LineTerminator is the UTF-16 Encoding (10.1.1) 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 (10.1.1) 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 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 29.

### Table 29 — String Single Character Escape Sequences

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Code Unit Value</th>
<th>Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>0x0008</td>
<td>backspace</td>
<td>&lt;BS&gt;</td>
</tr>
<tr>
<td>\t</td>
<td>0x0009</td>
<td>horizontal tab</td>
<td>&lt;HT&gt;</td>
</tr>
<tr>
<td>\n</td>
<td>0x000A</td>
<td>line feed (new line)</td>
<td>&lt;LF&gt;</td>
</tr>
<tr>
<td>\v</td>
<td>0x000B</td>
<td>vertical tab</td>
<td>&lt;VT&gt;</td>
</tr>
<tr>
<td>\f</td>
<td>0x000C</td>
<td>form feed</td>
<td>&lt;FF&gt;</td>
</tr>
<tr>
<td>\r</td>
<td>0x000D</td>
<td>carriage return</td>
<td>&lt;CR&gt;</td>
</tr>
<tr>
<td>&quot;</td>
<td>0x0022</td>
<td>double quote</td>
<td>&quot;</td>
</tr>
<tr>
<td>'</td>
<td>0x0027</td>
<td>single quote</td>
<td>'</td>
</tr>
<tr>
<td>\</td>
<td>0x003C</td>
<td>backslash</td>
<td>\</td>
</tr>
</tbody>
</table>

- The CV of CharacterEscapeSequence :: NonEscapeCharacter is the CV of the NonEscapeCharacter.
- The CV of NonEscapeCharacter :: SourceCharacter but not one of EscapeCharacter or LineTerminator is the UTF-16 Encoding (10.1.1) of the code point value of SourceCharacter.
- The CV of \texttt{HexEscapeSequence} :: \texttt{\textbackslash x HexDigit HexDigit} is the code unit value that is \((16 \times \text{MV of the first HexDigit}) + \text{MV of the second HexDigit})\).
- The CV of \texttt{UnicodeEscapeSequence} :: \texttt{\textbackslash u Hex4Digit} is the code unit value that is \((4096 \times \text{MV of the first HexDigit}) + (256 \times \text{MV of the second HexDigit}) + (16 \times \text{MV of the third HexDigit}) + \text{MV of the fourth HexDigit})\).
- The CV of \texttt{HexDigit HexDigit HexDigit HexDigit} is the code unit value that is \((4096 \times \text{MV of the first HexDigit}) + (256 \times \text{MV of the second HexDigit}) + (16 \times \text{MV of the third HexDigit}) + \text{MV of the fourth HexDigit})\).
- The CV of \texttt{UnicodeEscapeSequence} :: \texttt{\textbackslash{ HexDigits } } is the UTF-16 Encoding (10.1.1) of the MV of HexDigits.

11.8.5 Regular Expression Literals

NOTE: A regular expression literal is an input element that is converted to a RegExp object (see 21.1.5) each time the literal is evaluated. Two regular expression literals in a program evaluate to regular expression objects that never compare as \texttt{===} to each other even if the two literals' contents are identical. A RegExp object may also be created at runtime by \texttt{new RegExp} (see 21.2.3.2) or calling the \texttt{RegExp} constructor as a function (21.2.3.1).

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 \texttt{RegularExpressionBody} and the \texttt{RegularExpressionFlags} are subsequently parsed using the more stringent ECMAScript Regular Expression grammar (21.2.1).

An implementation may extend the ECMAScript Regular Expression grammar defined in 21.2.1, but it must not extend the \texttt{RegularExpressionBody} and \texttt{RegularExpressionFlags} productions defined below or the productions used by these productions.

Syntax

\[
\text{RegularExpressionLiteral} ::= \text{RegularExpressionBody} / \text{RegularExpressionFlags}
\]

\[
\text{RegularExpressionBody} ::=
\text{RegularExpressionFirstChar} \text{RegularExpressionChars}
\]

\[
\text{RegularExpressionChars} ::=
\text{[empty]} \text{RegularExpressionChars} \text{RegularExpressionChar}
\]

\[
\text{RegularExpressionFirstChar} ::=
\text{RegularExpressionNonTerminator} \text{but not one of} \star \text{or} \text{\slash} \text{or} / \text{or} \{
\]

\[
\text{RegularExpressionChar} ::=
\text{RegularExpressionNonTerminator} \text{but not one of} / \text{or} \}
\]

\[
\text{RegularExpressionBackslashSequence} ::=
\text{RegularExpressionNonTerminator}
\]

\[
\text{RegularExpressionNonTerminator} ::=
\text{SourceCharacter} \text{but not} \text{LineTerminator}
\]

\[
\text{RegularExpressionClass} ::=
[ \text{RegularExpressionClassChars} ]
\]

\[
\text{RegularExpressionClassChars} ::=
\text{UnaryEscapeCharacter} \text{RegularExpressionClassChars}
\]

\[
\text{UnaryEscapeCharacter} ::=
\text{\textbackslash extraclasschar}
\]

\[
\text{\textbackslash extraclasschar} ::=
\text{\textbackslash} \text{\textbackslash extraclasschar}
\]

\[
\text{\textbackslash extraclasschar } ::=\text{\textbackslash} \text{\textbackslash extraclasschar}
\]
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: /(?:)/.

11.8.5.1 Static Semantics: Early Errors

RegularExpressionFlags :: RegularExpressionFlags IdentifierPart

- It is a Syntax Error if IdentifierPart contains a Unicode escape sequence.

11.8.5.2 Static Semantics: BodyText

RegularExpressionLiteral :: / RegularExpressionBody / RegularExpressionFlags
1. Return the source code that was recognised as RegularExpressionBody.

11.8.5.3 Static Semantics: FlagText

RegularExpressionLiteral :: / RegularExpressionBody / RegularExpressionFlags
1. Return the source code that was recognised as RegularExpressionFlags.

11.8.6 Template Literal Lexical Components

Syntax

Template ::
  NoSubstitutionTemplate
  TemplateHead

NoSubstitutionTemplate ::
  ` TemplateCharactersopt

TemplateHead ::
  ` TemplateCharactersopt \{

TemplateSubstitutionTail ::
  TemplateMiddle
  TemplateTail

TemplateMiddle ::
  } TemplateCharactersopt \{

TemplateTail ::
  } TemplateCharactersopt`

TemplateCharacters ::
  TemplateCharacter TemplateCharactersopt
11.8.6.1 Static Semantics: TV's and TRV's

A template literal component is interpreted as a sequence of Unicode code points. The Template Value (TV) of a literal component is described in terms of code unit values (CV, 11.8.4) contributed by the various parts of the template literal component. As part of this process, some Unicode code points within the template component are interpreted as having a mathematical value (MV, 11.8.3). In determining a TV, escape sequences are replaced by the UTF-16 code unit(s) of the Unicode code point 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 NoSubstitutionTemplate :: 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 `$` or LineTerminatorSequence is the UTF-16 Encoding (10.1.1) of the code point value of SourceCharacter.
- The TV of TemplateCharacter :: `$` 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 TemplateCharacter :: LineTerminatorSequence is the TRV of LineTerminatorSequence.
- 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 `$` is the TRV of TemplateCharacters.
- The TRV of TemplateCharacter :: TemplateCharacter TemplateCharacters ` ` is a sequence consisting of the code units in the TRV of TemplateCharacter followed by all the code units in the TRV of TemplateCharacters in order.
- The TRV of TemplateCharacter :: SourceCharacter but not one of ` ` or `\` or `$` or LineTerminatorSequence is the UTF-16 Encoding (10.1.1) of the code point value of SourceCharacter.
- The TRV of TemplateCharacter :: `$` is the code unit value 0x0024.
- The TRV of TemplateCharacter :: `\` EscapeSequence is the sequence consisting of the code unit value 0x000C followed by the code units of TRV of EscapeSequence.
- The TRV of TemplateCharacter :: LineContinuation is the TRV of LineContinuation.
- The TRV of TemplateCharacter :: LineTerminatorSequence is the TRV of LineTerminatorSequence.
• The TRV of EscapeSequence :: CharacterEscapeSequence is the TRV of the CharacterEscapeSequence.
• The TRV of EscapeSequence :: 0 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 :: 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 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 :: x{ HexDigit } 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 :: HexDigit 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 LineTerminatorSequence :: 
LineTerminatorSequence is the sequence consisting of the code unit value 0x000C 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> is the code unit value 0x000D.
• The TRV of LineTerminatorSequence :: <LS> is the code unit value 0x0208.
• The TRV of LineTerminatorSequence :: <PS> is the code unit value 0x0209.
• The TRV of LineTerminatorSequence :: <CR><LF> is the sequence consisting of the code unit value 0x000A.

NOTE TV excludes the code units of LineContinuation while TRV includes them. <CR><LF> and <CR>LineTerminatorSequence are normalized to <LF> for both TV and TRV. An explicit EscapeSequence is needed to include a <CR> or <CR><LF> sequence.

11.9 Automatic Semicolon Insertion

Certain ECMAScript statements (empty statement, let and const declarations, variable statement, expression statement, debugger 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.

11.9.1 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 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 13.6.3).

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

- PostfixExpression
  - LeftHandSideExpression ++
  - LeftHandSideExpression --

- ContinueStatement
  - continue IdentiferReference ;

- BreakStatement
  - break IdentifierReference ;

- ReturnStatement
  - return Expression ;

- ThrowStatement
  - throw Expression ;

- YieldExpression
  - yield AssignmentExpression ;

- ModuleDeclaration
  - module BindingIdentifier FromClause ;

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, throw, or yield token is encountered and a LineTerminator is encountered before the next token, a semicolon is automatically inserted after the continue, break, return, throw, or yield 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 or an AssignmentExpression in a yield expression should start on the same line as the return, throw, or yield token.
- An IdentifierReference in a break or continue statement should be on the same line as the break or continue token.
11.9.2 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.

12 ECMAScript Language: Expressions

12.1 Primary Expressions

Syntax

PrimaryExpression[Yield] :  
this  
IdentifierReference[Yield]  
Literal  
ArrayInitialiser[Yield]  
ObjectLiteral[Yield]  
FunctionExpression  
ClassExpression  
GeneratorExpression  
GeneratorComprehension[Yield]  
RegularExpressionLiteral  
TemplateLiteral[Yield]  
CoverParenthesisedExpressionAndArrowParameterList[Yield]

CoverParenthesisedExpressionAndArrowParameterList[Yield] :  
( Expression[Yield], ... BindingIdentifier[Yield] )  
( )  
( ... BindingIdentifier[Yield] )  
( Expression[Yield], ... BindingIdentifier[Yield] )

Supplemental Syntax

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

ParenthesisedExpression[Yield] :  
( Expression[Yield] )

12.1.0 Semantics

12.1.0.1 Static Semantics: CoveredParenthesisedExpression

CoverParenthesisedExpressionAndArrowParameterList : ( Expression )

1. Return the result of parsing the lexical token stream matched by CoveredParenthesisedExpressionAndArrowParameterList using ParenthesisedExpression as the goal symbol.

12.1.0.2 Static Semantics: IsAnonymousFunctionDefinition

See also: 12.1.10.2, 12.2.1.2, 12.3.2, 12.4.2, 12.5.1, 12.6.1, 12.7.1, 12.8.1, 12.9.2, 12.10.2, 12.11.1, 12.12.1, 12.13.2, 12.14.2, 14.1.6, 14.4.5, 14.5.5.
PrimaryExpression:
  this
  IdentifierReference
  Literal
  ArrayInitialiser
  ObjectLiteral
  GeneratorComprehension
  RegularExpressionLiteral
  TemplateLiteral

1. Return false.

PrimaryExpression: CoverParenthesisedExpressionAndArrowParameterList

1. Let expr be CoverParenthesisedExpression of CoverParenthesisedExpressionAndArrowParameterList.

12.1.0.3 Static Semantics: IsIdentifierRef

See also: 12.2.1.3.

PrimaryExpression: IdentifierReference

1. Return true.

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

1. Return false.

12.1.0.4 Static Semantics: IsValidSimpleAssignmentTarget


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

1. Return false.
PrimaryExpression : IdentifierReference

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

PrimaryExpression : CoverParenthesisedExpressionAndArrowParameterList

1. Let expr be CoveredParenthesisedExpression of CoverParenthesisedExpressionAndArrowParameterList.  
2. Return IsValidSimpleAssignmentTarget of expr.

12.1.1 The this Keyword

12.1.1.1 Runtime Semantics: Evaluation

PrimaryExpression : this

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

12.1.2 Identifier Reference

IdentifierReference[Yield]:
  UnresolvedIdentifier[Yield]

UnresolvedIdentifier[Yield]:
  Identifier
  [Yield] yield

12.1.2.1 Static Semantics: Early Errors

IdentifierReference : UnresolvedIdentifier
  • It is a Syntax Error if StringValue of UnresolvedIdentifier does not statically resolve to a declarative environment record binding.

UnresolvedIdentifier : Identifier
  • It is a Syntax Error if this UnresolvedIdentifier is contained in strict code and if the Identifier is: implements, interface, let, package, private, protected, public, or static.

UnresolvedIdentifier : yield
  • It is a Syntax Error if the IdentifierReference is contained in strict code.

NOTE  Unlike a BindingIdentifier (13.2.1.1), an IdentifierReference in strict code may be the Identifier eval or arguments.

12.1.2.2 Static Semantics: StringValue

See also: 11.6.1.1, 13.2.1.4.

IdentifierReference : yield
1. Return "yield".
12.1.2.3 Runtime Semantics: Evaluation

**IdentifierReference : Identifier**

1. Let `ref` be the result of performing `ResolveBinding(StringValue(Identifier))`.
2. Return `ref`.

**IdentifierReference : yield**

2. Assert: The source code matching this production is not strict code.
3. Let `ref` be the result of performing `ResolveBinding("yield")`.
4. Return `ref`.

**NOTE 1:** The result of evaluating an `IdentifierReference` is always a value of type `Reference`.

**NOTE 2:** In non-strict code, the keyword `yield` may be used as an identifier. Evaluating the `IdentifierReference` production resolved the binding of `yield` as if it was an `Identifier`. The Early Error restriction ensures that such an evaluation only can occur for non-strict code. See 13.2.1 for the handling of `yield` in binding creation contexts.

12.1.3 Literals

**Syntax**

Literal :
   - NullLiteral
   - ValueLiteral

ValueLiteral :
   - BooleanLiteral
   - NumericLiteral
   - StringLiteral

12.1.3.1 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 11.8.3.

**ValueLiteral : StringLiteral**

1. Return the string whose elements are the `SV of StringLiteral` as defined in 11.8.4.

12.1.4 Array Initialiser

**Syntax**

ArrayInitialiser[
   ArrayInitialiser[
       ArrayLiteral[
       ArrayComprehension[
   ]]]]
12.1.4.1 Array Literal

NOTE An Array literal 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 \]
  [ ElementList\[\], Elision \]

ElementList\[\]:
  Elision\[\]
  AssignmentExpression\[\], Elision\[\]
  Elision\[\]
  SpreadElement\[\], Elision\[\]
  ElementList\[\], Elision\[\]
  AssignmentExpression\[\], Elision\[\]
  ElementList\[\], Elision\[\]
  SpreadElement\[\], Elision\[\]

Elision :
  ,

SpreadElement\[\]:
  ... AssignmentExpression\[\], Elision\[\]

12.1.4.1.1 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.

12.1.4.1.2 Runtime Semantics: Array Accumulation

With parameters array and nextIndex.

ElementList : Elision , 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. Let created be the result of calling the [[DefineOwnProperty]] internal method of array with arguments ToString(nextIndex+padding) and the PropertyDescriptor( [[Value]]: initValue, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true).
  6. Assert: created is true.
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. Let `created` be the result of calling the `[[DefineOwnProperty]]` internal method of `array` with arguments `ToString(ToUint32(postIndex+padding))` and the PropertyDescriptor `{{Value}: `initValue`, {Writable}: true, {Enumerable}: true, {Configurable}: true}`.
8. Assert: `created` is `true`.
9. Return `postIndex+padding+1`.

**ElementList**: `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 `spreadObj` be GetValue(`spreadRef`).
3. ReturnIfAbrupt(`spreadObj`).
4. If Type(`spreadObj`) is not `Object`, then throw a `TypeError` exception.
5. Let `iterator` be the result of performing `GetIterator(spreadObj)`.
6. ReturnIfAbrupt(`iterator`).
7. Repeat
   a. Let `next` be the result of `IteratorStep(iterator)`.
   b. ReturnIfAbrupt(`next`).
   c. If `next` is `false`, then return `nextIndex`.
   d. Let `nextValue` be `IteratorValue(next)`.
   e. ReturnIfAbrupt(`nextValue`).
   f. Let `defineStatus` be the result of `CreateDataPropertyOrThrow(A, ToString(ToUint32(nextIndex)), nextValue)`.
   g. ReturnIfAbrupt(`defineStatus`).
   h. Let `nextIndex` = `nextIndex` + 1.

**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 `[[Set]]`.

### 12.1.4.1.3 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).
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.

12.1.4.2 Array Comprehension

Syntax

ArrayComprehension : [ Comprehension ]

Comprehension : ComprehensionFor [ Yield ] ComprehensionTail

ComprehensionTail : AssignmentExpression [ In , ? ]

ComprehensionFor : for ( ForBinding [ Yield ] of AssignmentExpression [ In , ? ] )

ComprehensionIf : if ( AssignmentExpression [ In , ? ] )

ComprehensionFor : for ( ForBinding [ Yield ] of AssignmentExpression [ In , ? ] )

ComprehensionIf : if ( AssignmentExpression [ In , ? ] )

12.1.4.2.1 Static Semantics: Early Errors

ComprehensionFor : for ( ForBinding of AssignmentExpression )

- It is a Syntax Error if the BoundNames of ForBinding contains "let".
- It is a Syntax Error if the BoundNames of ForBinding contains any duplicate entries.

12.1.4.2.2 Runtime Semantics: BindingInitialisation

With arguments value and environment.
NOTE \textit{undefined} is passed for \textit{environment} to indicate that a PutValue operation should be used to assign the initialisation value. This is the case for \texttt{var} statements formal parameter lists of non-strict functions. In those cases a lexical binding is hosted and preinitialised prior to evaluation of its initialiser.

\textbf{ForBinding : BindingPattern}

\begin{enumerate}
\item If Type\texttt{(value)} is not Object, then throw a \texttt{TypeError} exception.
\item Return the result of performing BindingInitialisation for \textit{BindingPattern} passing \texttt{value} and \textit{environment} as the arguments.
\end{enumerate}

\textbf{12.1.4.2.3 Runtime Semantics: ComprehensionEvaluation}

\textbf{With argument} \textit{accumulator}.

NOTE \textit{undefined} is passed for \textit{accumulator} to indicate that a comprehension component is being evaluated as part of a generator comprehension. Otherwise, the value of \textit{accumulator} is the array object into the elements of an array comprehension are to be accumulated.

\textbf{Comprehension : ComprehensionFor ComprehensionTail}

\begin{enumerate}
\item Return the result of performing ComprehensionComponentEvaluation for \textit{ComprehensionFor} with arguments \textit{ComprehensionTail} and \textit{accumulator}.
\end{enumerate}

\textbf{ComprehensionTail : ComprehensionFor ComprehensionTail}

\begin{enumerate}
\item Return the result of performing ComprehensionComponentEvaluation for \textit{ComprehensionFor} with arguments \textit{ComprehensionTail} and \textit{accumulator}.
\end{enumerate}

\textbf{ComprehensionTail : ComprehensionIf ComprehensionTail}

\begin{enumerate}
\item Return the result of performing ComprehensionComponentEvaluation for \textit{ComprehensionIf} with arguments \textit{ComprehensionTail} and \textit{accumulator}.
\end{enumerate}

\textbf{ComprehensionTail : AssignmentExpression}

\begin{enumerate}
\item Let \textit{valueRef} be the result of evaluating \textit{AssignmentExpression}.
\item Let \textit{value} be GetValue\texttt{(valueRef)}.
\item ReturnIfAbrupt\texttt{(value)}.
\item If \textit{accumulator} is not \texttt{undefined}, then
\begin{enumerate}
\item \textbf{Assert:} this is part of an array comprehension.
\item \textbf{Assert:} \textit{accumulator} is an exotic array object so access to its \texttt{length} property should never fail.
\item Let \textit{len} be the result of Get\texttt{(accumulator, \texttt{"length"})}.
\item If \text{\texttt{len}}\texttt{≥2^{32}}-1, then throw a \texttt{RangeError} exception.
\item Let \textit{putStatus} be the result of Put\texttt{(accumulator, To\texttt{String(len)}, value, true)}.
\item ReturnIfAbrupt\texttt{(putStatus)}.
\item Increase \textit{len} by 1.
\item Let \textit{putStatus} be the result of Put\texttt{(accumulator, \texttt{"length"}, \textit{len}, true)}.
\item ReturnIfAbrupt\texttt{(putStatus)}.
\item \textbf{Return NormalCompletion(\texttt{undefined})}.
\end{enumerate}
\item \textbf{Assert:} \textit{accumulator} is \texttt{undefined}, so this is part of a generator comprehension.
\item Let \textit{yieldStatus} be the result of GeneratorYield\texttt{(CreateIterResultObject(value, false))}.
\item ReturnIfAbrupt\texttt{(yieldStatus)}.
\item Return NormalCompletion\texttt{(undefined)}.
\end{enumerate}

\textbf{12.1.4.2.4 Runtime Semantics: ComprehensionComponentEvaluation}

\textbf{With arguments} \textit{tail} and \textit{accumulator}.
NOTE undefined is passed for accumulator to indicate that a comprehension component is being evaluated as part of a generator comprehension. Otherwise, the value of accumulator is the array object into the elements of an array comprehension are to be accumulated.

ComprehensionFor: for (ForBinding of AssignmentExpression )
1. Let exprRef be the result of evaluating AssignmentExpression.
2. Let exprValue be GetValue(exprRef).
3. Let obj be ToObject(exprValue).
4. ReturnIfAbrupt(obj).
5. Let keys be GetIterator(obj).
6. ReturnIfAbrupt(keys).
7. Let oldEnv be the running execution context’s LexicalEnvironment.
8. Repeat
   a. Let nextResult be the result of IteratorStep(keys).
   b. ReturnIfAbrupt(nextResult).
   c. If nextResult is true, then return true.
   d. Let nextValue be IteratorValue(nextResult);
   e. ReturnIfAbrupt(nextValue).
   f. Let forEnv be the result of calling NewDeclarativeEnvironment passing oldEnv as the argument.
   g. For each element name of the BoundNames of ForBinding do
      i. Call forEnv’s CreateMutableBinding concrete method with argument name.
      ii. Assert: The above call to CreateMutableBinding will never return an abrupt completion.
   h. Let status be the result of performing BindingInitialisation for ForBinding passing nextValue and forEnv as the arguments.
   i. ReturnIfAbrupt(status).
   j. Set the running execution context’s LexicalEnvironment to forEnv.
   k. Let continue be the result of performing ComprehensionEvaluation for tail with argument accumulator.
   l. Set the running execution context’s LexicalEnvironment to oldEnv.
   m. ReturnIfAbrupt(continue).
12.1.4.2.5 Runtime Semantics: Evaluation

ArrayComprehension : [ Comprehension ]
1. Let array be the result of the abstract operation ArrayCreate with argument 0.
2. Let status be the result of performing ComprehensionEvaluation for Comprehension with argument array.
3. ReturnIfAbrupt(status).
4. Return array.
12.1.5 Object Initialiser

NOTE 1 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[ yield ] : 
{ } 
{ PropertyDefinitionList[ yield ] } 
{ PropertyDefinitionList[ yield ], } 

PropertyDefinitionList[ yield ] : 
PropertyDefinition[ yield ] 
PropertyDefinitionList[ yield ], PropertyDefinition[ yield ] 

PropertyDefinition[ yield ] : 
IdentifierReference[ yield ] 
CoverInitialisedName[ yield ] 
PropertyName[ yield ] : AssignmentExpression[ , yield ] 
MethodDefinition[ yield ] 

PropertyName[ yield, GeneratorParameter ] : 
LiteralPropertyName 
[ GeneratorParameter ] ComputedPropertyName 
[ GeneratorParameter ] ComputedPropertyName[ yield ]

LiteralPropertyName : 
IdentifierName 
StringLiteral 
NumericLiteral 

ComputedPropertyName[ yield ] : 
[ AssignmentExpression[ , yield ] ]

CoverInitialisedName[ yield ] : 
IdentifierName Initialiser, yield 

Initialiser[ yield ] : 
= AssignmentExpression[ , yield ]

NOTE 2 MethodDefinition is defined in 14.3.

NOTE 3 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.

12.1.5.1 Static Semantics: Early Errors

In addition to describing an actual object initialiser the ObjectLiteral productions are also used as a cover grammar for ObjectAssignmentPattern (12.13.5). 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 : IdentifierReference

• It is a Syntax Error if the PropertyDefinition is contained in strict code and if the IdentifierReference is: implements, interface, let, package, private, protected, public, static, or yield.
• It is a Syntax Error if IdentifierReference is a ReservedWord other than yield.

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 (12.13.5). It cannot occur in an actual object initialiser.

12.1.5.2 Static Semantics: Contains

With parameter symbol.

See also: 5.3, 12.2.1.1, 14.1.3, 14.2.3, 14.4.3, 14.5.4

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.

LiteralPropertyName : 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.

12.1.5.3 Static Semantics: HasComputedPropertyKey

See also: 14.3.3, 14.4.4

PropertyDefinitionList : PropertyDefinitionList , PropertyDefinition

If HasComputedPropertyKey of PropertyDefinitionList is true, then return true.
Return HasComputedPropertyKey of PropertyDefinition.
**PropertyDefinition**: IdentifierReference

1. Return `false`.

**PropertyDefinition**: PropertyName : AssignmentExpression

1. Return `IsComputedPropertyKey` of `PropertyName`.

**NOTE** An alternative semantics for this production is given in B.3.1.

12.1.5.4 **Static Semantics**: `IsComputedPropertyKey`

**PropertyName**: LiteralPropertyName

1. Return `false`.

**PropertyName**: ComputedPropertyName

1. Return `true`.

12.1.5.5 **Static Semantics**: `PropName`

See also: 14.3.5, 14.4.8, 14.5.11

**PropertyDefinition**: IdentifierReference

1. Return `StringValue` of `IdentifierReference`.

**PropertyDefinition**: PropertyName : AssignmentExpression

1. Return `PropName` of `PropertyName`.

**LiteralPropertyName**: StringLiteral

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

**LiteralPropertyName**: NumericLiteral

1. Let `nbr` be the result of forming the value of the `NumericLiteral`.
2. Return `ToString`(`nbr`).

**ComputedPropertyName**: [ AssignmentExpression ]

1. Return `empty`.

12.1.5.6 **Static Semantics**: `PropertyNameList`

**PropertyDefinitionList**: PropertyDefinition

1. If `PropName` of `PropertyDefinition` is `empty`, return a new empty List.
2. Return a new List containing `PropName` of `PropertyDefinition`.

**PropertyDefinitionList**: PropertyDefinitionList , PropertyDefinition

1. Let `list` be `PropertyNameList` of `PropertyDefinitionList`.
2. If `PropName` of `PropertyDefinition` is `empty`, return `list`.
3. Append `PropName` of `PropertyDefinition` to the end of `list`.
4. Return `list`.

Commented [AWB1040]: Issue: static semantic rules probably should call `ToString` (a runtime operation).
12.1.5.7 Runtime Semantics: Evaluation

ObjectLiteral : { }

1. Return the result of the abstract operation ObjectCreate with the intrinsic object %ObjectPrototype% as its argument.

ObjectLiteral : { PropertyDefinitionList } { PropertyDefinitionList , }

1. Let obj be the result of the abstract operation ObjectCreate with the intrinsic object %ObjectPrototype% as its argument.
2. Let status be the result of performing PropertyDefinitionEvaluation of PropertyDefinitionList with argument obj.
3. ReturnIfAbrupt(status).
4. Return obj.

PropertyDefinition : IdentifierReference

1. Return StringValue of IdentifierReference.

PropertyDefinition : PropertyName : AssignmentExpression

1. Return the result of evaluating PropertyName.

LiteralPropertyName : IdentifierName

1. Return StringValue of IdentifierName.

LiteralPropertyName : StringLiteral

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

LiteralPropertyName : NumericLiteral

1. Let nbr be the result of forming the value of the NumericLiteral.
2. Return ToString(nbr).

ComputedPropertyName : [ AssignmentExpression ]

1. Let propName be the result of evaluating AssignmentExpression.
2. Let propName be GetValue(exprValue).
3. Return IfAbrupt(propName).
4. Return ToPropertyKey(propName).

12.1.5.8 Runtime Semantics: PropertyDefinitionEvaluation

With parameter object.

See also: 14.3.9, 14.4.13, B.3.1

PropertyDefinitionList : PropertyDefinitionList , PropertyDefinition
1. Let status be the result of performing PropertyDefinitionEvaluation of PropertyDefinitionList with argument object.
2. ReturnIfAbrupt(status).
3. Let propDef be the result of performing PropertyDefinitionEvaluation of PropertyDefinition with argument object.
4. ReturnIfAbrupt(propDef).
5. If propDef is empty, then return propDef.
6. Let status be the result of DefinePropertyOrThrow(object, propDef.[[key]], propDef.[[descriptor]])
7. Return status.

PropertyDefinition : IdentifierReference
1. Let propName be StringValue of IdentifierReference.
2. Let exprValue be the result of evaluating IdentifierReference.
3. ReturnIfAbrupt(exprValue).
4. Let propValue be GetValue(exprValue).
5. ReturnIfAbrupt(propValue).
6. Return the result of DefinePropertyOrThrow(object, propName, desc).

PropertyDefinition : PropertyName : AssignmentExpression
1. Let propKey be the result of evaluating PropertyName.
2. ReturnIfAbrupt(propKey).
3. Let exprValue be the result of evaluating AssignmentExpression.
4. Let propValue be GetValue(exprValue).
5. ReturnIfAbrupt(propValue).
6. If IsAnonymousFunctionDefinition of AssignmentExpression is true, then
   a. SetFunctionName(propValue, propKey).
7. Let desc be the Property Descriptor([[[Value]]: propValue, [Writable]: true, [[Enumerable]]: true, [Configurable]: true])
8. Return the result of DefinePropertyOrThrow(object, propKey, desc).

NOTE An alternative semantics for this production is given in B.3.1.

12.1.6 Function Defining Expressions
See 14.1 for PrimaryExpression : FunctionExpression.
See 14.4 for PrimaryExpression : GeneratorExpression.
See 14.5 for PrimaryExpression : ClassExpression.

12.1.7 Generator Comprehensions
Syntax
GeneratorComprehension[? ]:
   { Comprehension[? ] }

NOTE The keyword yield may be used in identifier contexts within a GeneratorComprehension contained in non-strict code. The following early error rule ensures that a GeneratorComprehension never contains a YieldExpression.

12.1.7.1 Static Semantics: Early Errors
GeneratorComprehension : { Comprehension }
   • It is a Syntax Error if Comprehension Contains YieldExpression is true.
12.1.7.2 Runtime Semantics: Evaluation

\(\text{GeneratorComprehension} : (\text{Comprehension})\)

1. If \(\text{GeneratorComprehension}\) is contained in strict mode code, then let \(\text{strict}\) be \(\text{true}\); otherwise let \(\text{strict}\) be \(\text{false}\).
2. Let \(\text{scope}\) be the LexicalEnvironment of the running execution context.
3. Let \(\text{parameters}\) be the production: \(\text{FormalParameters : \{}\text{empty}\}\).
4. Let \(\text{closure}\) be the result of performing the \(\text{GeneratorFunctionCreate}\) abstract operation with arguments \(\text{Arrow}\), \(\text{parameters}\), \(\text{body}\), \(\text{scope}\), and \(\text{strict}\).
5. Let \(\text{prototype}\) be the result of the abstract operation \(\text{ObjectCreate}\) with the intrinsic object \%GeneratorPrototype\% as its argument.
6. Perform the abstract operation \(\text{MakeConstructor}\) with arguments \(\text{closure}\), \(\text{true}\), and \(\text{prototype}\).
7. Let \(\text{iterator}\) be the result of calling the \([\text{Call}]\) internal method of \(\text{closure}\) with \(\text{undefined}\) as this argument and an empty List as argumentsList.
8. Return \(\text{iterator}\).

**NOTE** The \(\text{GeneratorFunction}\) object created in step 5 is not observable from ECMAScript code so an implementation may choose to avoid its allocation and initialisation. In that case other semantically equivalent means must be used to allocate and initialise the \(\text{iterator}\) object in step 8. In either case, the \(\text{prototype}\) object created in step 6 must be created because it is potentially observable as the value of the \(\text{iterator}\) object’s \([\text{Prototype}]\) internal slot.

12.1.8 Regular Expression Literals

**Syntax**

See 11.8.4.

12.1.8.1 Static Semantics: Early Errors

\(\text{PrimaryExpression} : \text{RegularExpressionLiteral}\)

- It is a Syntax Error if \(\text{BodyText}\) of \(\text{RegularExpressionLiteral}\) cannot be recognised using the goal symbol \(\text{Pattern}\) of the ECMAScript RegExp grammar specified in 21.2.1.
- It is a Syntax Error if \(\text{FlagText}\) of \(\text{RegularExpressionLiteral}\) contains any character other than \("g\", \("i\", \("m\", \("u\", \("y\), or if it contains the same character more than once.

12.1.8.2 Runtime Semantics: Evaluation

\(\text{PrimaryExpression} : \text{RegularExpressionLiteral}\)

1. Let \(\text{pattern}\) be the string value consisting of the UTF-16 encoding of each code point of \(\text{BodyText}\) of \(\text{RegularExpressionLiteral}\).
2. Let \(\text{flags}\) be the string value consisting of the UTF-16 encoding of each code point of \(\text{FlagText}\) of \(\text{RegularExpressionLiteral}\).
3. Return the result of \(\text{RegExpCreate}(\text{pattern}, \text{flags})\).

12.1.9 Template Literals

**Syntax**

\(\text{TemplateLiteral}\)

- \(\text{NoSubstitutionTemplate}\)
- \(\text{TemplateHead Expression}\)
- \(\text{TemplateSpans}\)

© Ecma International 2013
12.1.9.1 Static Semantics

12.1.9.1.1 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 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 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 TemplateMiddle Expression

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

12.1.9.2 Runtime Semantics

12.1.9.2.1 Runtime Semantics: ArgumentListEvaluation

See also: 12.2.6.1

TemplateLiteral: NoSubstitutionTemplate

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

TemplateLiteral: TemplateHead Expression TemplateSpans

1. Let siteObj be the result of the abstract 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.

12.1.9.2.2 Runtime Semantics: GetTemplateCallSite

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
   a. 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 abstract operation ArrayCreate with argument count.
6. Let rawObj be the result of the abstract 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
      PropertyDescriptor{[[Value]]: cookedValue, [[Enumerable]]: true, [[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
      PropertyDescriptor{[[Value]]: rawValue, [[Enumerable]]: true, [[Writable]]: false, [[Configurable]]: false}.
   f. Let index be index+1.
9. Perform SetIntegrityLevel(rawObj, "frozen").
10. Call the `[[DefineOwnProperty]]` internal method of `siteObj` with arguments "raw" and `PropertyDescriptor{[[Value]]: rawObj, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}`.
11. Perform `SetIntegrityLevel(siteObj, "frozen").` 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.
12. 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 (12.1.9.2.4). 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.

**12.1.9.2.3 Runtime Semantics: SubstitutionEvaluation**

`TemplateSpans : TemplateTail`
1. Return an empty List.

`TemplateSpans : TemplateMiddleList TemplateTail`
1. Return the result of SubstitutionEvaluation of `TemplateMiddleList`.

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

`TemplateMiddleList : TemplateMiddleList TemplateMiddleExpression`
1. Let `preceding` be the result of SubstitutionEvaluation of `TemplateMiddleList`.
2. ReturnIfAbrupt(`preceding`).
3. Let `next` be the result of evaluating `Expression`.
4. ReturnIfAbrupt(`next`).
5. Append `next` as the last element of the List `preceding`.

**12.1.9.2.4 Runtime Semantics: Evaluation**

`TemplateLiteral : NoSubstitutionTemplate`
1. Return the string value whose elements are the `TV` of `NoSubstitutionTemplate` as defined in 11.8.6.

`TemplateLiteral : TemplateHead Expression TemplateSpans`
1. Let `head` be the `TV` of `TemplateHead` as defined in 11.8.6.
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 11.8.6.
2. Return the string whose elements are the code units of `tail`.  

Commented [AWB941]: Note that the conversion semantics are like `String.prototype.concat` rather than the `+` operator.
1. Let `head` be the result of evaluating `TemplateMiddleList`.
2. ReturnIfAbrupt(`head`).
3. Let `tail` be the TV of `TemplateTail` as defined in 11.8.6.
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 11.8.6.
2. Let `middle` be `ToString(sub)`.
3. ReturnIfAbrupt(`middle`).
4. Return the sequence of characters consisting of the code units of `head` followed by the elements of `middle`.

12.1.10 The Grouping Operator

12.1.10.1 Static Semantics: Early Errors

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

12.1.10.2 Static Semantics: IsAnonymousFunctionDefinition

See also: 12.1.0.2, 12.2.1.2, 12.3.2, 12.4.2, 12.5.1, 12.6.1, 12.7.1, 12.8.1, 12.9.2, 12.10.2, 12.11.1, 12.12.1, 12.13.2, 12.14.2, 14.1.6, 14.4.5, 14.5.5.

- ParenthesisedExpression : ( Expression )


12.1.10.3 Static Semantics: IsValidSimpleAssignmentTarget


- ParenthesisedExpression : ( Expression )

1. Return `IsValidSimpleAssignmentTarget` of `Expression`.

Commented [AWB942]: Note that the conversion semantics are like String.prototype.concat rather than the + operator.

Commented [AWB943]: Note that the conversion semantics are like String.prototype.concat rather than the + operator.
12.1.10.4 Runtime Semantics: Evaluation

PrimaryExpression : CoverParenthesisedExpressionAndArrowParameterList

1. Let \( expr \) be the result of evaluating \( \text{CoverParenthesisedExpressionAndArrowParameterList} \).
2. Return the result of evaluating \( expr \).

ParenthesisedExpression : ( 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 \texttt{delete} and \texttt{typeof} may be applied to parenthesised expressions.

12.2 Left-Hand-Side Expressions

Syntax

MemberExpression[\(\text{yield}N\)]:
  \([\text{Lexical goal InputElementRegExp}\text{PrimaryExpression}\text{yield}N]\)
  MemberExpression[\(\text{yield}N\)]: \([\text{Expression}N, \text{yield}N]\)
  MemberExpression[\(\text{yield}N\)]: IdentifierName
  MemberExpression[\(\text{yield}N\)]: TemplateLiteral[\(\text{yield}N\)]
  super [\(\text{Expression}N, \text{yield}N\)]
  super . IdentifierName
  new super [\(\text{ArgumentList}\text{yield}N\)]
  new MemberExpression[\(\text{yield}N\)]: Arguments[\(\text{yield}N\)]
NewExpression[\(\text{yield}N\)]:
  MemberExpression[\(\text{yield}N\)]: \(\text{new}\) NewExpression[\(\text{yield}N\)]
CallExpression[\(\text{yield}N\)]:
  MemberExpression[\(\text{yield}N\)]: Arguments[\(\text{yield}N\)]
  super [\(\text{ArgumentList}\text{yield}N\)]
  super [\(\text{Expression}N, \text{yield}N\)]
  CallExpression[\(\text{yield}N\)]: Arguments[\(\text{yield}N\)]
  CallExpression[\(\text{yield}N\)]: IdentifierName
  CallExpression[\(\text{yield}N\)]: TemplateLiteral[\(\text{yield}N\)]
ArgumentList[\(\text{yield}N\)]:
  ( )
  \{ ArgumentList[\(\text{yield}N\)\}]
ArgumentList[\(\text{yield}N\)]:
  AssignmentExpression[\(\text{yield}N\)]
  . . . AssignmentExpression[\(\text{yield}N\)]
  AssignmentList[\(\text{yield}N\)]: AssignmentExpression[\(\text{yield}N\)]
  AssignmentList[\(\text{yield}N\)]: . . . AssignmentExpression[\(\text{yield}N\)]
LeftHandSideExpression[\(\text{yield}N\)]:
  NewExpression[\(\text{yield}N\)]
  CallExpression[\(\text{yield}N\)]

12.2.1 Static Semantics

12.2.1.1 Static Semantics: Contains

With parameter \(\text{symbol}\).
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.

MemberExpression : new super

1. If symbol is the ReservedWord super, return true.
2. If symbol is the ReservedWord new, return true.
3. Return false.

MemberExpression : new super Arguments

1. If symbol is the ReservedWord super, return true.
2. If symbol is the ReservedWord new, return true.
3. Return the result of Arguments Contains symbol.

12.2.1.2  Static Semantics: IsAnonymousFunctionDefinition

See also: 12.1.0.2, 12.1.10.2, 12.3.2, 12.4.2, 12.5.1, 12.6.1, 12.7.1, 12.8.1, 12.9.2, 12.10.2, 12.11.1, 12.12.1, 12.13.2, 12.14.2, 14.1.6, 14.4.5, 14.5.5.

MemberExpression :
  MemberExpression , Expression
  MemberExpression , IdentifierName
  MemberExpression , TemplateLiteral
  super , Expression
  super , IdentifierName
  new super , Arguments
  new MemberExpression , Arguments

NewExpression :
  new NewExpression
CallExpression:
  MemberExpression Arguments
  super Arguments
  CallExpression Arguments
  CallExpression [ Expression ]
  CallExpression . IdentifierName
  CallExpression TemplateLiteral

1. Return false.

12.2.1.3 Static Semantics: IsIdentifierRef

See also: 12.1.0.3.

LeftHandSideExpression:
  CallExpression

MemberExpression:
  MemberExpression [ Expression ]
  MemberExpression . IdentifierName
  MemberExpression TemplateLiteral
  super [ Expression ]
  super . IdentifierName
  new super Arguments
  new MemberExpression Arguments

NewExpression:
  new NewExpression

1. Return false.

12.2.1.4 Static Semantics: IsValidSimpleAssignmentTarget


CallExpression:
  CallExpression [ Expression ]
  CallExpression . IdentifierName

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

1. Return true.

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

MemberExpression : 
  MemberExpression TemplateLiteral
  **new** super Arguments
  **new** MemberExpression Arguments

1. Return false.

12.2.2 Property Accessors

**NOTE** 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
CallExpression . 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`.

12.2.2.1 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. Let bv be CheckObjectCoercible(baseValue).
8. ReturnIfAbrupt(bv).
9. Let propertyNameString be ToString(propertyNameValue).
10. 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.
11. Return a value of type Reference whose base value is `bv` and whose referenced name is `propertyNameString`, and whose strict reference flag is strict.

CallExpression : CallExpression [ Expression ]
12.2.3 The new Operator

12.2.3.1 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 `IsConstructor(constructor)` is `false`, throw a `TypeError` exception.
5. 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 a List of argument values (12.2.6).
5. ReturnIfAbrupt(`argList`).
6. If `IsConstructor(constructor)` is `false`, throw a `TypeError` exception.
7. Return the result of calling the `[[Construct]]` internal method on `constructor` with `argList` as the argument.
8. If `value` is `true`, then perform the `PrepareForTailCall` abstract operation.
9. Let `result` be the result of calling the `[[Construct]]` internal method on `constructor`, passing `argList` as the argument.
10. Assert: If `tailPosition` is `true`, the above call of `[[Construct]]` will not return here, but instead evaluation will continue with the resumed `LeafCallerContext` as the running execution context.
11. Return `result`.

12.2.4 Function Calls

12.2.4.1 Runtime Semantics: Evaluation

**CallExpression**: `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 performing `ArgumentListEvaluation` of `arguments`.
5. ReturnIfAbrupt(`argList`).
6. If `constructor` is `true`, then perform the `PrepareForTailCall` abstract operation.
7. Let `result` be the result of calling the `[[Construct]]` internal method on `constructor`, passing `argList` as the argument.
8. If `tailPosition` is `true`, then perform the `PrepareForTailCall` abstract operation.
9. Let `result` be the result of calling the `[[Construct]]` internal method on `constructor`, passing `argList` as the argument.
10. Assert: If `tailPosition` is `true`, the above call of `[[Construct]]` will not return here, but instead evaluation will continue with the resumed `LeafCallerContext` as the running execution context.
11. Return `result`.

12.2.4.2 Runtime Semantics: EvaluateCall

The abstract operation `EvaluateCall` takes as arguments a value `ref`, 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 perform the PrepareForTailCall abstract operation.
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. Assert: If result is not an abrupt completion then Type(result) is an ECMAScript language type
13. Return result.

12.2.5 The super Keyword

12.2.5.1 Static Semantics: Early Errors

MemberExpression : super [ Expression ]
   super . IdentifierName
   new super Arguments

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.

12.2.5.2 Runtime Semantics: Evaluation

MemberExpression : super [ Expression ]

1. Let propertyNameReference be the result of evaluating Expression.
2. Let propertyNameValue be GetValue(propertyNameReference).
3. Let propertyKey be ToPropertyKey(propertyNameValue).
4. 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.
5. Return the result of MakeSuperReference(propertyKey, strict).

MemberExpression : super . IdentifierName

1. Let propertyKey be StringValue of IdentifierName.
2. 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.
3. Return the result of MakeSuperReference(propertyKey, strict).

MemberExpression : new super Arguments

1. 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.
2. Let ref be the result of MakeSuperReference(undefined, strict).
3. Let constructor be GetValue(ref).
4. ReturnIfAbrupt(constructor).
5. If Arguments is present, then
   a. Let argList be the result of evaluating Arguments, producing a List of argument values (12.2.6).
   b. ReturnIfAbrupt(argList).
6. Else,
   a. Let argList be a new empty List.
7. If IsConstructor(constructor) is false, throw a TypeError exception.
8. If this MemberExpression is in a tail position (14.6) then let tailPosition be true, otherwise let tailPosition be false.
9. If tailPosition is true, then perform the PrepareForTailCall abstract operation.
10. Let result be the result of calling the [[Construct]] internal method on constructor, passing argList as the argument.
11. Assert: If tailPosition is true, the above call of [[Construct]] will not return here, but instead evaluation will continue with the resumption of leafCallerContext as the running execution context.
12. Return result.

CallExpression : super Arguments

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

12.2.5.3 Runtime Semantics: MakeSuperReference(propertyKey, strict)

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 a ReferenceError exception.
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 bv be CheckObjectCoercible(baseValue).
6. ReturnIfAbrupt(bv).
7. If propertyKey is undefined, then
   a. Let propertyKey be the result of calling the GetMethodName concrete method of env.
8. Return a value of type Reference that is a Super Reference whose base value is bv, whose referenced name is propertyKey, whose thisValue is actualThis, and whose strict reference flag is strict.

12.2.6 Argument Lists

NOTE The evaluation of an argument list produces a List of values (see 6.2.1).

12.2.6.1 Runtime Semantics: ArgumentListEvaluation

See also: 12.1.9.2.1

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 spreadObj be GetValue(spreadRef).
4. ReturnIfAbrupt(spreadObj).
5. If Type(spreadObj) is not Object, then throw a TypeError exception.
6. Let iterator be the result of performing GetIterator(spreadObj).
7. ReturnIfAbrupt(iterator).
8. Repeat
   a. Let next be the result of IteratorStep(iterator).
   b. ReturnIfAbrupt(next).
   c. If next is false, then return list.
   d. Let nextArg be IteratorValue(next).
   e. ReturnIfAbrupt(nextArg).
   f. Append nextArg as the last element of 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 spreadObj be GetValue(spreadRef).
4. ReturnIfAbrupt(spreadObj).
5. If Type(spreadObj) is not Object, then throw a TypeError exception.
6. Let iterator be the result of performing GetIterator(spreadObj).
7. ReturnIfAbrupt(iterator).
8. Repeat
   a. Let next be the result of IteratorStep(iterator).
   b. ReturnIfAbrupt(next).
   c. If next is false, then return precedingArgs.
   d. Let nextArg be IteratorValue(next).
   e. ReturnIfAbrupt(nextArg).
   f. Append nextArg as the last element of precedingArgs.

12.2.7 Tagged Templates
12.2.7.1 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.

12.3 Postfix Expressions

Syntax

PostfixExpression::

  LeftHandSideExpression ++
  LeftHandSideExpression --

12.3.1 Static Semantics: Early Errors

PostfixExpression:

  LeftHandSideExpression ++
  LeftHandSideExpression --

• It is an early Reference Error if IsValidSimpleAssignmentTarget of LeftHandSideExpression is false.

12.3.2 Static Semantics: IsAnonymousFunctionDefinition

See also: 12.1.0.2, 12.1.10.2, 12.2.1.2, 12.4.2, 12.5.1, 12.6.1, 12.7.1, 12.8.1, 12.9.2, 12.10.2, 12.11.1, 12.12.1, 12.13.2, 12.14.1

PostfixExpression:

  LeftHandSideExpression ++
  LeftHandSideExpression --

1. Return false.

12.3.3 Static Semantics: IsValidSimpleAssignmentTarget


PostfixExpression:

  LeftHandSideExpression ++
  LeftHandSideExpression --

1. Return false.

12.3.4 Postfix Increment Operator

12.3.4.1 Runtime Semantics: Evaluation

PostfixExpression : LeftHandSideExpression ++

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 12.6.5).
5. Let status be PutValue(lhs, newValue).
6. ReturnIfAbrupt(status).
7. Return oldValue.
12.3.5 Postfix Decrement Operator

12.3.5.1 Runtime Semantics: Evaluation

PostfixExpression : LeftHandSideExpression --

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 (12.6.5).
4. Let \( status \) be PutValue(\( lhs \), \( newValue \)).
5. ReturnIfAbrupt(\( status \)).
6. Return \( oldValue \).

12.4 Unary Operators

Syntax

UnaryExpression:\n  PostfixExpression;\n  delete UnaryExpression;\n  void UnaryExpression;\n  typeof UnaryExpression;\n  ++ UnaryExpression;\n  -- UnaryExpression;\n  + UnaryExpression;\n  - UnaryExpression;\n  ~ UnaryExpression;\n  ! UnaryExpression;

12.4.1 Static Semantics: Early Errors

UnaryExpression:\n  ++ UnaryExpression\n  -- UnaryExpression

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

12.4.2 Static Semantics: IsAnonymousFunctionDefinition

See also: 12.1.0.2, 12.1.10.2, 12.2.1.2, 12.3.2, 12.5.1, 12.6.1, 12.7.1, 12.8.1, 12.9.2, 12.10.2, 12.11.1, 12.12.1, 12.13.2, 12.14.2, 14.1.6, 14.4.5, 14.5.5.

UnaryExpression:\n  delete UnaryExpression\n  void UnaryExpression\n  typeof UnaryExpression\n  ++ UnaryExpression\n  -- UnaryExpression\n  + UnaryExpression\n  - UnaryExpression\n  ~ UnaryExpression\n  ! UnaryExpression

1. Return false.
12.4.3 Static Semantics: IsValidSimpleAssignmentTarget


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

1. Return false.

12.4.4 The delete Operator

12.4.4.1 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 PrimaryExpression: IdentifierReference.
- It is a Syntax Error if the derived UnaryExpression is PrimaryExpression: CoverParenthesisedExpressionAndArrowParameterList 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.

NOTE The last rule means that expressions such as delete (((foo))) produce early errors because of recursive application of the first rule.

12.4.4.2 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 deleteStatus.
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.

12.4.5 The `void` Operator

12.4.5.1 Runtime Semantics: Evaluation

UnaryExpression : `void` UnaryExpression

1. Let `expr` be the result of evaluating UnaryExpression.
2. Let `status` be 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.

12.4.6 The `typeof` Operator

12.4.6.1 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 according to Table 30.

<table>
<thead>
<tr>
<th>Type of val</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>&quot;undefined&quot;</td>
</tr>
<tr>
<td>Null</td>
<td>&quot;object&quot;</td>
</tr>
<tr>
<td>Boolean</td>
<td>&quot;boolean&quot;</td>
</tr>
<tr>
<td>Number</td>
<td>&quot;number&quot;</td>
</tr>
<tr>
<td>String</td>
<td>&quot;string&quot;</td>
</tr>
<tr>
<td>Symbol</td>
<td>&quot;symbol&quot;</td>
</tr>
<tr>
<td>Object (ordinary and does not implement [[Call]])</td>
<td>&quot;object&quot;</td>
</tr>
<tr>
<td>Object (standard exotic and does not implement [[Call]])</td>
<td>&quot;object&quot;</td>
</tr>
<tr>
<td>Object (non-standard exotic and does not implement [[Call]])</td>
<td>Implementation-defined. May not be &quot;undefined&quot;, &quot;boolean&quot;, &quot;number&quot;, &quot;symbol&quot;, or &quot;string&quot;.</td>
</tr>
</tbody>
</table>

NOTE Implementations are discouraged from defining new `typeof` result values for non-standard exotic objects. If possible "object" should be used for such objects.
12.4.7 Prefix Increment Operator

12.4.7.1 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 12.6.5).
5. Let status be PutValue(expr, newValue).
6. ReturnIfAbrupt(status).
7. Return newValue.

12.4.8 Prefix Decrement Operator

12.4.8.1 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 12.6.5).
5. Let status be PutValue(expr, newValue).
6. ReturnIfAbrupt(status).
7. Return newValue.

12.4.9 Unary + Operator

NOTE The unary + operator converts its operand to Number type.

12.4.9.1 Runtime Semantics: Evaluation

UnaryExpression : + UnaryExpression

1. Let expr be the result of evaluating UnaryExpression.
2. Return ToNumber(GetValue(expr)).

12.4.10 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.

12.4.10.1 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.
12.4.11 Bitwise NOT Operator ( ~ )

12.4.11.1 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.

12.4.11.2 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.

12.5 Multiplicative Operators

Syntax

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

12.5.1 Static Semantics: IsAnonymousFunctionDefinition

See also: 12.1.0.2, 12.1.10.2, 12.2.1.2, 12.3.2, 12.4.2, 12.6.1, 12.7.1, 12.8.1, 12.9.2, 12.10.2, 12.11.1, 12.12.1, 12.13.2, 12.14.2, 14.4.5, 14.5.5.

MultiplicativeExpression :
    MultiplicativeExpression * UnaryExpression
    MultiplicativeExpression / UnaryExpression
    MultiplicativeExpression % UnaryExpression

1. Return false.

12.5.2 Static Semantics: IsValidSimpleAssignmentTarget


MultiplicativeExpression :
    MultiplicativeExpression * UnaryExpression
    MultiplicativeExpression / UnaryExpression
    MultiplicativeExpression % UnaryExpression
1. Return false.

12.5.3 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 Inum be ToNumber(leftValue).
7. ReturnIfAbrupt(Inum).
8. Let rnum be ToNumber(rightValue).
9. ReturnIfAbrupt(rnum).
10. Return the result of applying the specified operation (*, /, or %) to Inum and rnum. See the Notes below 12.5.3.1, 12.5.3.2, 12.5.3.3.

12.5.3.1 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 nor 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.

12.5.3.2 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.

12.5.3.3 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.

12.6 Additive Operators

Syntax

AdditiveExpression\{Yield\}:
  MultiplicativeExpression\{Yield\}
  AdditiveExpression\{Yield\} + MultiplicativeExpression\{Yield\}
  AdditiveExpression\{Yield\} - MultiplicativeExpression\{Yield\}

12.6.1 Static Semantics: IsAnonymousFunctionDefinition

See also: 12.1.0.2, 12.1.10.2, 12.2.1.2, 12.3.2, 12.4.2, 12.5.1, 12.7.1, 12.8.1, 12.9.2, 12.10.2, 12.11.1, 12.12.1, 12.13.2, 12.14.2, 14.1.6, 14.4.5, 14.5.5.
AdditiveExpression:
  AdditiveExpression + MultiplicativeExpression
  AdditiveExpression - MultiplicativeExpression

1. Return false.

12.6.2 Static Semantics: IsValidSimpleAssignmentTarget


AdditiveExpression:
  AdditiveExpression + MultiplicativeExpression
  AdditiveExpression - MultiplicativeExpression

1. Return false.

12.6.3 The Addition operator (+)

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

12.6.3.1 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 12.6.5.

NOTE 1  No hint is provided in the calls to ToPrimitive in steps 7 and 9. All standard 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 11 differs from step 5 of the Abstract Relational Comparison algorithm (7.2.8), by using the logical-or operation instead of the logical-and operation.

12.6.4 The Subtraction Operator (-)

12.6.4.1 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. Return IfAbrupt(lnum).
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 12.6.5.

12.6.5 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.

NOTE 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 \( a + (-b) \).

12.7 Bitwise Shift Operators

Syntax

\[
\text{ShiftExpression} : \\
\text{AdditiveExpression} \| \text{Yield} \\
\text{ShiftExpression} \ll \text{AdditiveExpression} \| \text{Yield} \\
\text{ShiftExpression} \gg \text{AdditiveExpression} \| \text{Yield} \\
\text{ShiftExpression} \ggg \text{AdditiveExpression} \| \text{Yield}
\]

12.7.1 Static Semantics: IsAnonymousFunctionDefinition

See also: 12.1.0.2, 12.1.10.2, 12.2.1.2, 12.3.2, 12.4.2, 12.5.1, 12.6.1, 12.8.1, 12.9.2, 12.10.2, 12.11.1, 12.12.1, 12.13.2, 12.14.2, 14.1.6, 14.4.5, 14.5.5.

\[
\text{ShiftExpression} : \\
\text{ShiftExpression} \ll \text{AdditiveExpression} \\
\text{ShiftExpression} \gg \text{AdditiveExpression} \\
\text{ShiftExpression} \ggg \text{AdditiveExpression} \\
\]

1. Return \text{false}. 
12.7.2 Semantics: IsValidSimpleAssignmentTarget


**ShiftExpression**: 
- `ShiftExpression << AdditiveExpression`
- `ShiftExpression >> AdditiveExpression`
- `ShiftExpression >>> AdditiveExpression`

1. Return false.

12.7.3 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.

12.7.4 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.
12.7.5 The Unsigned Right Shift Operator ( \( \gg\gg \) )

NOTE Performs a zero-filling bitwise right shift operation on the left operand by the amount specified by the right operand.

12.7.5.1 Runtime Semantics: Evaluation

\( \text{ShiftExpression : ShiftExpression} \gg\gg \text{AdditiveExpression} \)

1. Let \( lref \) be the result of evaluating \( \text{ShiftExpression} \).
2. Let \( lval \) be \( \text{GetValue}(lref) \).
3. ReturnIfAbrupt(\( lval \)).
4. Let \( rref \) be the result of evaluating \( \text{AdditiveExpression} \).
5. Let \( rval \) be \( \text{GetValue}(rref) \).
6. ReturnIfAbrupt(\( rval \)).
7. Let \( lnum \) be \( \text{ToUint32}(lval) \).
8. ReturnIfAbrupt(\( lnum \)).
9. Let \( rnum \) be \( \text{ToUint32}(rval) \).
10. ReturnIfAbrupt(\( rnum \)).
11. Let \( \text{shiftCount} \) 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 \( \text{shiftCount} \) bits. Vacated bits are filled with zero. The result is an unsigned 32-bit integer.

12.8 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

\( \text{RelationalExpression}_{\text{In}} \), \( \text{Yield} \):

\( \text{ShiftExpression}_{\text{In}} \), \( \text{Yield} \) < \( \text{ShiftExpression}_{\text{In}} \)
\( \text{ShiftExpression}_{\text{In}} \) > \( \text{ShiftExpression}_{\text{In}} \)
\( \text{ShiftExpression}_{\text{In}} \) <= \( \text{ShiftExpression}_{\text{In}} \)
\( \text{ShiftExpression}_{\text{In}} \) >= \( \text{ShiftExpression}_{\text{In}} \)
\( \text{ShiftExpression}_{\text{In}} \) instanceof \( \text{ShiftExpression}_{\text{In}} \)
\( \text{ShiftExpression}_{\text{In}} \) in \( \text{ShiftExpression}_{\text{In}} \)

NOTE The \( \text{[in]} \) grammar parameter is needed to avoid confusing the \( \text{in} \) operator in a relational expression with the \( \text{in} \) operator in a for statement.

12.8.1 Static Semantics: IsAnonymousFunctionDefinition

See also: 12.1.0.2, 12.1.10.2, 12.2.1.2, 12.3.2, 12.4.2, 12.5.1, 12.6.1, 12.7.1, 12.9.2, 12.10.2, 12.11.1, 12.12.1, 12.13.2, 12.14.2, 14.1.6, 14.4.5, 14.5.5.

RelationalExpression:

\( \text{RelationalExpression} \leftarrow \text{ShiftExpression} \)
\( \text{RelationalExpression} \rightarrow \text{ShiftExpression} \)
\( \text{RelationalExpression} \leftarrow \text{ShiftExpression} \)
\( \text{RelationalExpression} \rightarrow \text{ShiftExpression} \)
\( \text{RelationalExpression} \leftarrow \text{ShiftExpression} \)
\( \text{RelationalExpression} \rightarrow \text{ShiftExpression} \)
\( \text{RelationalExpression} \text{ instanceof} \text{ShiftExpression} \)
\( \text{RelationalExpression} \text{ in} \text{ShiftExpression} \)
1. Return false.

12.8.2 Static Semantics: IsValidSimpleAssignmentTarget


RelationalExpression:
RelationalExpression < ShiftExpression
RelationalExpression <= ShiftExpression
RelationalExpression == ShiftExpression
RelationalExpression instanceof ShiftExpression
RelationalExpression in ShiftExpression

1. Return false.

12.8.3 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 7.2.8)
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 \textbf{true} or \textbf{undefined}, return \textbf{false}. Otherwise, return \textbf{true}.

RelationalExpression : RelationalExpression \textbf{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 InstanceofOperator(\( lval \), \( rval \)).

RelationalExpression : RelationalExpression \textbf{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 \textbf{TypeError} exception.
8. Return the result of HasProperty(\( rval \), ToPropertyKey(\( lval \))).

12.8.4 Runtime Semantics: InstanceofOperator(\( O \), \( C \))

The abstract operation InstanceofOperator(\( O \), \( C \)) 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 \textbf{TypeError} exception.
2. Let \( instOfHandler \) be the result of GetMethod(\( C \), \text{ @@hasInstance }).
3. ReturnIfAbrupt(\( instOfHandler \)).
4. If \( instOfHandler \) is not \textbf{undefined}, then
   a. Let \( result \) be the result of calling the \text{ [[Call]] } internal method of \( instOfHandler \) passing \( C \) as thisArgument and a new List containing \( O \) as argumentsList.
   b. Return ToBoolean(\( result \)).
5. If IsCallable(\( C \)) is \textbf{false}, then throw a \textbf{TypeError} exception.
6. Return the result of OrdinaryHasInstance(\( C \), \( O \)).

\textbf{NOTE} Steps 5 and 6 provide compatibility with previous editions of ECMAScript that did not use a \text{ @@hasInstance } method to define the \textbf{instanceof} operator semantics. If a function object does not define or inherit \text{ @@hasInstance } it uses the default \textbf{instanceof} semantics.

12.9 Equality Operators

\textbf{NOTE} The result of evaluating an equality operator is always of type \textbf{Boolean}, reflecting whether the relationship named by the operator holds between its two operands.
Syntax

EqualityExpression[?yield]:
  RelationalExpression[?yield] == RelationalExpression[?yield]
  RelationalExpression[?yield] != RelationalExpression[?yield]
  RelationalExpression[?yield] === RelationalExpression[?yield]
  RelationalExpression[?yield] !== RelationalExpression[?yield]

12.9.1 Static Semantics: IsAnonymousFunctionDefinition

See also: 12.1.0.2, 12.1.10.2, 12.2.1.2, 12.3.2, 12.4.2, 12.5.1, 12.6.1, 12.7.1, 12.8.1, 12.10.2, 12.11.1, 12.12.1, 12.13.2, 12.14.2, 14.1.6, 14.4.5, 14.5.5.

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

1. Return false.

12.9.2 Static Semantics: IsValidSimpleAssignmentTarget


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

1. Return false.

12.9.3 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 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 rval == lval.
8. If r is true, return false. Otherwise, return true.

EqualityExpression : EqualityExpression === RelationalExpression

© Ecma International 2009 – All rights reserved
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 Strict Equality Comparison 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 rval === lval.
8. If r is true, return false. Otherwise, return true.

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 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.

12.10 Binary Bitwise Operators

Syntax

BitwiseANDExpression<y, yval> :
  EqualityExpression<y, yval> & EqualityExpression<y, yval>

BitwiseXORExpression<y, yval> :
  BitwiseANDExpression<y, yval> ^ BitwiseANDExpression<y, yval>

BitwiseORExpression<y, yval> :
  BitwiseXORExpression<y, yval> | BitwiseXORExpression<y, yval>
12.10.1 Static Semantics: IsAnonymousFunctionDefinition

See also: 12.1.0.2, 12.1.10.2, 12.2.1.2, 12.3.2, 12.4.2, 12.5.1, 12.6.1, 12.7.1, 12.8.1, 12.9.2, 12.10.2, 12.11.1, 12.12.1, 12.13.2, 12.14.2, 14.1.6, 14.4.5, 14.5.5.

BitwiseANDExpression : BitwiseANDExpression & EqualityExpression
BitwiseXORExpression : BitwiseXORExpression ^ BitwiseANDExpression
BitwiseORExpression : BitwiseORExpression | BitwiseXORExpression

1. Return false.

12.10.2 Static Semantics: IsValidSimpleAssignmentTarget


BitwiseANDExpression : BitwiseANDExpression & EqualityExpression
BitwiseXORExpression : BitwiseXORExpression ^ BitwiseANDExpression
BitwiseORExpression : BitwiseORExpression | BitwiseXORExpression

1. Return false.

12.10.3 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.

12.11 Binary Logical Operators

Syntax
LogicalANDExpression, γval1 : LogicalORExpression[γval1, γval2] && LogicalORExpression[γval1, γval2]
LogicalORExpression, γval1 : LogicalANDExpression[γval1, γval2] || LogicalANDExpression[γval1, γval2]

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.

12.11.1 Static Semantics: IsAnonymousFunctionDefinition

See also: 12.1.0.2, 12.1.10.2, 12.2.1.2, 12.3.2, 12.4.2, 12.5.1, 12.6.1, 12.7.1, 12.8.1, 12.9.2, 12.10.2, 12.11.1, 12.12.1, 12.13.2, 12.14.2, 14.1.6, 14.4.5, 14.5.5.
LogicalANDExpression : LogicalANDExpression && BitwiseORExpression
LogicalORExpression : LogicalORExpression || LogicalANDExpression

1. Return false.

12.11.2 Static Semantics: IsValidSimpleAssignmentTarget

LogicalANDExpression : LogicalANDExpression && BitwiseORExpression
LogicalORExpression : LogicalORExpression || LogicalANDExpression

1. Return false.

12.11.3 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).

12.12 Conditional Operator ( ? : )

Syntax

ConditionalExpression[?][?][?]
   : LogicalORExpression[?][?][?]
   LogicalORExpression[?][?][?]
   LogicalExpression[?][?][?]
   LogicalExpression[?][?][?]
   AssignmentExpression[?][?][?]
   AssignmentExpression[?][?][?]

NOTE The grammar for a ConditionalExpression in ECMAScript is slightly 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.

12.12.1 Static Semantics: IsAnonymousFunctionDefinition
See also: 12.1.0.2, 12.1.10.2, 12.2.1.2, 12.3.2, 12.4.2, 12.5.1, 12.6.1, 12.7.1, 12.8.1, 12.9.2, 12.10.2, 12.11.1, 12.13.2, 12.14.2, 14.1.6, 14.4.5, 14.5.5.
ConditionalExpression : LogicalORExpression ? AssignmentExpression : AssignmentExpression

1. Return false.

12.12.2 Static Semantics: IsValidSimpleAssignmentTarget


ConditionalExpression : LogicalORExpression ? AssignmentExpression : AssignmentExpression

1. Return false.

12.12.3 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).

12.13 Assignment Operators

Syntax

AssignmentExpression[Yield]:
  ConditionalExpression[?In, ?Yield]
  [?Yield] YieldExpression
  ArrowFunction[?In]
  LeftHandSideExpression[?Yield] = AssignmentExpression[?In, ?Yield]
  LeftHandSideExpression[?Yield] AssignmentOperator AssignmentExpression[?In, ?Yield]

AssignmentOperator : one of
  * = / = %= += -= <<= >>= >>>= &= ^= |=

12.13.1 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 IdentifierReference 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(LeftHandSideExpression) is false.

12.13.2 Static Semantics: IsValidAnonymousFunctionDefinition

See also: 12.1.0.2, 12.1.10.2, 12.2.1.2, 12.3.2, 12.4.2, 12.5.1, 12.6.1, 12.7.1, 12.8.1, 12.9.2, 12.10.2, 12.11.1, 12.12.1, 12.14.2, 14.1.6, 14.4.5, 14.5.5.

AssignmentExpression : ArrowFunction
1. Return true.

AssignmentExpression : YieldExpression
  LeftHandSideExpression = AssignmentExpression
  LeftHandSideExpression AssignmentOperator AssignmentExpression
1. Return false.

12.13.3 Static Semantics: IsValidSimpleAssignmentTarget


AssignmentExpression : YieldExpression
  ArrowFunction
  LeftHandSideExpression = AssignmentExpression
  AssignmentOperator AssignmentExpression
1. Return false.

12.13.4 Runtime Semantics: Evaluation


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. If both IsAnonymousFunctionDefinition of AssignmentExpression and IsIdentifierRef of LeftHandSideExpression are true, then
      i. Let hasNameProperty be the result of HasOwnProperty(rval, "name").
      ii. ReturnIfAbrupt(hasNameProperty).
      iii. If hasNameProperty is false, then
          1. SetFunctionName(rval, GetReferencename(lref)).
      f. Let status be PutValue(lref, rval).
      g. ReturnIfAbrupt(status).
      h. Return rval.
   2. Let AssignmentPattern be the parse of the source code corresponding to LeftHandSideExpression using AssignmentPattern[?Yield] as the goal symbol.
3. Let rref be the result of evaluating AssignmentExpression.
4. Let rval be GetValue(rref).
5. ReturnIfAbrupt(rval).
6. If Type(rval) is not Object, then throw a TypeError exception.
7. Let status be the result of performing DestructuringAssignmentEvaluation of AssignmentPattern using rval as the argument.
8. ReturnIfAbrupt(status).
9. 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 or step 9 of the second algorithm it is a 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 for which the IsExtensible predicate returns the value false. In these cases a TypeError exception is thrown.

12.13.5 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\[yield\] : ObjectAssignmentPattern\[yield\]
   ArrayAssignmentPattern\[yield\]
ObjectAssignmentPattern\[yield\] : 
   { } 
   { AssignmentPropertyList\[yield\] } 
   { AssignmentPropertyList\[yield\] , } 
ArrayAssignmentPattern\[yield\] : 
   [ Elision\[yield\] AssignmentRestElement\[yield\] ] 
   [ AssignmentElementList\[yield\] ] 
   [ AssignmentElementList\[yield\] , Elision\[yield\] AssignmentRestElement\[yield\] ]
AssignmentPropertyList\[yield\] : 
   AssignmentProperty\[yield\] 
   AssignmentProperty\[yield\] , AssignmentProperty\[yield\]
AssignmentElementList\[yield\] : 
   AssignmentElement\[yield\] 
   AssignmentElement\[yield\] , AssignmentElement\[yield\]
AssignmentRestElement\[yield\] : 
   Elision\[yield\] AssignmentElement\[yield\]
AssignmentProperty\(\{y\}\) :
  IdentifierReference\(\{y\}\) Initialiser\(\{y\}\)

AssignmentElement\(\{y\}\) :
  DestructuringAssignmentTarget\(\{y\}\) Initialiser\(\{y\}\)

AssignmentRestElement\(\{y\}\) :
  \ldots , DestructuringAssignmentTarget\(\{y\}\)

DestructuringAssignmentTarget\(\{y\}\) :
  LeftHandSideExpression\(\{y\}\)

12.13.5.1 Static Semantics: Early Errors

AssignmentProperty : IdentifierReference Initialiser\(\{y\}\)

- It is a Syntax Error if IsValidSimpleAssignment(IdentifierReference) is false.
- It is a Syntax Error if IdentifierReference statically resolves to a mutable binding.

AssignmentRestElement : \ldots , DestructuringAssignmentTarget

- It is a Syntax Error if IsValidSimpleAssignmentTarget of DestructuringAssignmentTarget is false.

DestructuringAssignmentTarget : LeftHandSideExpression

- 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.
- It is a Syntax Error if LeftHandSideExpression is neither an ObjectLiteral nor an ArrayLiteral and IsValidSimpleAssignmentTarget(LeftHandSideExpression) is false.
- It is a Syntax Error if LeftHandSideExpression is an IdentifierReference 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 CoverParenthesisedExpressionAndArrowParameterList : ( Expression ) and Expression derives a production that would produce a Syntax Error according to these rules if that production is substituted for LeftHandSideExpression. This rule is recursively applied.

NOTE The last rule means that the other rules are applied even if multiple levels of nested parentheses surround Expression.

12.13.5.2 Runtime Semantics: DestructuringAssignmentEvaluation

with parameter obj

ObjectAssignmentPattern : { }
  1. Return NormalCompletion(empty).

ArrayAssignmentPattern : [ ]
  1. Let iterator be the result of performing GetIterator(obj).
2. `ReturnIfAbrupt(iterator)`.
3. `Return NormalCompletion(empty)`.

**ArrayAssignmentPattern**: `[Elision]`

1. Let `iterator` be the result of performing `GetIterator(obj)`.
2. `ReturnIfAbrupt(iterator)`.
3. Return the result of performing `IteratorDestructuringAssignmentEvaluation` of `Elision` with `iterator` as the argument.

**ArrayAssignmentPattern**: `[ Elision, AssignmentRestElement ]`

1. Let `iterator` be the result of performing `GetIterator(obj)`.
2. `ReturnIfAbrupt(iterator)`.
3. If `Elision` is present, then
   a. Let `status` be the result of performing `IteratorDestructuringAssignmentEvaluation` of `Elision` with `iterator` as the argument.
   b. `ReturnIfAbrupt(status)`.
4. Return the result of performing `IteratorDestructuringAssignmentEvaluation` of `AssignmentRestElement` with `iterator` as the argument.

**ArrayAssignmentPattern**: `[ AssignmentElementList ]`

1. Let `iterator` be the result of performing `GetIterator(obj)`.
2. `ReturnIfAbrupt(iterator)`.
3. Return the result of performing `IteratorDestructuringAssignmentEvaluation` of `AssignmentElementList` using `iterator` as the argument.

**ArrayAssignmentPattern**: `[ AssignmentElementList, Elision?, AssignmentRestElement? ]`

1. Let `iterator` be the result of performing `GetIterator(obj)`.
2. `ReturnIfAbrupt(iterator)`.
3. Let `status` be the result of performing `IteratorDestructuringAssignmentEvaluation` of `AssignmentElementList` using `iterator` as the argument.
4. `ReturnIfAbrupt(status)`.
5. If `Elision` is present, then
   a. Let `status` be the result of performing `IteratorDestructuringAssignmentEvaluation` of `Elision` with `iterator` as the argument.
   b. `ReturnIfAbrupt(status)`.
6. If `AssignmentRestElement` is present, then return the result of performing `IteratorDestructuringAssignmentEvaluation` of `AssignmentRestElement` with `iterator` as the argument.
7. Return `lastIndex`.

**AssignmentPropertyList**: `AssignmentPropertyList`, `AssignmentProperty`

1. Let `status` be the result of performing `DestructuringAssignmentEvaluation` for `AssignmentPropertyList` using `obj` as the argument.
2. `ReturnIfAbrupt(status)`.
3. Return the result of performing `DestructuringAssignmentEvaluation` for `AssignmentProperty` using `obj` as the argument.

**AssignmentProperty**: `IdentifierReference Initialiser?`

1. Let `P` be `StringValue` of `IdentifierReference`.
2. Let `v` be the result of calling `Get(obj, P)`.
3. `ReturnIfAbrupt(v)`.
4. If `Initialiser` is present and `v` is `undefined`, then
   a. Let `defaultValue` be the result of evaluating `Initialiser`.
   b. Let `v` be `ToObject(GetValue(defaultValue))`.

© Ecma International 2009 – All rights reserved
5. ReturnIfAbrupt(v).
6. Let lref be the result of ResolveBinding(P).

AssignmentProperty : PropertyName : AssignmentElement
1. Let name be the result of evaluating PropertyName.
2. ReturnIfAbrupt(name).
3. Return the result of performing KeyedDestructuringAssignmentEvaluation of AssignmentElement with obj and name as the arguments.

12.13.5.3 Runtime Semantics: IteratorDestructuringAssignmentEvaluation

with parameters iterator

AssignmentElementList : AssignmentElisionElement
1. Return the result of performing IteratorDestructuringAssignmentEvaluation of AssignmentElisionElementList using iterator as the argument.

AssignmentElementList : AssignmentElementList , AssignmentElisionElement
1. Let status be the result of performing IteratorDestructuringAssignmentEvaluation of AssignmentElementList using iterator as the argument.
2. ReturnIfAbrupt(status).
3. Return the result of performing IteratorDestructuringAssignmentEvaluation of AssignmentElisionElementList using iterator as the argument.

AssignmentElisionElement : AssignmentElement
1. Return the result of performing IteratorDestructuringAssignmentEvaluation of AssignmentElement with iterator as the argument.

AssignmentElisionElement : Elision AssignmentElement
1. Let status be the result of performing IteratorDestructuringAssignmentEvaluation of Elision with iterator as the argument.
2. ReturnIfAbrupt(status).
3. Return the result of performing IteratorDestructuringAssignmentEvaluation of AssignmentElement with iterator as the argument.

Elision : ,
1. Return the result of IteratorStep(iterator).

Elision : Elision ,
1. Let status be the result of performing IteratorDestructuringAssignmentEvaluation of Elision with iterator as the argument.
2. ReturnIfAbrupt(status).
3. Return the result of IteratorStep(iterator).

AssignmentElement : DestructuringAssignmentTarget Initialiser_opt
1. If DestructuringAssignmentTarget is neither an ObjectLiteral nor an ArrayLiteral then
a. Let lref be the result of evaluating DestructuringAssignmentTarget.
   b. ReturnIfAbrupt(lref).
2. Let next be the result of IteratorStep(iterator).
3. ReturnIfAbrupt(next).
4. If next is false, then let v be undefined
5. Else
   a. Let v be IteratorValue(next).
   b. ReturnIfAbrupt(v).
6. If InitialiserOpt is present and v is undefined, then
   a. Let defaultValue be the result of evaluating Initialiser.
   b. Let v be GetValue(defaultValue)
   c. ReturnIfAbrupt(v).
7. If DestructuringAssignmentTarget is an ObjectLiteral or an ArrayLiteral then
   a. Let nestedAssignmentPattern be the parse of the source code corresponding to
   DestructuringAssignmentTarget using AssignmentPattern[?Yield] as the goal symbol
   b. If Type(v) is not Object, then throw a TypeError exception.
   c. Return the result of performing DestructuringAssignmentEvaluation of nestedAssignmentPattern
      with v as the argument.
8. Else,

NOTE Left to right evaluation order is maintained by evaluating a
DestructuringAssignmentTarget that is not a
destructuring pattern prior to accessing the iterator or evaluating the Initialiser.

AssignmentRestElement : \... DestructuringAssignmentTarget
1. Let lref be the result of evaluating DestructuringAssignmentTarget.
2. ReturnIfAbrupt(lref).
3. Let A be the result of the abstract operation ArrayCreate with argument 0.
4. Let n=0;
5. Repeat
   a. Let next be the result of IteratorStep(iterator).
   b. ReturnIfAbrupt(next).
   c. If next is false, then
      i. Return PutValue(lref, A).
   d. Let nextValue be IteratorValue(next).
   e. ReturnIfAbrupt(nextValue).
   f. Let defineStatus be the result of CreateDataPropertyOrThrow(A, ToString(ToUint32(n)), nextValue).
   g. ReturnIfAbrupt(defineStatus).
     h. Increment n by 1.

12.13.5.4 Runtime Semantics: KeyedDestructuringAssignmentEvaluation

with parameters obj and propertyName

AssignmentElement : DestructuringAssignmentTarget Initialiser \=0
1. Let v be the result of Get(obj, name).
2. ReturnIfAbrupt(v).
3. If InitialiserOpt is present and v is undefined, then
   a. Let defaultValue be the result of evaluating Initialiser.
   b. Let v be GetValue(defaultValue)
   c. ReturnIfAbrupt(v).
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[?Yield] as the goal symbol
   b. If Type(v) is not Object, then throw a TypeError exception.
   c. Return the result of performing DestructuringAssignmentEvaluation of AssignmentPattern with v as
      the argument.
5. Let lref be the result of evaluating DestructuringAssignmentTarget.

Commented [AW48]: Note that indices wrap. For example consider:
[...... {0: 'x', length: Math.pow(2,32)-2}]
6. Return `PutValue(lref, v)`.

12.14 Comma Operator ( , )

Syntax

```
Expression, yield :
  AssignmentExpression, yield
  Expression, AssignmentExpression, yield
```

12.14.1 Static Semantics: `IsAnonymousFunctionDefinition`

See also: 12.1.0.2, 12.1.10.2, 12.2.1.2, 12.3.2, 12.4.2, 12.5.1, 12.6.1, 12.7.1, 12.8.1, 12.9.2, 12.10.2, 12.11.1, 12.12.1, 12.13.2, 14.1.6, 14.4.5, 14.5.5.

Expression : Expression , AssignmentExpression

1. Return `false`.

12.14.2 Static Semantics: `IsValidSimpleAssignmentTarget`

See also: 12.1.0.3, 12.1.10.3, 12.2.1.3, 12.3.3, 12.4.3, 12.5.2, 12.6.2, 12.7.2, 12.8.2, 12.9.2, 12.10.2, 12.11.2, 12.12.2, 12.13.3.

Expression : Expression , AssignmentExpression

1. Return `false`.

12.14.3 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.
13 ECMAScript Language: Statements and Declarations

Syntax

Statement::=
  BlockStatement[Yield, Return]
  VariableStatement[Yield]
  EmptyStatement
  ExpressionStatement[Yield]
  IfStatement[Yield, Return]
  BreakableStatement[Yield, Return]
  ContinueStatement
  BreakStatement
  [+]ReturnStatement
  WithStatement[Yield, Return]
  TryStatement[Yield, Return]
  DebuggerStatement

Declaration::=
  FunctionDeclaration[Yield, Default]
  GeneratorDeclaration[Yield, Default]
  ClassDeclaration[Yield]
  LexicalDeclaration[Yield]

BreakableStatement::=
  IterationStatement[Yield, Return]
  SwitchStatement[Yield, Return]

13.0 Statement Semantics

13.0.1 Static Semantics: VarDeclaredNames

See also: 13.1.8, 13.5.1, 13.6.1.1, 13.6.2.1, 13.6.3.1, 13.6.4.3, 13.10.2, 13.11.4, 13.12.2, 13.14.2, 14.1.11, 14.4.10, 14.5.14, 15.1.0.12, 15.2.5.

Statement::=
  EmptyStatement
  ExpressionStatement
  ContinueStatement
  BreakStatement
  ReturnStatement
  ThrowStatement
  DebuggerStatement

1. Return a new empty List.

13.0.2 Runtime Semantics: LabelledEvaluation

With argument labelSet.

See also: 13.6.1.2, 13.6.2.2, 13.6.3.2, 13.6.4.5, 13.12.3.

BreakableStatement::= IterationStatement

1. Let stmtResult be the result of performing LabelledEvaluation of IterationStatement with argument labelSet.
2. If stmtResult.[[type]] is break and stmtResult.[[target]] is empty, then
   a. If stmtResult.[[value]] is empty, then let stmtResult be NormalCompletion(undefined).
   b. Else, let stmtResult be NormalCompletion(stmtResult.[[value]])
3. Return stmtResult.

**BreakableStatement** : SwitchStatement

1. Let stmtResult be the result of evaluating SwitchStatement.
2. If stmtResult.[[type]] is break and stmtResult.[[target]] is empty, then
   a. If stmtResult.[[value]] is empty, then let stmtResult be NormalCompletion(undefined).
   b. Else, let stmtResult be NormalCompletion(stmtResult.[[value]])
3. Return stmtResult.

**NOTE** A BreakableStatement is one that can be exited via an unlabelled BreakStatement.

13.0.3 Runtime Semantics: Evaluation

**BreakableStatement** : IterationStatement

1. Let newLabelSet be a new empty List.
2. Return the result of performing LabelledEvaluation of this BreakableStatement with argument newLabelSet.

13.1 Block

**Syntax**

```
BlockStatement[Yield, Return] : Block[Yield, Return]

Block[Yield, Return] :
{ StatementList[Yield, Return][opt] }
```

```
StatementList[Yield, Return] :
StatementListitem[Yield, Return][opt]

StatementListitem[Yield, Return] :
Statement[Yield, Return]
Declarationitem[Yield]
```

13.1.1 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.

13.1.2 Static Semantics: LexicalDeclarations

See also: 13.11.2, 15.1.0.10.

**StatementList : StatementList StatementListItem**

1. Let declarations be LexicalDeclarations of StatementList.
2. Append to declarations the elements of the LexicalDeclarations of StatementListItem.
3. Return declarations.
1. Return a new empty List.

StatementListItem : Declaration
1. Return a new List containing Declaration.

13.1.3 Static Semantics: LexicallyDeclaredNames
See also: 13.11.3, 14.1.10, 14.2.7, 14.4.7, 14.5.8, 15.1.0.7, 15.2.3.

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.

StatementList : StatementList StatementListItem
1. Let declarations be TopLevelLexicallyScopedDeclarations of StatementList.
2. Append to declarations the elements of the TopLevelLexicallyScopedDeclarations of StatementListItem.
3. Return declarations.

NOTE At the top level of a function, or script, function declarations are treated like var declarations rather than like lexical declarations.

13.1.5 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.
1. Return a new empty List.

   StatementListItem : Declaration
   1. If Declaration is Declaration : FunctionDeclaration, then return a new empty List.
   2. If Declaration is Declaration : GeneratorDeclaration, then return a new empty List.
   3. Return a new List containing Declaration.

13.1.6 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. If Declaration is Declaration : GeneratorDeclaration, then return the LexicallyDeclaredNames of Declaration.
   3. Return a new empty List.

13.1.7 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. Return VarDeclaredNames of Statement.

   NOTE At the top level of a function or script, inner function declarations are treated like var declarations.

13.1.8 Static Semantics: VarDeclaredNames

See also: 13.0.1, 13.5.1, 13.6.1.1, 13.6.2.1, 13.6.3.1, 13.6.4.3, 13.10.2, 13.11.4, 13.12.2, 13.14.2, 14.1.11, 14.4.10, 14.5.14, 15.1.0.12, 15.2.5.

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

1. Return a new empty List.

13.1.9 Static Semantics: VarScopedDeclarations

See also: 15.1.0.13, 15.2.6.

StatementList : StatementList StatementListItem

1. Let declarations be VarScopedDeclarations of StatementList.
2. Append to declarations the elements of the VarScopedDeclarations 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.

13.1.10 Runtime Semantics: Evaluation

Block : { }

1. Return NormalCompletion(undefined).

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. If blockValue.[[type]] is normal and blockValue.[[value]] is empty, then
   a. Return NormalCompletion(undefined).
8. 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 5 and 6 of the above algorithm ensure that the value of a StatementList is the value of the last value producing Statement in the StatementList. For example, the following calls to the eval function all return the value 1:

\[
\text{eval}('1; ; ; ;')
\]
\[
\text{eval}('1; \{\};')
\]
\[
\text{eval}('1; \text{var } a;')
\]

### 13.1.11 Runtime Semantics: 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. Let functionsToInitialise be an empty List.
3. 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. Let status be the result of calling env’s CreateMutableBinding concrete method passing `dn` and false as the arguments.  
         2. Assert: status is never an abrupt completion.
   b. If `d` is a GeneratorDeclaration or a FunctionDeclaration production, then
      1. Append `d` to functionsToInitialise.
4. For each production `f` in functionsToInitialise, 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 InitialiseBinding concrete method passing `fn`, and `fo` as the arguments.

### 13.2 Declarations and the Variable Statement

#### 13.2.1 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 Initialiser the variable is assigned the value undefined when the LexicalBinding is evaluated.

**Syntax**

\[
\text{let} \text{OrConst BindingList[In, Yield]:}
\]

\[
\text{let}
\]

\[
\text{const}
\]

**Commented [AWB1352]:** ISSUE: above changes to completion reform will mean this evaluates to undefined rather than 1 is

**Commented [AWB753]:** Jan 19 meeting notes: Current tentative decision is to support let, const, and local functions in nonstrict ES5 in the same way as in strict ES6.Fallback to either specifying limited cases or doing the ES5 nonstrict status quo (i.e. syntax error + Clause 16) if experiments show this to not be viable. We won’t resolve this discussion without running some experiments.
13.2.1.1 Static Semantics: Early Errors

LexicalBinding : BindingIdentifier Initialiser

- It is a Syntax Error if the BoundNames of BindingIdentifier contains "let".
- It is a Syntax Error if Initialiser is not present and IsConstantDeclaration of the LexicalDeclaration containing this production is true.

LexicalBinding : yield

- It is a Syntax Error if this production is contained in strict code.
- It is a Syntax Error if this production is contained within the FunctionBody of a GeneratorMethod, GeneratorDeclaration, or GeneratorExpression.

LexicalBinding : Identifier

- It is a Syntax Error if the BindingIdentifier is contained in strict code and if the Identifier is: arguments, eval, implements, interface, let, package, private, protected, public, or static.

13.2.1.2 Static Semantics: BoundNames

See also: 13.2.3.2, 13.6.4.2, 14.1.2, 14.2.2, 14.4.2, 14.5.2, 15.1.1.2, 0.

LexicalDeclaration : LetOrConst BindingList ;

1. Return the BoundNames of BindingList.

LexicalBinding : 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

1. Return the BoundNames of BindingIdentifier.

LexicalBinding : BindingPattern Initialiser

1. Return the BoundNames of BindingPattern.

LexicalBinding : Identifier

1. Return a new List containing the StringValue of Identifier.

LexicalBinding : default

1. Return a new List containing the string "default".
1. Return a new List containing the string "yield".

13.2.1.3 Static Semantics: IsConstantDeclaration

See also: 14.1.6, 14.4.4, 14.5.5.

LexicalDeclaration : LetOrConst BindingList ;
1. Return IsConstantDeclaration of LetOrConst.

LetOrConst : let
1. Return false.

LetOrConst : const
1. Return true.

13.2.1.4 Static Semantics: StringValue

See also: 11.6.1.1.

BindingIdentifier : Identifier
1. Return the StringValue of Identifier.

BindingIdentifier : default
1. Return the string "default".

BindingIdentifier : yield
1. Return the string "yield".

13.2.1.5 Runtime Semantics: BindingInitialisation

With arguments value and environment.

See also: 12.1.4.2.2, 13.2.2.2, 13.2.3.4, 13.14.3.

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 preinitialised prior to evaluation of its initialiser.

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 InitialiseBinding concrete method of env passing name and value as the arguments.
   d. Return NormalCompletion(undefined).
2. Else
   a. Let lhs be the result of ResolveBinding(StringValue(Identifier)).
b. Return PutValue(lhs, value).

13.2.1.6 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 BindingInitialisation 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 : Initialiser
1. Let rhs be the result of evaluating Initialiser.
2. Let value be GetValue(rhs).
3. ReturnIfAbrupt(value).
4. If IsAnonymousFunctionDefinition of Initialiser is true, then
   a. Let hasNameProperty be the result of HasOwnProperty(value, "name").
   b. ReturnIfAbrupt(hasNameProperty).
   c. If hasNameProperty is false, then
      i. SetFunctionName(value, StringValue(BindingIdentifier)).
5. Let env be the running execution context’s LexicalEnvironment.
6. Return the result of performing BindingInitialisation for BindingIdentifier passing value and env as the arguments.

LexicalBinding : BindingPattern Initialiser
1. Let rhs be the result of evaluating Initialiser.
2. Let value be GetValue(rhs).
3. ReturnIfAbrupt(value).
4. If Type(value) is not Object, then throw a TypeError exception.
5. Let env be the running execution context’s LexicalEnvironment.
6. Return the result of performing BindingInitialisation for BindingPattern using value and env as the arguments.

13.2.2 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 VariableEnvironment a common Identifier may appear in more than one VariableDeclaration but those declarations collective define only one variable. A variable defined by a VariableDeclaration with an Initialiser is assigned the value of its Initialiser’s AssignmentExpression when the VariableDeclaration is executed, not when the variable is created.
Syntax

```
VariableStatement : 
  var VariableDeclarationList ;
```

```
VariableDeclarationList , VariableDeclaration :
  VariableDeclaration , VariableDeclarationList ;

VariableDeclaration , VariableDeclaration :
  VariableDeclaration ;
```

```
VariableDeclaration , VariableDeclaration :
  BindingIdentifier Initialiser ;
```

```
VariableDeclaration , VariableDeclaration :
  BindingPattern Initialiser ;
```

### 13.2.2.1 Static Semantics: BoundNames

See also: 13.2.1.2, 13.2.3.2, 13.6.4.2, 14.1.2, 14.2.2, 14.4.2, 14.5.2, 15.1.1.2, 0.

```
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 :
  1. Return the BoundNames of BindingIdentifier.
```

```
VariableDeclaration : BindingPattern Initialiser :
  1. Return the BoundNames of BindingPattern.
```

### 13.2.2.2 Runtime Semantics: BindingInitialisation

With arguments \textit{value} and \textit{environment}.

See also: 12.1.4.2.2, 13.2.1.5, 13.2.3.4, 13.14.3.

**NOTE** \textit{undefined} is passed for \textit{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 preinitialised prior to evaluation of its initialiser.

```
VariableDeclaration : BindingIdentifier :
  1. Return the result of performing BindingInitialisation for BindingIdentifier passing value and undefined as the arguments.
```

```
VariableDeclaration : BindingIdentifier Initialiser :
  1. Return the result of performing BindingInitialisation for BindingIdentifier passing value and undefined as the arguments.
```

```
VariableDeclaration : BindingPattern Initialiser :
  1. Return the result of performing BindingInitialisation for BindingPattern passing value and undefined as the arguments.
```
13.2.2.3 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. If IsAnonymousFunctionDefinition of Initialiser is true, then
   a. Let hasNameProperty be the result of HasOwnProperty(value, "name").
   b. ReturnIfAbrupt(hasNameProperty).
   c. If hasNameProperty is false, then
      i. SetFunctionName(value, StringValue(BindingIdentifier)).
5. Return the result of performing BindingInitialisation 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
1. Let rhs be the result of evaluating Initialiser.
2. Let rval be GetValue(rhs).
3. ReturnIfAbrupt(rval).
4. If Type(rval) is not Object, then throw a TypeError exception.
5. Return the result of performing BindingInitialisation for BindingPattern passing rval and undefined as arguments.

13.2.3 Destructuring Binding Patterns

Syntax

- `BindingPattern[Yield,GeneratorParameter]:
- ObjectBindingPattern[Yield]
- ArrayBindingPattern[Yield]`

- `ObjectBindingPattern[Yield,GeneratorParameter]:
  { }
  { BindingPropertyList[Yield,GeneratorParameter] }
  { BindingPropertyList[Yield,GeneratorParameter] , }`
13.2.3.1 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".

13.2.3.2 Static Semantics: BoundNames

See also: 13.2.1.2, 13.2.2.1, 13.6.4.2, 14.1.2, 14.2.2, 14.4.2, 14.5.2, 15.1.1.2, 0.

**ObjectBindingPattern**: { }

1. Return an empty List.

**ArrayBindingPattern**: [ Elisionopt ]
1. Return an empty List.

ArrayBindingPattern : [ Elisionopt BindingRestElement ]
1. Return the BoundNames of BindingRestElement.

ArrayBindingPattern : [ BindingElementList , Elisionopt ]
1. Return the BoundNames of BindingElementList.

ArrayBindingPattern : [ BindingElementList , Elisionopt 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 : BindingElementList , BindingElisionElement
1. Let names be BoundNames of BindingElementList.
2. Append to names the elements of BoundNames of BindingElisionElement.
3. Return names.

BindingElisionElement : Elisionopt BindingElement
1. Return BoundNames of BindingElement.

BindingProperty : PropertyName : BindingElement
1. Return the BoundNames of BindingElement.

SingleNameBinding : BindingIdentifier Initialiseropt
1. Return the BoundNames of BindingIdentifier.

BindingElement : BindingPattern Initialiseropt
1. Return the BoundNames of BindingPattern.

13.2.3.3 Static Semantics: HasInitialiser

See also: 14.1.5.

BindingElement : BindingPattern
1. Return false.

BindingElement : BindingPattern Initialiser
1. Return true.

SingleNameBinding : BindingIdentifier
1. Return false.
SingleNameBinding : BindingIdentifier Initialiser

1. Return true.

13.2.3.4 Runtime Semantics: BindingInitialisation

   With parameters value and environment.

See also: 12.1.4.2.2, 13.2.1.5, 13.2.2.2, 13.14.3.

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 preinitialised 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 BindingInitialisation for ObjectBindingPattern using value and environment as arguments.

BindingPattern : ArrayBindingPattern

1. Assert: Type(value) is Object
2. Let iterator be the result of performing GetIterator(obj).
3. ReturnIfAbrupt(iterator).
4. Return the result of performing IteratorBindingInitialisation for ArrayBindingPattern using iterator, and environment as arguments.

ObjectBindingPattern : { }

1. Return NormalCompletion(empty).

BindingPropertyList : BindingPropertyList , BindingProperty

1. Let status be the result of performing BindingInitialisation for BindingPropertyList using value and environment as arguments.
2. ReturnIfAbrupt(status).
3. Return the result of performing BindingInitialisation 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 KeyedBindingInitialisation for SingleNameBinding using value, environment, and name as the arguments.

BindingProperty : PropertyName : BindingElement

1. Let P be the result of evaluating PropertyName
2. ReturnIfAbrupt(P).
3. Return the result of performing KeyedBindingInitialisation for BindingElement using value, environment, and P as arguments.

13.2.3.5 Runtime Semantics: IteratorBindingInitialisation

   With parameters iterator, and environment.
See also: 14.1.13, .

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 preinitialised in order to deal with the possibility of multiple parameters with the same name.

ArrayBindingPattern : [ ]
1. Return NormalCompletion(empty).

ArrayBindingPattern : [ Elision ]
1. Return the result of performing IteratorDestructuringAssignmentEvaluation of Elision with iterator as the argument.

ArrayBindingPattern : [ Elision opt BindingRestElement ]
1. If Elision is present, then
   a. Let status be the result of performing IteratorDestructuringAssignmentEvaluation of Elision with iterator as the argument.
   b. ReturnIfAbrupt(status).
2. Return the result of performing IteratorBindingInitialisation for BindingRestElement using iterator and environment as arguments.

ArrayBindingPattern : [ BindingElementList , ]
1. Return the result of performing IteratorBindingInitialisation for BindingElementList using iterator and environment as arguments.

ArrayBindingPattern : [ BindingElementList , Elision ]
1. Let status be the result of performing IteratorBindingInitialisation for BindingElementList using iterator and environment as arguments.
2. ReturnIfAbrupt(status).
3. Return the result of performing IteratorDestructuringAssignmentEvaluation of Elision with iterator as the argument.

ArrayBindingPattern : [ BindingElementList , Elision opt BindingRestElement ]
1. Let status be the result of performing IteratorBindingInitialisation for BindingElementList using iterator and environment as arguments.
2. ReturnIfAbrupt(status).
3. If Elision is present, then
   a. Let status be the result of performing IteratorDestructuringAssignmentEvaluation of Elision with iterator as the argument.
   b. ReturnIfAbrupt(status).
4. Return the result of performing IteratorBindingInitialisation for BindingRestElement using iterator and environment as arguments.

ArrayBindingPattern : [ BindingElementList , BindingElisionElement ]
1. Return the result of performing IteratorBindingInitialisation for BindingElisionElement using iterator and environment as arguments.
BindingElementList : BindingElementList , BindingElisionElement

1. Let status be the result of performing IteratorBindingInitialisation for BindingElementList iterator and environment as arguments.
2. ReturnIfAbrupt(status).
3. Return the result of performing IteratorBindingInitialisation for BindingElement using iterator and environment as arguments.

BindingElisionElement : BindingElement

1. Return the result of performing IteratorBindingInitialisation of BindingElement with iterator as the argument.

BindingElisionElement : Elision BindingElement

1. Let status be the result of performing IteratorDestructuringAssignmentEvaluation of Elision with iterator as the argument.
2. ReturnIfAbrupt(status).
3. Return the result of performing IteratorBindingInitialisation of BindingElement with iterator as the argument.

BindingElement : SingleNameBinding

1. Return the result of performing IteratorBindingInitialisation for SingleNameBinding using iterator and environment as the arguments.

SingleNameBinding : BindingIdentifier Initialiser

1. Let next be the result of IteratorStep(iterator).
2. ReturnIfAbrupt(next).
3. If next is false, then let v be undefined.
4. Else
   a. Let v be IteratorValue(next).
   b. ReturnIfAbrupt(v).
5. If Initialiser is present and v is undefined, then
   a. Let defaultValue be the result of evaluating Initialiser.
   b. Let v be GetValue(defaultValue).
   c. ReturnIfAbrupt(v).
   d. If IsAnonymousFunctionDefinition of Initialiser is true, then
      i. Let hasNameProperty be the result of HasOwnProperty(v, "name").
      ii. ReturnIfAbrupt(hasNameProperty).
      iii. If hasNameProperty is false, then
           1. SetFunctionName(v, StringValue(BindingIdentifier)).
6. Return the result of performing BindingInitialisation for BindingIdentifier passing v and environment as arguments.

BindingElement : BindingPattern Initialiser

1. Let next be the result of IteratorStep(iterator).
2. ReturnIfAbrupt(next).
3. If next is false, then let v be undefined.
4. Else
   a. Let v be IteratorValue(next).
   b. ReturnIfAbrupt(v).
5. If Initialiser is present and v is undefined, then
   a. Let defaultValue be the result of evaluating Initialiser.
b. Let \( v \) be \( \text{GetValue}(\text{defaultValue}) \)
   c. ReturnIfAbrupt(\( v \)).
6. If Type(\( v \)) is not \( \text{Object} \), then throw a \( \text{TypeError} \) exception.
7. Return the result of performing BindingInitialisation of \( \text{nestedAssignmentPattern} \) with \( v \) as the argument.

**BindingRestElement**  
1. Let \( A \) be the result of the abstract operation ArrayCreate with argument 0.
2. Let \( n = 0 \).
3. Repeat,
   a. Let next be the result of IteratorStep(\( \text{iterator} \)).
   b. ReturnIfAbrupt(\( \text{next} \)).
   c. If next is false, then
      i. Return the result of performing BindingInitialisation for BindingIdentifier using \( A \) and environment as arguments.
   d. Let \( \text{nextValue} \) be IteratorValue(\( \text{next} \)).
   e. ReturnIfAbrupt(\( \text{nextValue} \)).
   f. Let defineStatus be the result of CreateDataPropertyOrThrow(\( A \), ToUint32(ToUint32(\( n \))), \( \text{nextValue} \)).
   g. ReturnIfAbrupt(\( \text{defineStatus} \)).
   h. Increment \( n \) by 1.

13.2.3.6 Runtime Semantics: KeyedBindingInitialisation

With parameters \( \text{obj}, \text{environment}, \) and \( \text{propertyName} \).

NOTE When \( \text{undefined} \) is passed for environment it indicates that a \( \text{PutValue} \) 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 preinitialised in order to deal with the possibility of multiple parameters with the same name.

**BindingElement** : BindingPattern Initialiser

1. Let \( v \) be the result of \( \text{Get}(\text{obj}, \text{propertyName}) \).
2. ReturnIfAbrupt(\( v \)).
3. If Initialiser is present and \( v \) is \( \text{undefined} \), then
   a. Let default\( \text{Value} \) be the result of evaluating Initialiser.
   b. Let \( v \) be \( \text{GetValue}(\text{default\text{Value}}) \).
   c. ReturnIfAbrupt(\( v \)).
4. If Type(\( v \)) is not \( \text{Object} \), then throw a \( \text{TypeError} \) exception.
5. Return the result of performing BindingInitialisation for BindingPattern passing \( v \) and environment as arguments.

**SingleNameBinding**  
1. Let \( v \) be the result of \( \text{Get}(\text{obj}, \text{propertyName}) \).
2. ReturnIfAbrupt(\( v \)).
3. If Initialiser is present and \( v \) is \( \text{undefined} \), then
   a. Let default\( \text{Value} \) be the result of evaluating Initialiser.
   b. Let \( v \) be \( \text{GetValue}(\text{default\text{Value}}) \).
   c. ReturnIfAbrupt(\( v \)).
   d. If IsAnonymousFunctionDefinition of Initialiser is true, then
      i. Let has\( \text{NameProperty} \) be the result of HasOwnProperty(\( v \), "\text{name}").
      ii. ReturnIfAbrupt(has\( \text{NameProperty} \)).
      iii. If has\( \text{NameProperty} \) is false, then
         1. Set\( \text{FunctionName}(\text{v}, \text{StringValue}(\text{BindingIdentifier})) \).
   e. Return the result of performing BindingInitialisation for BindingIdentifier passing \( v \) and environment as arguments.
13.3 Empty Statement

Syntax
EmptyStatement : ;

13.3.1 Runtime Semantics: Evaluation

EmptyStatement : ;
1. Return NormalCompletion(empty).

13.4 Expression Statement

Syntax
ExpressionStatement[?Yield] :
  [lookahead  { { function, class, let { } Expression, Yield } ;

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. An ExpressionStatement cannot start with the two token sequence let { because that would make it ambiguous with a let LexicalDeclaration whose first LexicalBinding was an ArrayBindingPattern.

13.4.1 Runtime Semantics: Evaluation

ExpressionStatement : Expression ;
1. Let exprRef be the result of evaluating Expression.
2. Let value be GetValue(exprRef).
3. ReturnIfAbrupt(value).
4. Return NormalCompletion(value).

13.5 The if Statement

Syntax
IfStatement[?Yield, ?Return] :
  if ( Expression ) Statement[?Yield, ?Return]

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.

13.5.1 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.

13.5.2 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. Let stmtValue be the result of evaluating the first Statement.
5. Else,
   a. Let stmtValue be the result of evaluating the second Statement.
6. If stmtValue.[[type]] is normal and stmtValue.[[value]] is empty, then
   a. Return NormalCompletion(undetermined).
7. Return stmtValue.

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, then
   a. Return NormalCompletion(undetermined).
5. Else,
   a. Let stmtValue be the result of evaluating Statement.
6. If stmtValue.[[type]] is normal and stmtValue.[[value]] is empty, then
   a. Return NormalCompletion(undetermined).
7. Return stmtValue.

13.6 Iteration Statements

Syntax

IterationStatement(End, begin) ::= do Statement(End, begin) while (Expression, Begin) opt
while (Expression, Begin) Statement(End, begin)
for (Expression, Begin, Expression, Begin, Expression, Begin) Statement(End, begin)
for (var VariableDeclarationList, Expression, Begin) Statement(End, begin)
for (LexicalDeclarationExpression, Begin, Expression, Begin, Expression, Begin) Statement(End, begin)
for (LeftHandSideExpression in Expression, Begin) Statement(End, begin)
for (var ForBinding in Expression, Begin) Statement(End, begin)
for (ForDeclaration of AssignmentExpression, Begin) Statement(End, begin)
for (ForDeclaration of AssignmentExpression, Begin) Statement(End, begin)

ForDeclaration(End, begin) :
   LetOrConst ForBinding(End, begin)

NOTE 1   ForBinding is defined in 12.1.4.2.

NOTE 2   A semicolon is not required after a do-while statement.
13.6.0 Semantics

13.6.0.1 Runtime Semantics: LoopContinues(completion, labelSet)

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.

13.6.1 The do-while Statement

13.6.1.1 Static Semantics: VarDeclaredNames

See also: 13.0.1, 13.1.8, 13.5.1, 13.6.2.1, 13.6.3.1, 13.6.4.3, 13.10.2, 13.11.4, 13.12.2, 13.14.2, 14.1.11, 14.4.10, 14.5.14, 15.1.0.12, 15.2.5.

IterationStatement: do Statement while (Expression) ; opt
1. Return the VarDeclaredNames of Statement.

13.6.1.2 Runtime Semantics: LabelledEvaluation

With argument labelSet.

See also: 13.0.2, 13.6.2.2, 13.6.3.2, 13.6.4.5, 13.12.3.

IterationStatement: do Statement while (Expression) ; opt
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 not true, then
      i. Assert: exprValue is an abrupt completion.
      ii. If LoopContinues(exprValue,labelSet) is false, return exprValue.

13.6.2 The while Statement

13.6.2.1 Static Semantics: VarDeclaredNames

See also: 13.0.1, 13.1.8, 13.5.1, 13.6.1.1, 13.6.3.1, 13.6.4.3, 13.10.2, 13.11.4, 13.12.2, 13.14.2, 14.1.11, 14.4.10, 14.5.14, 15.1.0.12, 15.2.5.

IterationStatement: while (Expression) Statement
1. Return the VarDeclaredNames of Statement.
13.6.2.2 Runtime Semantics: LabelledEvaluation

With argument labelSet.

See also: 13.0.2, 13.6.1.2, 13.6.3.2, 13.6.4.5, 13.12.3.

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. If exprValue is not true, [kern]
      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.

13.6.3 The for Statement

13.6.3.1 Static Semantics: VarDeclaredNames

See also: 13.0.1, 13.1.8, 13.5.1, 13.6.1.1, 13.6.2.1, 13.6.4.3, 13.10.2, 13.11.4, 13.12.2, 14.1.11, 14.4.10, 14.5.14, 15.1.0.12, 15.2.5.

IterationStatement: for ( Expressionopt ; Expressionopt ; Expressionopt ) Statement

1. Return the VarDeclaredNames of Statement.

IterationStatement: for ( var VariableDeclarationList ; Expressionopt ; Expressionopt ) Statement

1. Let names be BoundNames of VariableDeclarationList.
2. Append to names the elements of the VarDeclaredNames of Statement.
3. Return names.

IterationStatement: for ( LexicalDeclaration Expressionopt ; Expressionopt ) Statement

1. Return the VarDeclaredNames of Statement.

13.6.3.2 Runtime Semantics: LabelledEvaluation

With argument labelSet.

See also: 13.0.2, 13.6.1.2, 13.6.2.2, 13.6.4.5, 13.12.3.

IterationStatement: for ( Expressionopt ; Expressionopt ; Expressionopt ) Statement

1. If the first Expression is present, then
   a. Let exprRef be the result of evaluating the first Expression.
   b. Let exprValue be GetValue(exprRef).
   c. If LoopContinues(exprValue, labelSet) is false, return exprValue.
2. Return the result of performing ForBodyEvaluation 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 VariableDeclarationList ; Expressionopt ; Expressionopt ) Statement

1. Let varDcl be the result of evaluating VariableDeclarationList.
2. If LoopContinues(varDcl, labelSet) is false, return varDcl.
3. Return the result of performing ForBodyEvaluation 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 ( LexicalDeclaration Expressionopt ; Expressionopt ) 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 LexicalDeclaration.
4. For each element dn of the BoundNames of LexicalDeclaration 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.
      ii. Assert: The above call to CreateMutableBinding will never return an abrupt completion.
5. Set the running execution context’s LexicalEnvironment to loopEnv.
6. Let forDcl be the result of evaluating LexicalDeclaration.
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 ForBodyEvaluation 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.

13.6.3.3 Runtime Semantics: ForBodyEvaluation

The abstract operation ForBodyEvaluation 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).
   b. If result is [empty], let V = result.
   c. If LoopContinues(result, labelSet) is false, return testExprValue.
   d. If incrementExpr is not [empty], then
      i. Let incrementExprRef be the result of evaluating incrementExpr.
      ii. Let incrementExprValue be GetValue(incrementExprRef).
      iii. If LoopContinues(incrementExprValue, labelSet) is false, return incrementExprValue.

13.6.4 The for-in and for-of Statements

13.6.4.1 Static Semantics: Early Errors

IterationStatement :
for ( LeftHandSideExpression in Expression ) Statement
for ( LeftHandSideExpression of AssignmentExpression ) Statement
• 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 a `BindingIdentifier` 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 `CoverParenthesisedExpressionAndArrowParameterList`: `(Expression)` and `Expression` derives a production that would produce a Syntax Error according to these rules if that production is substituted for `LeftHandSideExpression`. This rule is recursively applied.

NOTE The last rule means that the other rules are applied even if multiple levels of nested parentheses surround `Expression`.

IterationStatement:
```
for (ForDeclaration in Expression ) Statement
for (ForDeclaration of AssignmentExpression ) Statement
```

• It is a Syntax Error if any element of the `BoundNames` of `ForDeclaration` also occurs in the `VarDeclaredNames` of `Statement`.

13.6.4.2 Static Semantics: `BoundNames` See also: 13.2.1.2, 13.2.2.1, 13.2.3.2, 14.1.2, 14.2.2, 14.4.2, 14.5.2, 15.1.1.2, 0.

`ForDeclaration` : LetOrConst `ForBinding`
1. Return the `BoundNames` of `ForBinding`.

13.6.4.3 Static Semantics: `VarDeclaredNames`

See also: 13.0.1, 13.1.8, 13.5.1, 13.6.1.1, 13.6.2.1, 13.6.3.1, 13.10.2, 13.11.4, 13.12.2, 13.14.2, 14.1.11, 14.4.10, 14.5.14, 15.1.0.12, 15.2.5.

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 AssignmentExpression ) Statement`
1. Return the `VarDeclaredNames` of `Statement`.

IterationStatement: `for ( var ForBinding of AssignmentExpression ) Statement`
1. Return the `VarDeclaredNames` of `Statement`.
1. Let names be the BoundNames of ForBinding.
2. Append to names the elements of the VarDeclaredNames of Statement.
3. Return names

\[\text{IterationStatement: for } \{ \text{ForDeclaration of AssignmentExpression} \} \text{ Statement}\]

1. Return the VarDeclaredNames of Statement.

### 13.6.4.4 Runtime Semantics: BindingInstantiation

With arguments value and environment.

See also: 13.0.2, 13.6.1.2, 13.6.2.2, 13.6.3.2, 13.12.3.

\[\text{ForDeclaration: LetOrConst ForBinding}\]

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 BindingInitialisation for ForBinding passing value and environment as the arguments.

### 13.6.4.5 Runtime Semantics: LabelledEvaluation

With argument labelSet.

See also: 13.0.2, 13.6.1.2, 13.6.2.2, 13.6.3.2, 13.12.3.

\[\text{IterationStatement: for } \{ \text{LeftHandSideExpression in Expression} \} \text{ Statement}\]

1. Let keyResult be the result of performing ForIn/OfExpressionEvaluation with Expression, enumerate, and labelSet.
2. ReturnIfAbrupt(keyResult).
3. Return the result of performing ForIn/OfBodyEvaluation with LeftHandSideExpression, Statement, keyResult, assignment, and labelSet.

\[\text{IterationStatement: for } \{ \text{var ForBinding in Expression} \} \text{ Statement}\]

1. Let keyResult be the result of performing ForIn/OfExpressionEvaluation with Expression, enumerate, and labelSet.
2. ReturnIfAbrupt(keyResult).
3. Return the result of performing ForIn/OfBodyEvaluation with ForBinding, Statement, keyResult, varBinding, and labelSet.

\[\text{IterationStatement: for } \{ \text{ForDeclaration in Expression} \} \text{ Statement}\]

1. Let keyResult be the result of performing ForIn/OfExpressionEvaluation with Expression, enumerate, and labelSet.
2. ReturnIfAbrupt(keyResult).
3. Return the result of performing ForIn/OfBodyEvaluation with ForDeclaration, Statement, keyResult, lexicalBinding, and labelSet.
IterationStatement: `for ( LeftHandSideExpression of AssignmentExpression )` Statement

1. Let `keyResult` be the result of performing ForIn/OfExpressionEvaluation with `AssignmentExpression`, `iterate`, and `labelSet`.
2. ReturnIfAbrupt(`keyResult`).
3. Return the result of performing ForIn/OfBodyEvaluation with `LeftHandSideExpression`, `Statement`, `keyResult`, `assignment`, and `labelSet`.

IterationStatement: `for ( var ForBinding of AssignmentExpression )` Statement

1. Let `keyResult` be the result of performing ForIn/OfExpressionEvaluation with `AssignmentExpression`, `iterate`, and `labelSet`.
2. ReturnIfAbrupt(`keyResult`).
3. Return the result of performing ForIn/OfBodyEvaluation with `ForBinding`, `Statement`, `keyResult`, `varBinding`, and `labelSet`.

IterationStatement: `for ( ForDeclaration of AssignmentExpression )` Statement

1. Let `keyResult` be the result of performing ForIn/OfExpressionEvaluation with `AssignmentExpression`, `iterate`, and `labelSet`.
2. ReturnIfAbrupt(`keyResult`).
3. Return the result of performing ForIn/OfBodyEvaluation with `ForDeclaration`, `Statement`, `keyResult`, `lexicalBinding`, and `labelSet`.

13.6.6 Runtime Semantics: ForIn/OfExpressionEvaluation Abstract Operation

The abstract operation ForIn/OfExpressionEvaluation 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 `exprValue` be GetValue(`exprRef`).
3. If `exprValue` is an abrupt completion, then:
   a. If LoopContinues(`exprValue`, `labelSet`) is `false`, then return `exprValue`.
   b. Else, return Completion(`[[type]]: break`, `[[value]]: empty`, `[[target]]: empty`).
4. If `exprValue` is null or `undefined`, return Completion(`[[type]]: break`, `[[value]]: empty`, `[[target]]: empty`).
5. Let `obj` be ToObject(`exprValue`).
6. If `iterationKind` is `enumerate`, then:
   a. Let `keys` be the result of calling the `[[Enumerate]]` internal method of `obj` with no arguments.
   b. Else, Assert: `iterationKind` is `iterate`.
   c. Let `keys` be GetIterator(`obj`).
7. If `keys` is an abrupt completion, then:
   a. If LoopContinues(`keys`, `labelSet`) is `false`, then return `keys`.
   b. Assert: `keys. [[type]]` is `continue`.
   c. Return Completion(`[[type]]: break`, `[[value]]: empty`, `[[target]]: empty`).
8. Return `keys`.

13.6.7 Runtime Semantics: ForIn/OfBodyEvaluation

The abstract operation ForIn/OfBodyEvaluation 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 `V` = `undefined`.
3. Repeat:
   a. Let `nextResult` be the result of IteratorStep(`keys`).
   b. ReturnIfAbrupt(`nextResult`).
   c. If `nextResult` is `false`, then return NormalCompletion(`V`).

Commented [AWB1550]: NOTE that if this is a for-in, this places the evaluation of the AssignmentExpression outside the scope of the bindings. Perhaps in should be within that scope and subject to TDZ checks on the bond values.

Commented [AWB6668]: Note a continue in the initializer expression is just like a break.

Commented [AWB6669]: Break/continue in the expression works normally (future for do {} or block lambda expressions).

Commented [AWB6670]: Note a continue in the initializer expression is just like a break.

Commented [AWB6671]: Completion value reform.
d. Let nextValue be the result of IteratorValue(nextResult).

f. If lhsKind is assignment, then
   i. Assert: lhs is a LeftHandSideExpression.
   ii. If lhs is not an ObjectLiteral or an ArrayLiteral then
        1. Let lhsRef be the result of evaluating lhs (it may be evaluated repeatedly).
        2. Let status be the result of performing PutValue(lhsRef, nextValue).
   iii. Else
        1. Let assignmentPattern be the parse of the source code corresponding to lhs using AssignmentPattern as the goal symbol.
        2. If Type(nextValue) is not Object, then throw a TypeError exception.
        3. Let status be the result of performing DestructuringAssignmentEvaluation of AssignmentPattern using nextValue as the argument.

g. Else if lhsKind is VarBinding, then
   i. Assert: lhs is a ForBinding.
   ii. Let status be the result of performing BindingInitialisation for lhs passing nextValue and undefined as the arguments.

h. Else,
   i. Assert: lhsKind is lexicalBinding.
   ii. Assert: lhs is a ForDeclaration.
   iii. Let iterationEnv be the result of calling NewDeclarativeEnvironment passing oldEnv as the argument.
   iv. Perform BindingInstantiation for lhs passing nextValue and iterationEnv as arguments.
   v. Let status be NormalCompletion(empty)
   vi. Set the running execution context’s LexicalEnvironment to iterationEnv.
   i. If status.[[type]] is normal, then
      i. Let status be the result of evaluating stmt.
      ii. If status.[[type]] is normal and status.[[value]] is not empty, then
         1. Let V = status.[[value]].

e. If status is an abrupt completion and LoopContinues(status,labelSet) is false, then return status.

13.7 The continue Statement

Syntax
ContinueStatement : continue ;
   continue [no LineTerminator here] IdentifierReference ;

13.7.1 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 IdentifierReference ;
   • It is a Syntax Error if StringValue(UnresolvedIdentifier) does not appear in the CurrentLabelSet of an enclosing (but not crossing function boundaries) IterationStatement.

13.7.2 Runtime Semantics: Evaluation
ContinueStatement : continue ;
1. Return Completion{[[type]]=[continue], [[value]]=empty, [[target]]=empty}.

ContinueStatement : continue UnresolvedIdentifier ;

1. Let label be the StringValue of UnresolvedIdentifier.
2. Return Completion{[[type]]=[continue], [[value]]=empty, [[target]]=label}.

13.8 The break Statement

Syntax
BreakStatement;uyo : break ;
break [no LineTerminator here] UnresolvedIdentifier ;

13.8.1 Static Semantics: Early Errors

BreakStatement : break ;

• It is a Syntax Error if this production is not nested, directly or indirectly (but not crossing function boundaries), within an IterationStatement or a SwitchStatement.

BreakStatement : break UnresolvedIdentifier ;

• It is a Syntax Error if StringValue(UnresolvedIdentifier) does not appear in the CurrentLabelSet of an enclosing (but not crossing function boundaries) Statement.

13.8.2 Runtime Semantics: Evaluation

BreakStatement : break ;

1. Return Completion{[[type]]=[break], [[value]]=empty, [[target]]=empty}.

BreakStatement : break UnresolvedIdentifier ;

1. Let label be the StringValue of UnresolvedIdentifier.
2. Return Completion{[[type]]=[break], [[value]]=empty, [[target]]=label}.

13.9 The return Statement

Syntax
ReturnStatement;uyo : 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.

13.9.1 Runtime Semantics: Evaluation

ReturnStatement : return ;

1. Return Completion{[[type]]=[return], [[value]]=undefined, [[target]]=empty}.

ReturnStatement : return 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}.

13.10 The with Statement

Syntax

WithStatement[Yield, Return]:
  with ( Expression[In, ?] ) Statement[Yield, Return]

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.

13.10.1 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.

13.10.2 Static Semantics: VarDeclaredNames

See also: 13.0.1, 13.1.8, 13.5.1, 13.6.1.1, 13.6.2.1, 13.6.3.1, 13.6.4.3, 13.11.4, 13.12.2, 13.14.2, 14.1.11, 14.4.10, 14.5.14, 15.1.0.12, 15.2.5.

WithStatement : with ( Expression ) Statement
1. Return the VarDeclaredNames of Statement.

13.10.3 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’s environment record 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.

13.11 The switch Statement

Syntax

SwitchStatement[Yield, Return]:
  switch ( Expression[In] ) CaseBlock[Yield, Return]
CaseBlock : ( CaseClauses? )
  { CaseClauses opt, DefaultClause? }

CaseClauses : CaseClause CaseClauses opt

CaseClause : case Expression Yield : StatementList? ;

DefaultClause : default : StatementList?

13.11.1 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.

13.11.2 Static Semantics: LexicalDeclarations

See also: 13.1.2, 15.1.0.10.

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

1. If the StatementList is present, return the LexicalDeclarations of StatementList.
2. Else return a new empty List.

DefaultClause : default : StatementList

1. If the StatementList is present, return the LexicalDeclarations of StatementList.
2. Else return a new empty List.

13.11.3 Static Semantics: LexicallyDeclaredNames

See also: 13.1.3, 14.1.10, 14.2.7, 14.4.7, 14.5.8, 15.1.0.7, 15.2.3.
CaseBlock : { }

1. Return a new empty List.

CaseBlock : { CaseClausesopt DefaultClause CaseClausesopt }

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 : StatementListopt

1. If the StatementList is present, return the VarDeclaredNames of StatementList.
2. Else return a new empty List.

DefaultClause : default : StatementListopt

1. If the StatementList is present, return the VarDeclaredNames of StatementList.
2. Else return a new empty List.

13.11.4 Static Semantics: VarDeclaredNames

See also: 13.0.1, 13.1.8, 13.5.1, 13.6.1.1, 13.6.2.1, 13.6.3.1, 13.6.4.3, 13.10.2, 13.12.2, 13.14.2, 14.1.11, 14.4.10, 14.5.14, 15.1.0.12, 15.2.5.

SwitchStatement : switch (Expression) CaseBlock

1. Return the VarDeclaredNames of CaseBlock.

CaseBlock : { }

1. Return a new empty List.

CaseBlock : { CaseClausesopt DefaultClause CaseClausesopt }

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

1. If the StatementList is present, return the VarDeclaredNames of StatementList.
2. Else return a new empty List.

DefaultClause : default : StatementList

1. If the StatementList is present, return the VarDeclaredNames of StatementList.
2. Else return a new empty List.

13.11.5 Runtime Semantics: CaseBlock Evaluation

With argument input.

CaseBlock : { CaseClauses }

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 CaseSelectorEvaluation of C.
   c. ReturnIfAbrupt(clauseSelector).
   d. Let matched be the result of performing Strict Equality Comparison input === clauseSelector.
   e. If matched is true, then
       i. Set searching to false.
       ii. If C has a StatementList, then
           1. Let V be the result of evaluating C's StatementList.
           2. ReturnIfAbrupt(V).
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. Let R be the result of evaluating C's StatementList.
      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 DefaultClause CaseClauses }

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 CaseSelectorEvaluation 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. Let found be the result of performing Strict Equality Comparison input === clauseSelector.
   b. If found is true, then
      i. Let R be the result of evaluating CaseClause C.
      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

© Ecma International 2009 – All rights reserved

209
a. Let \( B \) be a new List containing 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 CaseSelectorEvaluation of \( C \).
         a. If clauseSelector.[[value]] is empty, return Completion{[[type]]: 
            clauseSelector.[[type]], [[value]]: undefined, [[target]]: 
            clauseSelector.[[target]]}
         b. Else, return clauseSelector.
      2. Let foundInB be the result of performing Strict Equality Comparison input === 
         clauseSelector.
   ii. If foundInB is true, then
      Let \( R \) be the result of evaluating CaseClause \( C \).
      If \( R.[[value]] \) is not empty, then let \( V = R.[[value]] \).
      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. Let \( R \) be the result of evaluating DefaultClause.
   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 containing 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)
      a. Let \( R \) be the result of evaluating CaseClause \( C \).
      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).

13.11.6 Runtime Semantics: CaseSelectorEvaluation

CaseClause : case Expression : StatementListopt

1. Let exprRef be the result of evaluating Expression.
2. Return GetValue(exprRef).

NOTE CaseSelectorEvaluation 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.

13.11.7 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 CaseBlockEvaluation 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 :
CaseClause: case Expression : StatementList
1. Return the result of evaluating StatementList.

DefaultClause: default:
1. Return NormalCompletion(empty).

DefaultClause: default: StatementList
1. Return the result of evaluating StatementList.

13.12 Labelled Statements

Syntax
LabelledStatement[Yield][Return]:
  UnresolvedIdentifier : Statement[Yield][Return]

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.

13.12.1 Static Semantics: Early Errors
• It is a Syntax Error if a LabelledStatement is enclosed by a LabelledStatement with the same IdentifierReference 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.

13.12.2 Static Semantics: VarDeclaredNames

LabelledStatement:
  IdentifierReference : Statement
  yield : Statement

1. Return the VarDeclaredNames of Statement.

13.12.3 Runtime Semantics: LabelledEvaluation

With argument labelSet.
See also: 13.0.2, 13.6.1, 13.6.2, 13.6.3, 13.6.4.

LabelledStatement: IdentifierReference : Statement
1. Let label be the StringValue of IdentifierReference.
2. Return the result of LabelledStatementEvaluation(label, Statement, labelSet).

LabelledStatement: yield : Statement

© Ecma International 2009 – All rights reserved
13.12.3.1 Runtime Semantics: LabelledStatementEvaluation(label, stmt, labelSet)

The abstract operation LabelledStatementEvaluation with arguments label, stmt, and labelSet is performed as follows:

1. Let newLabelSet be a new List containing label and the elements of labelSet.
2. If stmt is either LabelledStatement or BreakableStatement, then
   a. Let stmtResult be the result of performing LabelledEvaluation of stmt with argument newLabelSet.
3. Else, a. Let stmtResult be the result of evaluating stmt.
4. If stmtResult.[[type]] is break and SameValue(stmtResult.[[target]], label), then
   a. Let result be NormalCompletion(stmtResult.[[value]])
5. Else, a. Let result be stmtResult.
6. Return result.

13.12.4 Runtime Semantics: Evaluation

LabelledStatement:
  IdentifierReference : Statement
  yield : Statement
1. Let newLabelSet be a new empty List.
2. Return the result of performing LabelledEvaluation of this LabelledStatement with argument newLabelSet.

13.13 The throw Statement

Syntax

13.13.1 Runtime Semantics: Evaluation

The production ThrowStatement : throw 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]]: exprValue, [[target]]: empty}).

13.14 The try Statement

Syntax
TryStatement : try Block [+ Yield, ? Return]
try Block [Yield Finally Yield, ? Return]
try Block [Yield Catch Try Yield, ? Return]

CatchTry : catch Parameter Yield Block Try Yield

FinallyTry : finally Block Try Yield


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 `CatchParameter` is bound to that exception.

### 13.14.1 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`.

### 13.14.2 Static Semantics: VarDeclaredNames

See also: 13.0.1, 13.1.8, 13.5.1, 13.6.1.1, 13.6.2.1, 13.6.3.1, 13.6.4.3, 13.10.2, 13.11.4, 13.12.2, 14.1.11, 14.4.10, 14.5.14, 15.1.0.12, 15.2.5.

**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`.

### 13.14.3 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 preinitialised prior to evaluation of its initialiser.

See also: 12.1.4.2.2, 13.2.1.5, 13.2.2.2, 13.2.3.4.

**CatchParameter**: `BindingPattern`

1. If `Type(value)` is not `Object`, then throw a `TypeError` exception.
2. Return the result of performing BindingInitialisation for BindingPattern passing value and environment as the arguments.

13.14.4 Runtime Semantics: CatchClauseEvaluation

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.
   b. Assert: The above call to CreateMutableBinding will never return an abrupt completion.
4. Let status be the result of performing BindingInitialisation 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.

13.14.5 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 CatchClauseEvaluation 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 CatchClauseEvaluation 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.

13.15 The debugger statement

Syntax

DebuggerStatement :
  debugger ;
13.15.1 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 \texttt{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(\texttt{empty}).
3. Return result.

14 ECMAScript Language: Functions and Classes

NOTE Various ECMAScript language elements cause the creation of ECMAScript function objects (9.2). Evaluation of such functions starts with the execution of their \texttt{[[Call]]} internal method (9.2.1).

14.1 Function Definitions

Syntax

FunctionDeclaration[\texttt{yield}] ::=
\texttt{function BindingIdentifier[\texttt{yield}] ( FormalParameters ) \{ FunctionBody \}}

FunctionExpression ::=
\texttt{function BindingIdentifieropt ( FormalParameters ) \{ FunctionBody \}}

StrictFormalParameters[\texttt{yield}, GeneratorParameter][\texttt{yield}] ::=
FormalParameter[\texttt{yield}, GeneratorParameter]

FormalParameter[\texttt{yield}, GeneratorParameter][\texttt{yield}] ::=
FormalParameterList[\texttt{yield}, GeneratorParameter]

FormalParameterList[\texttt{yield}, GeneratorParameter][\texttt{yield}] ::=
[empty]

FormalParameterList[\texttt{yield}, GeneratorParameter][\texttt{yield}] ::=
FunctionRestParameter[\texttt{yield}]

FunctionRestParameter[\texttt{yield}][\texttt{yield}] ::=
FormalParameterList[\texttt{yield}, GeneratorParameter]

FunctionRestParameter[\texttt{yield}][\texttt{yield}] ::=
FormalParameterList[\texttt{yield}, GeneratorParameter], FormalParameterList[\texttt{yield}, GeneratorParameter]

FunctionRestParameter[\texttt{yield}][\texttt{yield}] ::=
FormalParameterList[\texttt{yield}, GeneratorParameter], FormalParameterList[\texttt{yield}, GeneratorParameter], FormalParameterList[\texttt{yield}, GeneratorParameter]

FunctionRestParameter[\texttt{yield}][\texttt{yield}] ::=
BindingRestElement[\texttt{yield}]

BindingRestElement[\texttt{yield}] ::=

FormalParameter[\texttt{yield}, GeneratorParameter][\texttt{yield}] ::=
BindingElement[\texttt{yield}, GeneratorParameter]

FormalParameterList[\texttt{yield}, GeneratorParameter][\texttt{yield}] ::=
FunctionStatementList[\texttt{yield}]

FunctionStatementList[\texttt{yield}] ::=

StatementList[\texttt{yield}, Return]
Supplemental Syntax

The following productions are used as an aid in specifying the semantics of certain ECMAScript language features. They are not used when parsing ECMAScript source code.

**FunctionBody:**

```
ThrowTypeError
```

**ThrowTypeError:**

```
[empty]
```

### 14.1.1 Static Semantics: Early Errors

**FunctionDeclaration:** `function BindingIdentifier ( FormalParameters ) { FunctionBody }`

and

**FunctionExpression:** `function BindingIdentifier opt ( FormalParameters ) { FunctionBody }`

- If the source code matching this production is strict code, the Early Error rules for `StrictFormalParameters : FormalParameters` are applied.

- It is a Syntax Error if `IsSimpleParameterList of FormalParameters` is `false` and any element of the `BoundNames of FormalParameters` also occurs in the `VarDeclaredNames of FunctionBody`.

- It is a Syntax Error if any element of the `BoundNames of FormalParameters` also occurs in the `LexicallyDeclaredNames of FunctionBody`.

**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`.

**StrictFormalParameters : FormalParameters**

- It is a Syntax Error if `BoundNames of FormalParameters` contains any duplicate elements.

**FormalParameters : FormalParameterList**

- 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".

**NOTE** Multiple occurrences of the same `Identifier` in a `FormalParameterList` is only allowed for non-strict functions and generator functions that have simple parameter lists.

**FunctionStatementList : 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`.

### 14.1.2 Static Semantics: BoundNames

See also: 13.2.1.2, 13.2.2.1, 13.2.3.2, 13.6.4.2, 14.2.2, 14.4.2, 14.5.2.15.1.1.2, 0.

**FunctionDeclaration : function BindingIdentifier ( FormalParameters ) { FunctionBody }**

1. Return the `BoundNames of BindingIdentifier`.  

© Ecma International 2013
FormalParameters : [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.

14.1.3 Static Semantics: Contains

With parameter symbol.

See also: 5.3, 12.1.5.2, 12.2.1.1, 14.2.3, 14.4.3, 14.5.4

FunctionDeclaration : function BindingIdentifier ( FormalParameters ) { FunctionBody }
1. Return false.

FunctionExpression : function BindingIdentifier ( FormalParameters ) { FunctionBody }
1. Return false.

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

14.1.4 Static Semantics: ExpectedArgumentCount

See also: 14.2.5, 14.3.2.

FormalParameters : [empty]
1. Return 0.

FormalParameterList : 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 Initialiser is allowed after the first parameter with an Initialiser 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.
1. Let \( \text{count} \) be the \text{ExpectedArgumentCount} of \text{FormalsList}.
2. If \text{HasInitialiser} of \text{FormalsList} is \text{true} or \text{HasInitialiser} of \text{FormalParameter} is \text{true}, then return \text{count}.
3. Return \text{count}+1.

### 14.1.5 Static Semantics: \text{HasInitialiser}

See also: 13.2.3.3.

**FormalsList**: \text{FormalsList}, \text{FormalParameter}

1. If \text{HasInitialiser} of \text{FormalsList} is \text{true}, then return \text{true}.
2. Return \text{HasInitialiser} of \text{FormalParameter}.

### 14.1.6 Static Semantics: \text{IsAnonymousFunctionDefinition}

See also: 12.1.0.2, 12.1.10.2, 12.2.1.2, 12.3.2, 12.4.2, 12.5.1, 12.6.1, 12.7.1, 12.8.1, 12.9.2, 12.10.2, 12.11.1, 12.12.1, 12.13.2, 12.14.2, 14.4.5, 14.5.5.

**FunctionExpression**: \text{function} (\text{FormalParameters}) \{\text{FunctionBody}\}

1. Return \text{true}.

**FunctionExpression**: \text{function} \text{BindingIdentifier} (\text{FormalParameters}) \{\text{FunctionBody}\}

1. Return \text{false}.

### 14.1.7 Static Semantics: \text{IsConstantDeclaration}

See also: 13.2.1.3, 14.4.4, 14.5.5.

**FunctionDeclaration**: \text{function} \text{BindingIdentifier} (\text{FormalParameters}) \{\text{FunctionBody}\}

1. Return \text{false}.

### 14.1.8 Static Semantics: \text{IsSimpleParameterList}

**FormalParameters**: \[\text{empty}\]

1. Return \text{true}.

**FormalParameterList**: \text{FunctionRestParameter}

1. Return \text{false}.

**FormalParameterList**: \text{FormalList}, \text{FunctionRestParameter}

1. Return \text{false}.

**FormalList**: \text{FormalList}, \text{FormalParameter}

1. If \text{IsSimpleParameterList} of \text{FormalList} is \text{false}, return \text{false}.
2. Return \text{IsSimpleParameterList} of \text{FormalParameter}.

**FormalParameter**: \text{BindingElement}
1. If HasInitialiser of BindingElement is true, return false.
2. If FormalParameter Contains BindingPattern is true, return false.
3. Return true.

14.1.9 Static Semantics: IsStrict

See also: 15.1.0.2, 15.2.2.

FunctionStatementList : StatementListopt
1. If this FunctionStatementList is contained in strict code or if StatementList is strict code, then return true. Otherwise, return false.

14.1.10 Static Semantics: LexicallyDeclaredNames

See also: 13.1.3, 13.11.3, 14.2.7, 14.4.7, 14.5.8, 15.1.0.7, 15.2.3.

FunctionDeclaration : function BindingIdentifier ( FormalParameters ) { FunctionBody }
1. Return the BoundNames of BindingIdentifier.

FunctionStatementList : [empty]
1. Return an empty List.

FunctionStatementList : StatementList
1. Return TopLevelLexicallyDeclaredNames of StatementList.

14.1.11 Static Semantics: VarDeclaredNames

See also: 13.0.1, 13.1.8, 13.5.1, 13.6.1.1, 13.6.2.1, 13.6.3.1, 13.6.4.3, 13.10.2, 13.11.4, 13.12.2, 13.14.2, 14.4.10, 14.5.14, 15.1.0.12, 15.2.5.

FunctionDeclaration : function BindingIdentifier ( FormalParameters ) { FunctionBody }
1. Return an empty List.

FunctionStatementList : [empty]
1. Return an empty List.

FunctionStatementList : StatementList
1. Return TopLevelVarDeclaredNames of StatementList.

14.1.12 Runtime Semantics: EvaluateBody

With parameter functionObject.

See also: 14.2.9, 14.4.11.

FunctionBody : FunctionStatementList
1. The code of this FunctionBody is strict mode code if it is contained in strict mode code or if the Directive Prologue (15.3) of its FunctionStatementList contains a Use Strict Directive or if any of the conditions in 10.2.1 apply. If the code of this FunctionBody is strict mode code, FunctionStatementList is evaluated in the following steps as strict mode code. Otherwise, StatementList is evaluated in the following steps as non-strict mode code.
2. Let result be the result of evaluating FunctionStatementList.
3. If `result[[type]]` is `return` then return `NormalCompletion(result[[value]])`.
4. ReturnIfAbrupt(result).
5. Return NormalCompletion(undefined).

FunctionBody: `ThrowTypeError`

1. Throw a `TypeError` exception.

14.1.13 Runtime Semantics: IteratorBindingInitialisation

With parameters `iterator` 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 preinitialised in order to deal with the possibility of multiple parameters with the same name.

See also: 13.2.3.5.

FormalParameters: `[empty]`

1. Return `NormalCompletion(empty)`.

FormalParameterList: FormalList, `FunctionRestParameter`

1. Let `restIndex` be the result of performing `IteratorBindingInitialisation` for `FormalsList` using `iterator`, and `environment` as the arguments.
2. ReturnIfAbrupt(restIndex).
3. Return the result of performing `IteratorBindingInitialisation` for `FunctionRestParameter` using `iterator` and `environment` as the arguments.

FormList: FormalList, FormalParameter

1. Let `status` be the result of performing `IteratorBindingInitialisation` for `FormalsList` using `iterator` and `environment` as the arguments.
2. ReturnIfAbrupt(status).
3. Return the result of performing `IteratorBindingInitialisation` for `FormalParameter` using `iterator` and `environment` as the arguments.

14.1.14 Runtime Semantics: InstantiateFunctionObject

With parameter `scope`.

See also: 14.4.12.

FunctionDeclaration: `function` BindingIdentifier `{ FormalParameters } { 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 `name` be `StringValue` of `BindingIdentifier`.
3. Let `F` be the result of performing the `FunctionCreate` abstract operation with arguments `Normal`, `FormalParameters`, `FunctionBody`, `scope`, `strict`.
4. Perform the abstract operation `MakeConstructor` with argument `F`.
5. Set `FunctionName(F, name)`.

© Ecma International 2013
14.1.15 Runtime Semantics: Evaluation

FunctionDeclaration : function BindingIdentifier ( FormalParameters ) { FunctionBody }

1. Return NormalCompletion( empty )

FunctionExpression : function ( FormalParameters ) { 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, FormalParameters, FunctionBody, scope, and strict.
4. Perform the abstract operation MakeConstructor with argument closure.
5. Return closure.

FunctionExpression : function BindingIdentifier ( FormalParameters ) { 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 name be StringValue of BindingIdentifier.
4. Call the CreateImmutableBinding concrete method of funcEnv passing name as the argument.
5. Let closure be the result of performing the FunctionCreate abstract operation with arguments Normal, FormalParameters, FunctionBody, funcEnv, strict.
6. Perform the abstract operation MakeConstructor with argument closure.
7. SetFunctionName( closure, name ).
8. Call the InitialiseBinding concrete method of envRec passing name and closure as the arguments.
9. Call the InitialiseBinding concrete method of envRec passing name and closure as the arguments.
10. 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.

FunctionStatementList : [ empty ]

1. Return NormalCompletion( undefined ).

14.2 Arrow Function Definitions

Syntax

ArrowFunctionO : ArrowParameters => ConciseBodyO?

ArrowParameters : BindingIdentifier

ConciseBodyO : [ ]| AssignmentExpression

FunctionBody : [ ]| AssignmentExpression

© Ecma International 2009 – All rights reserved
Supplemental Syntax

When processing the production ArrowParameters : CoverParenthesisedExpressionAndArrowParameterList the following grammar is used to refine the interpretation of CoverParenthesisedExpressionAndArrowParameterList.

ArrowFormalParameters :
  ( StrictFormalParameters )

14.2.1 Static Semantics: Early Errors

ArrowFunction : ArrowParameters => ConciseBody

- It is a Syntax Error if any element of the BoundNames of ArrowParameters also occurs in the VarDeclaredNames of ConciseBody.
- It is a Syntax Error if any element of the BoundNames of ArrowParameters also occurs in the LexicallyDeclaredNames of ConciseBody.

ArrowParameters : CoverParenthesisedExpressionAndArrowParameterList

- It is a Syntax Error if the lexical token sequence matched by CoverParenthesisedExpressionAndArrowParameterList cannot be parsed with no tokens left over using ArrowFormalParameters as the goal symbol.
- It is a Syntax Error if any early errors are present for CoveredFormalsList of CoverParenthesisedExpressionAndArrowParameterList.

14.2.2 Static Semantics: BoundNames

See also: 13.2.1.2, 13.2.2.1, 13.2.3.2, 13.6.4.2, 14.1.2, 14.4.2, 14.5.2, 15.1.1.2, 0.

ArrowParameters : CoverParenthesisedExpressionAndArrowParameterList

1. Let formals be CoveredFormalsList of CoverParenthesisedExpressionAndArrowParameterList.
2. Return the BoundNames of formals.

14.2.3 Static Semantics: Contains

With parameter symbol.

See also: 5.3, 12.1.5.2, 12.2.1.1, 14.1.3, 14.4.3, 14.5.4

ArrowFunction : ArrowParameters => ConciseBody

1. If symbol is neither super nor this, then return false.
2. If ArrowParameters Contains symbol is true, return true;
3. Return ConciseBody Contains symbol.

NOTE Normally, Contains does not look inside most function forms However, Contains is used to detect this and super usage within an ArrowFunction.

ArrowParameters : CoverParenthesisedExpressionAndArrowParameterList

1. Let formals be CoveredFormalsList of CoverParenthesisedExpressionAndArrowParameterList.
2. Return formals Contains symbol.
14.2.4 Static Semantics: CoveredFormalsList

ArrowParameters : BindingIdentifier
1. Return BindingIdentifier.

CoverParenthesisedExpressionAndArrowParameterList :
( Expression )
( )
( ... IdentifierName )
( Expression , ... IdentifierName )
1. Return the result of parsing the lexical token stream matched by
CoverParenthesisedExpressionAndArrowParameterList using ArrowFormalParameters as the goal symbol.

14.2.5 Static Semantics: ExpectedArgumentCount

See also: 14.1.4, 14.3.2.

ArrowParameters : BindingIdentifier
1. Return 1.

ArrowParameters : CoverParenthesisedExpressionAndArrowParameterList
1. Let formals be CoveredFormalsList of CoverParenthesisedExpressionAndArrowParameterList.
2. Return the ExpectedArgumentCount of formals.

14.2.6 Static Semantics: IsSimpleParameterList

ArrowParameters : BindingIdentifier
1. Return true.

ArrowParameters : CoverParenthesisedExpressionAndArrowParameterList
1. Let formals be CoveredFormalsList of CoverParenthesisedExpressionAndArrowParameterList.
2. Return the IsSimpleParameterList of formals.

14.2.7 Static Semantics: LexicallyDeclaredNames

See also: 13.1.3, 13.11.3, 14.1.10, 14.4.7, 14.5.8, 15.1.0.7, 15.2.3.

ConciseBody : AssignmentExpression
1. Return an empty List.

14.2.8 Runtime Semantics: IteratorBindingInitialisation

With parameters iterator and environment.

See also: 12.1.4.2.2, 13.2.1.5, 13.2.2.2, 13.2.3.4, 13.14.3, 14.1.13.

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 preinitialised in order to deal with the possibility of multiple parameters with the same name.

ArrowParameters : BindingIdentifier
1. Let next be the result of IteratorStep(iterator).
2. ReturnIfAbrupt(next).
3. If next is false, then let v be undefined
4. Else
   a. Let v be IteratorValue(next).
   b. ReturnIfAbrupt(v).
5. Return the result of performing BindingInitialisation for BindingIdentifier using v and environment as the arguments.

ArrowParameters : CoverParenthesisedExpressionAndArrowParameterList

1. Let forms be CoveredFormalsList of CoverParenthesisedExpressionAndArrowParameterList.
2. Return the result of performing IteratorBindingInitialisation of forms with arguments value and environment.

14.2.9 Runtime Semantics: EvaluateBody

With parameter functionObject.

See also: 14.1.12, 14.4.11.

ConciseBody : 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 10.2.1 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. If exprValue.[[type]] is return then return NormalCompletion(exprValue.[[value]]).
5. ReturnIfAbrupt(exprValue).
6. Return NormalCompletion(exprValue).

NOTE Even though the absence of extensions to this specification, the test is step 4 will never be true.

14.2.10 Runtime Semantics: Evaluation

ArrowFunction : ArrowParameters => ConciseBody

1. If the code of this ArrowFunction is contained in strict mode code or if any of the conditions in 10.2.1 apply, then let strict be true. Otherwise let strict be false.
2. Let scope be the LexicalEnvironment of the running execution context.
3. Let parameters be CoveredFormalsList of ArrowParameters.
4. Let closure be the result of performing the FunctionCreate abstract operation with arguments Arrow, parameters, ConciseBody, scope, and strict.
5. Return closure.

NOTE Even though an ArrowFunction may contain references to super, the FunctionCreate call in step 4 is not passed the optional homeObject and methodName parameters. An ArrowFunction that references super is always contained within a non-ArrowFunction and the necessary state to implement super is accessible via the scope that is captured by the function object of the ArrowFunction.
14.3 Method Definitions

Syntax
MethodDefinition :=
  PropertyName ( StrictFormalParameters ) { FunctionBody }
  GeneratorMethod
get PropertyName ( ) { FunctionBody }
set PropertyName ( PropertySetParameterList ) { FunctionBody }

PropertySetParameterList :
  BindingIdentifier
  BindingPattern

NOTE: The single element of a PropertySetParameterList may not have a default value initializer because set accessor are always called with an implicitly provided argument.

14.3.1 Static Semantics: Early Errors

MethodDefinition := PropertyName ( StrictFormalParameters ) { FunctionBody }

• It is a Syntax Error if any element of the BoundNames of StrictFormalParameters also occurs in the VarDeclaredNames of FunctionBody.
• It is a Syntax Error if any element of the BoundNames of StrictFormalParameters also occurs in the LexicallyDeclaredNames of FunctionBody.

MethodDefinition := set PropertyName ( PropertySetParameterList ) { FunctionBody }

• It is a Syntax Error if IsSimpleParameterList of PropertySetParameterList is false 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 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.

14.3.2 Static Semantics: ExpectedArgumentCount

See also: 14.1.4, 14.2.5.

PropertySetParameterList := BindingIdentifier
1. Return 1.

PropertySetParameterList := BindingPattern
1. Return 1.

14.3.3 Static Semantics: HasComputedPropertyKey

See also: 12.1.5.3, 14.4.4

MethodDefinition :
  PropertyName ( StrictFormalParameters ) { FunctionBody }
  get PropertyName ( ) { FunctionBody }
  set PropertyName ( PropertySetParameterList ) { FunctionBody }
1. Return HasComputedPropertyKey of PropertyName.
14.3.4 Static Semantics: `IsSimpleParameterList`

PropertySetParameterList : BindingIdentifier

1. Return `true`.

PropertySetParameterList : BindingPattern

1. Return `false`.

14.3.5 Static Semantics: `PropName`

See also: 12.1.5.5, 14.4.8, 14.5.11

MethodDefinition :
  PropertyName ( StrictFormalParameters ) { FunctionBody }
  set PropertyName ( PropertySetParameterList ) { FunctionBody }

1. Return PropName of PropertyName.

14.3.6 Static Semantics: ReferencesSuper

See also: 14.4.9.

MethodDefinition : PropertyName ( StrictFormalParameters ) { FunctionBody }

1. If StrictFormalParameters Contains `super` is `true`, then return `true`.
2. Return FunctionBody Contains `super`.

MethodDefinition : get PropertyName ( ) { FunctionBody }

1. Return FunctionBody Contains `super`.

MethodDefinition : set PropertyName ( PropertySetParameterList ) { FunctionBody }

1. If PropertySetParameterList Contains `super` is `true`, then return `true`.
2. Return FunctionBody Contains `super`.

14.3.7 Static Semantics: `SpecialMethod`

MethodDefinition : PropertyName ( StrictFormalParameters ) { FunctionBody }

1. Return `false`.

MethodDefinition :
  GeneratorMethod
  get PropertyName ( ) { FunctionBody }
  set PropertyName ( PropertySetParameterList ) { FunctionBody }

1. Return `true`.

14.3.8 Runtime Semantics: `DefineMethod`

With parameters `object` and optional parameter `functionPrototype`. 

© Ecma International 2013
MethodDefinition : PropertyName (StrictFormalParameters ) { FunctionBody }

1. Let propName be the result of evaluating PropertyName.
2. Return IfAbrupt(propName).
3. Let strict be IsStrict of FunctionBody.
4. Let scope be the running execution context’s LexicalEnvironment.
5. If ReferencesSuper of MethodDefinition is true, then
   a. Let closure be the result of performing the FunctionCreate abstract operation with arguments Method,
      StrictFormalParameters, FunctionBody, scope, and strict and with propName as the methodName optional
      argument and object as the homeObject optional argument. If functionPrototype was passed as a parameter
      then also pass its value as the functionPrototype optional argument of FunctionCreate.
6. Else
   a. Let closure be the result of performing the FunctionCreate abstract operation with arguments Method,
      StrictFormalParameters, FunctionBody, scope, and strict. If functionPrototype was passed as a parameter
      then also pass its value as the functionPrototype optional argument of FunctionCreate.
7. Return the Record([key]: propName, ([closure]: closure)).

14.3.9 Runtime Semantics: PropertyDefinitionEvaluation

With parameter object.

See also: 12.1.5.8, 14.4.13, B.3.1

MethodDefinition : PropertyName (StrictFormalParameters ) { FunctionBody }

1. Let methodDef be the result of DefineMethod of this MethodDefinition with argument object.
2. Return IfAbrupt(methodDef).
3. SetFunctionName(methodDef[[closure]], methodDef[[key]]).
4. Let desc be the Property Descriptor{[[Value]]: methodName, [[Configurable]]: true, [[Enumerable]]: true.
   [[Writable]]: true, [[Configurable]]: true}. 
5. Return the result of DefinePropertyOrThrow(object, methodDef[[key]], desc).

MethodDefinition : GeneratorMethod

See 14.4.

MethodDefinition : get PropertyName ( ) { FunctionBody }

1. Let propName be the result of evaluating PropertyName.
2. Return IfAbrupt(propName).
3. Let strict be IsStrict of FunctionBody.
4. Let scope be the running execution context’s LexicalEnvironment.
5. Let formalParameterList be the production FormalParameters : (empty)
6. If ReferencesSuper of MethodDefinition is true, then
   a. Let closure be the result of performing the FunctionCreate abstract operation with arguments Method,
      formalParameterList, FunctionBody, scope, and strict and with propName as the methodName optional
      argument and object as the homeObject 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. SetFunctionName(closure, propName, "get").
9. Let desc be the Property Descriptor{[[Get]]: closure, [[Enumerable]]: true, [[Configurable]]: true}
10. Return the result of DefinePropertyOrThrow(object, propName, desc).

MethodDefinition : set PropertyName ( PropertySetParameterList ) { FunctionBody }

1. Let propName be the result of evaluating PropertyName.
2. Return IfAbrupt(propName).
3. Let strict be IsStrict of FunctionBody.
4. Let `scope` be the running execution context’s LexicalEnvironment.
5. If `ReferencesSuper of MethodDefinition` is `true`, then
   a. Let `closure` be the result of performing the `FunctionCreate abstract operation` with arguments `Method`, `PropertySetParameterList`, `FunctionBody`, `scope`, and `strict` and with `propKey` as the `methodName` optional argument and `object` as the `homeObject` optional argument.
6. Else
   a. Let `closure` be the result of performing the `FunctionCreate abstract operation` with arguments `Method`, `PropertySetParameterList`, `FunctionBody`, `scope`, and `strict`.
7. Set `FunctionName(closure, propKey, "set")`.
8. Let `desc` be the `PropertyDescriptor{[[Set]]: closure, [[Enumerable]]: true, [[Configurable]]: true}`
9. Return the result of `DefinePropertyOrThrow(object, propKey, desc)`.

14.4 Generator Function Definitions

**Syntax**

```plaintext
GeneratorMethod: * PropertyName (StrictFormalParameters) { FunctionBody }
GeneratorDeclaration: function * BindingIdentifier (FormalParameters) ( FunctionBody )
GeneratorExpression: function * BindingIdentifier (FormalParameters) ( FunctionBody )
YieldExpression: yield ...
```

**Supplemental Syntax**

The following productions are used as an aid in specifying the semantics of certain ECMAScript language features. They are not used when parsing ECMAScript source code.

- `GeneratorBody`: `FunctionBody Comprehension`
- `NOTE:` Abstract operations relating to generator objects are defined in 25.3.3.

14.4.1 Static Semantics: Early Errors

```plaintext
GeneratorMethod: * PropertyName (StrictFormalParameters) { FunctionBody }

- It is a Syntax Error if any element of the `BoundNames` of `StrictFormalParameters` also occurs in the `VarDeclaredNames` of `FunctionBody`.
- It is a Syntax Error if any element of the `BoundNames` of `StrictFormalParameters` also occurs in the `LexicallyDeclaredNames` of `FunctionBody`.
```

```plaintext
GeneratorDeclaration: function * BindingIdentifier (FormalParameters) ( FunctionBody )
and
GeneratorExpression: function * BindingIdentifier (FormalParameters) ( FunctionBody )

- If the source code matching this production is strict code, the Early Error rules for `StrictFormalParameters`: `FormalParameters` are applied.
```
• It is a Syntax Error if IsSimpleParameterList of FormalParameters is `false` and any element of the BoundNames of FormalParameters also occurs in the VarDeclaredNames of FunctionBody.

14.4.2 Static Semantics: BoundNames

See also: 13.2.1.2, 13.2.2.1, 13.2.3.2, 13.6.4.2, 14.1.2, 14.2.2, 14.5.2, 15.1.1.2, 0.

GeneratorDeclaration: `function * BindingIdentifier ( FormalParameters ) { FunctionBody }`

1. Return the BoundNames of BindingIdentifier.

14.4.3 Static Semantics: Contains

With parameter symbol.

See also: 5.3, 12.1.5.2, 12.2.1.1, 14.1.3, 14.2.3, 14.5.4

GeneratorDeclaration: `function * BindingIdentifier ( FormalParameters ) { FunctionBody }`

1. Return `false`.

GeneratorExpression: `function * BindingIdentifier_{opt} ( FormalParameters ) { FunctionBody }`

1. Return `false`.

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

14.4.4 Static Semantics: HasComputedPropertyKey

See also: 12.1.5.3, 14.3.3

GeneratorMethod: `* PropertyName ( StrictFormalParameters ) { FunctionBody }

1. Return IsComputedPropertyKey of PropertyName.

14.4.5 Static Semantics: IsAnonymousFunctionDefinition

See also: 12.1.0.2, 12.1.10.2, 12.2.1.2, 12.3.2, 12.4.2, 12.5.1, 12.6.1, 12.7.1, 12.8.1, 12.9.2, 12.10.2, 12.11.1, 12.12.1, 12.13.2, 12.14.2, 14.1.6, 14.5.5.

GeneratorExpression: `function * ( FormalParameters ) { FunctionBody }`

1. Return `true`.

GeneratorExpression: `function * BindingIdentifier ( FormalParameters ) { FunctionBody }`

1. Return `false`.

14.4.6 Static Semantics: IsConstantDeclaration

See also: 13.2.1.3, 14.1.6, 14.5.5.

GeneratorDeclaration: `function * BindingIdentifier ( FormalParameters ) { FunctionBody }`

1. Return `false`. 
14.4.7 Static Semantics: LexicallyDeclaredNames
See also: 13.1.3, 13.11.3, 14.1.10, 14.2.7, 14.5.8, 15.1.0.7, 15.2.3.

GeneratorDeclaration: `function * BindingIdentifier { FormalParameters } { FunctionBody }

1. Return the BoundNames of BindingIdentifier.

14.4.8 Static Semantics: PropName
See also: 12.1.5.5, 14.3.5, 14.5.11

GeneratorMethod: ` * PropertyName (StrictFormalParameters ) { FunctionBody }

1. Return PropName of PropertyName.

14.4.9 Static Semantics: ReferencesSuper
See also: 14.3.6.

GeneratorMethod: ` * PropertyName ( StrictFormalParameters ) { FunctionBody }

1. If StrictFormalParameters Contains `super` is true, then return true.
2. Return FunctionBody Contains super.

14.4.10 Static Semantics: VarDeclaredNames
See also: 13.0.1, 13.1.8, 13.5.1, 13.6.2.1, 13.6.3.1, 13.6.4.3, 13.10.2, 13.11.4, 13.12.2, 13.14.2, 14.1.11, 14.5.14, 15.1.0.12, 15.2.5.

GeneratorDeclaration: `function * BindingIdentifier { FormalParameters } { FunctionBody }

1. Return an empty List.

14.4.11 Runtime Semantics: EvaluateBody

   With parameter functionObject.

See also: 14.1.12, 14.2.9.

GeneratorBody: FunctionBody

1. Assert: A Function Environment Record containing a this binding has already been activated as the current environment.
2. Let env be the result of the GetThisEnvironment abstract operation.
3. Let G be the result of calling the GetThisBinding concrete method of env.
4. If Type(G) is not Object or if Type(G) is Object and G does not have a [[GeneratorState]] internal slot or if Type(G) is Object and G has a [[GeneratorState]] internal slot and the value of G’s [[GeneratorState]] internal slot is not undefined, then
   a. Let newG be the result of calling OrdinaryCreateFromConstructor(functionObject, "%GeneratorPrototype!", ( [[GeneratorState]], [[GeneratorContext]] ) ).
   b. ReturnIfAbrupt(newG).
   c. Let G be newG.
5. Return the result of GeneratorStart(G, FunctionBody).
1. Let $G$ be the result of ObjectCreate() with arguments `%GeneratorPrototype%`, `[[GeneratorState]]`, and `[[GeneratorContext]]`. 
2. ReturnIfAbrupt($G$). 
3. Assert: The value of $G$’s `[[GeneratorState]]` internal slot is `undefined`. 
4. Let `startStatus` be the result of `GeneratorStart(G, Comprehension)`. 
5. ReturnIfAbrupt(startStatus). 

14.4.12 Runtime Semantics: `InstantiateFunctionObject` 

With parameter `scope`. 

See also: 14.1.14.

`GeneratorDeclaration` : `function` `*` `BindingIdentifier` `{ ` `FormalParameters` `}` `{ ` `FunctionBody` `}`

1. If the `GeneratorDeclaration` is contained in strict code or if its `FunctionBody` is strict code, then let `strict` be `true`. Otherwise let `strict` be `false`. 
2. Let `name` be `StringValue` of `BindingIdentifier`. 
3. Using `FunctionBody` from the production that is being evaluated, let `body` be the supplemental syntactic grammar production: `GeneratorBody` : `FunctionBody`. 
4. Let `F` be the result of performing the `GeneratorFunctionCreate` abstract operation with arguments `Normal`, `FormalParameters`, `body`, `scope`, `strict`, and `name`. 
5. Let `prototype` be the result of the abstract operation `ObjectCreate` with the intrinsic object `%GeneratorPrototype%` as its argument. 
6. Perform the abstract operation `MakeConstructor` with arguments `F`, `true`, and `prototype`. 
7. Return `F`.

14.4.13 Runtime Semantics: `PropertyDefinitionEvaluation` 

With parameter `object`. 

See also: 12.1.5.8, 14.3.9, B.3.1

`GeneratorMethod` : `*` `PropertyName` `{ ` `StrictFormalParameters` `}` `{ ` `FunctionBody` `}`

1. Let `propKey` be the result of evaluating `PropertyName`. 
2. ReturnIfAbrupt(propKey). 
3. Let `strict` be `IsStrict` of `FunctionBody`. 
4. Let `scope` be the running execution context’s `LexicalEnvironment`. 
5. Using `FunctionBody` from the production that is being evaluated, let `body` be the supplemental syntactic grammar production: `GeneratorBody` : `FunctionBody`. 
6. If `RecursiveSuper` of `GeneratorMethod` is `true`, then 
   a. Let `closure` be the result of performing the `GeneratorFunctionCreate` abstract operation with arguments `Method`, `StrictFormalParameters`, `body`, `scope`, and `strict` and with `propKey` as the `methodName` optional argument and `object` as the `homeObject` optional argument. 
7. Else 
   a. Let `closure` be the result of performing the `GeneratorFunctionCreate` abstract operation with arguments `Method`, `StrictFormalParameters`, `body`, `scope`, and `strict`. 
8. Let `prototype` be the result of the abstract operation `ObjectCreate` with the intrinsic object `%GeneratorPrototype%` as its argument. 
9. Perform the abstract operation `MakeConstructor` with arguments `closure`, `true`, and `prototype`. 
10. Set `FunctionName(closure, propKey)`. 
12. Return the result of `DefinePropertyOrThrow(object, propKey, desc)`.

© Ecma International 2009 – All rights reserved

231
14.4.14 Runtime Semantics: Evaluation

GeneratorDeclaration : function * BindingIdentifier { FormalParameters } { FunctionBody }

1. Return NormalCompletion(empty)

GeneratorExpression : function * { FormalParameters } { FunctionBody }

1. If the GeneratorExpression is contained in strict code or if its FunctionBody is strict code, then let strict be true. Otherwise let strict be false.
2. Using FunctionBody from the production that is being evaluated, let body be the supplemental syntactic grammar production: GeneratorBody : FunctionBody.
3. Let scope be the LexicalEnvironment of the running execution context.
4. Let closure be the result of performing the GeneratorFunctionCreate abstract operation with arguments Normal, FormalParameters, body, scope, and strict.
5. Let prototype be the result of the abstract operation ObjectCreate with the intrinsic object %GeneratorPrototype% as its argument.
6. Perform the abstract operation MakeConstructor with arguments closure, true, and prototype.
7. Return closure.

GeneratorExpression : function * BindingIdentifier { FormalParameters } { FunctionBody }

1. If the GeneratorExpression is contained in strict code or if its FunctionBody is strict code, then let strict be true. Otherwise let strict be false.
2. Using FunctionBody from the production that is being evaluated, let body be the supplemental syntactic grammar production: GeneratorBody : FunctionBody.
3. Let funcEnv be the result of calling NewDeclarativeEnvironment passing the running execution context’s Lexical Environment as the argument.
4. Let envRec be funcEnv’s environment record.
5. Let name be StringValue of BindingIdentifier.
6. Call the CreateImmutableBinding concrete method of envRec passing name as the argument.
7. Let closure be the result of performing the GeneratorFunctionCreate abstract operation with arguments Normal, FormalParameters, body, funcEnv, and strict and with name as the methodName optional argument.
8. Let prototype be the result of the abstract operation ObjectCreate with the intrinsic object %GeneratorPrototype% as its argument.
9. Perform the abstract operation MakeConstructor with arguments closure, true, and prototype.
10. Call the InitialiseBinding concrete method of envRec passing name and closure as the arguments.
11. Return NormalCompletion(closure).

NOTE 1 The BindingIdentifier in a GeneratorExpression can be referenced from inside the GeneratorExpression’s FunctionBody to allow the generator code to call itself recursively. However, unlike in a GeneratorDeclaration, the BindingIdentifier in a GeneratorExpression cannot be referenced from and does not affect the scope enclosing the GeneratorExpression.

YieldExpression : yield

1. Return the result of GeneratorYield(CreateIterResultObject(undefined, false)).

YieldExpression : yield AssignmentExpression

1. Let exprRef be the result of evaluating AssignmentExpression.
2. Let value be GetValue(exprRef).
3. ReturnIfAbrupt(value).
4. Return the result of GeneratorYield(CreateIterResultObject(value, false)).
YieldExpression: \textbf{yield}\, * AssignmentExpression

1. Let \textit{exprRef} be the result of evaluating \textit{AssignmentExpression}.
2. Let \textit{value} be \textit{GetValue(exprRef)}.
3. ReturnIfAbrupt(\textit{value}).
4. Let \textit{iterator} be the result of \textit{GetIterator(value)}.
5. ReturnIfAbrupt(\textit{iterator}).
6. Let \textit{received} be \textit{NormalCompletion}(\textit{undefined}).
7. Repeat
   a. If \textit{received}.[[type]] is \texttt{normal}, then
      i. Let \textit{innerResult} be the result of \textit{IteratorNext(iterator, received}.[[value]]).
      ii. ReturnIfAbrupt(\textit{innerResult}).
   b. Else
      i. Assert: \textit{received}.[[type]] is \texttt{throw}.
         ii. If \textit{HasProperty(iterator, "throw")] is \texttt{true}, then
             1. Let \textit{innerResult} be the result of \textit{Invoke(iterator, "throw", (received}.[[value]])).
             2. ReturnIfAbrupt(\textit{innerResult}).
             3. If Type(\textit{innerResult}) is not \texttt{Object}, then throw a \texttt{TypeError} exception.
         iii. Else, return \textit{received}.
    c. Let \textit{done} be \textit{IteratorComplete(\textit{innerResult})}.
    d. ReturnIfAbrupt(\textit{done}).
    e. If \textit{done} is \texttt{true}, then
       i. Return \textit{IteratorValue(\textit{innerResult})}.
    f. Let \textit{received} be the result of \textit{GeneratorYield(\textit{innerResult})}.

14.5 Class Definitions

Syntax

ClassDeclaration[default]
   \textbf{class} BindingIdentifier[default] ClassTail

ClassExpression
   \textbf{class} BindingIdentifier \opt ClassTail

ClassTail
   ClassHeritage \opt \{ ClassBody \opt \}

ClassHeritage
   \textbf{extends} AssignmentExpression

ClassBody
   \textbf{ClassElementList}

ClassElementList
   \textbf{ClassElement ClassElementList} \textbf{ClassElement}

ClassElement
   MethodDefinition \opt \textbf{static} MethodDefinition

\textbf{NOTE}
   A \textit{ClassBody} is always strict code.

14.5.1 Static Semantics: Early Errors

ClassBody : ClassElementList
• It is a Syntax Error if PrototypePropertyNameList 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.

• It is a Syntax Error if StaticPropertyNameList 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.

ClassElement : static MethodDefinition

• It is a Syntax Error if PropName of MethodDefinition is "prototype".

14.5.2 Static Semantics: BoundNames

See also: 13.2.1.2, 13.2.2.1, 13.2.3.2, 13.6.4.2, 14.1.2, 14.2.2, 14.4.2, 15.1.1.2, 0.

ClassDeclaration: class BindingIdentifier ClassTail

1. Return the BoundNames of BindingIdentifier.

14.5.3 Static Semantics: ConstructorMethod

ClassElementList : ClassElement

1. If ClassElement is the production ClassElement : ; then, return empty.
2. If IsStatic of ClassElement is true, return empty.
3. If PropName of ClassElement is not "constructor", return empty.
4. Return ClassElement.

ClassElementList : ClassElementList ClassElement

1. Let head be ConstructorMethod of ClassElementList.
2. If head is not empty, return head.
3. If ClassElement is the production ClassElement : ; then, return empty.
4. If IsStatic of ClassElement is true, return empty.
5. If PropName of ClassElement is not "constructor", return empty.
6. Return ClassElement.

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.

14.5.4 Static Semantics: Contains

With parameter symbol.

See also: 5.3, 12.1.5.2, 12.2.1.1, 14.1.3, 14.2.3, 14.4.3

ClassTail : ClassHeritageopt ( 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.

14.5.5 Static Semantics: IsAnonymousFunctionDefinition

See also: 12.1.0.2, 12.1.10.2, 12.2.1.2, 12.3.2, 12.4.2, 12.5.1, 12.6.1, 12.7.1, 12.8.1, 12.9.2, 12.10.2, 12.11.1, 12.12.1, 12.13.1, 12.14.2, 14.1.6, 14.4.5.

ClassExpression : class ClassTail

1. Return true.

ClassExpression : class BindingIdentifier ClassTail

1. Return false.

14.5.6 Static Semantics: IsConstantDeclaration

See also: 13.2.1.3, 14.1.6, 14.4.4.

ClassDeclaration : class BindingIdentifier ClassTail

1. Return false.

14.5.7 Static Semantics: IsStatic

ClassElement : MethodDefinition

1. Return false.

ClassElement : static MethodDefinition

1. Return true.

ClassElement : ;

1. Return false.

14.5.8 Static Semantics: LexicallyDeclaredNames

See also: 13.1.3, 13.11.3, 14.1.10, 14.2.7, 14.4.7, 15.1.0.7, 15.2.3.

ClassDeclaration : class BindingIdentifier ClassTail

1. Return the BoundNames of BindingIdentifier.

14.5.9 Static Semantics: PrototypeMethodDefinitions

ClassElementList : ClassElement

1. If ClassElement is the production ClassElement : ; then, return a new empty List.
2. If IsStatic of ClassElement is true, return a new empty List.
3. If PropName of ClassElement is "constructor", return a new empty List.
4. Return a List containing ClassElement.
1. Let list be PrototypeMethodDefinitions of ClassElementList.
2. If ClassElement is the production ClassElement : ; then, return list.
3. If IsStatic of ClassElement is true, return list.
4. If PropName of ClassElement is "constructor", return list.
5. Append ClassElement to the end of list.
6. Return list.

14.5.10 Static Semantics: PrototypePropertyNameList

ClassElementList : ClassElement

1. If PropName of ClassElement is empty, return a new empty List.
2. If IsStatic of ClassElement is true, return a new empty List.
3. Return a List containing PropName of ClassElement.

ClassElementList : ClassElementList ClassElement

1. Let list be PrototypePropertyNameList of ClassElementList.
2. If PropName of ClassElement is empty, return list.
3. If IsStatic of ClassElement is true, return list.
4. Append PropName of ClassElement to the end of list.
5. Return list.

14.5.11 Static Semantics: PropName

See also: 12.1.5.5, 14.3.5, 14.4.8

ClassElement : ;

1. Return empty.

14.5.12 Static Semantics: StaticPropertyNameList

ClassElementList : ClassElement

1. If PropName of ClassElement is empty, return a new empty List.
2. If IsStatic of ClassElement is false, return a new empty List.
3. Return a List containing PropName of ClassElement.

ClassElementList : ClassElementList ClassElement

1. Let list be StaticPropertyNameList of ClassElementList.
2. If PropName of ClassElement is empty, return list.
3. If IsStatic of ClassElement is false, return list.
4. Append PropName of ClassElement to the end of list.
5. Return list.

14.5.13 Static Semantics: StaticMethodDefinitions

ClassElementList : ClassElement

1. If ClassElement is the production ClassElement : ; then, return a new empty List.
2. If IsStatic of ClassElement is false, return a new empty List.
3. Return a List containing ClassElement.
ClassElementList : ClassElementList ClassElement

1. Let list be StaticMethodDefinitions of ClassElementList.
2. If ClassElement is the production ClassElement : ; then, return list.
3. If IsStatic of ClassElement is false, return list.
4. Append ClassElement to the end of list.
5. Return list.

14.5.14 Static Semantics: VarDeclaredNames

See also: 13.0.1, 13.1.8, 13.5.1, 13.6.1.1, 13.6.2.1, 13.6.3.1, 13.6.4.3, 13.10.2, 13.11.4, 13.12.2, 13.14.2, 14.1.11, 14.4.10, 15.1.0.12, 15.2.5.

ClassDeclaration: class BindingIdentifier ClassTail

1. Return an empty List.

14.5.15 Runtime Semantics: ClassDefinitionEvaluation

With parameter className.

ClassTail : ClassHeritageopt { ClassBody }

1. If ClassHeritageopt 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. Let protoParent be the intrinsic object %FunctionPrototype%.
   c. Let constructorParent be the intrinsic object %FunctionPrototype%.
   d. If superclass is null, then
      i. Let protoParent be null.
      ii. Let constructorParent be the intrinsic object %FunctionPrototype%.
   e. Else if IsConstructor(superclass) is false, then throw a TypeError exception.
   f. Else
      i. Let protoParent be the result of Get(superclass, "prototype").
      ii. Let constructorParent be the intrinsic object %FunctionPrototype%.
      iii. If Type(protoParent) is neither Object nor 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.
7. If constructor is empty, then
   a. If ClassHeritageopt is present, then
      i. Let constructor be the result of parsing the String "constructor(... args){super (...args);}") using the syntactic grammar with the goal symbol MethodDefinition.
   b. Else
      i. Let constructor be the result of parsing the String "constructor( ){}" using the syntactic grammar with the goal symbol MethodDefinition.
8. Let strict be true.
9. Let constructorInfo be the result of performing DefineMethod for constructor with arguments proto and constructorParent as the optional functionPrototype argument.
10. Let F be constructorInfo.[[closure]].
11. Perform the abstract operation MakeConstructor with argument F and false as the optional writablePrototype argument and proto as the optional prototype argument.

© Ecma International 2009 – All rights reserved
12. If className is not undefined, then
   a. Call the InitialiseBinding concrete method of envRec passing className and F as the arguments.
13. Let desc be the PropertyDescriptor([[Enumerable]], false, [[Writable]], true, [[Configurable]], true).
14. Call the [[DefineOwnProperty]] internal method of proto with arguments "constructor" and desc.
15. Let protoMethods be PrototypeMethodDefinitions of ClassBody.
16. For each MethodDefinition m in order from protoMethods
   a. Let status be the result of performing PropertyDefinitionEvaluation for m with argument proto.
   b. ReturnIfAbrupt(status).
17. Let staticMethods be StaticMethodDefinitions of ClassBody.
18. For each MethodDefinition s in order from staticMethods
   a. Let status be the result of performing PropertyDefinitionEvaluation for s with argument F.
   b. ReturnIfAbrupt(status).
19. Set the running execution context’s LexicalEnvironment to lex.
20. Return F.

14.5.16 Runtime Semantics: Evaluation

```javascript
ClassDeclaration: class BindingIdentifier ClassTail
```
1. Let className be StringValue(BindingIdentifier).
2. Let value be the result of ClassDefinitionEvaluation of ClassTail with argument className.
3. ReturnIfAbrupt(value).
4. Let hasNameProperty be the result of HasOwnProperty(value, "name").
5. ReturnIfAbrupt(hasNameProperty).
6. If hasNameProperty is false, then
   a. Perform SetFunctionName(value, className).
7. Let env be the running execution context’s LexicalEnvironment.
8. Let status be the result of performing BindingInitialisation for BindingIdentifier passing value and env as the arguments.
9. ReturnIfAbrupt(status).
10. Return NormalCompletion(empty).

```javascript
ClassExpression: class BindingIdentifier ClassTail
```
1. If BindingIdentifier is not present, then let className be undefined.
2. Else, let className be StringValue(BindingIdentifier).
3. Let value be the result of ClassDefinitionEvaluation of ClassTail with argument className.
4. ReturnIfAbrupt(value).
5. If className is not undefined, then
   a. Let hasNameProperty be the result of HasOwnProperty(value, "name").
   b. ReturnIfAbrupt(hasNameProperty).
   c. If hasNameProperty is false, then
      i. Perform SetFunctionName(value, className).
6. Return NormalCompletion(value).

14.6 Tail Position Calls

14.6.1 Static Semantics: Tail Position

With parameter nonTerminal.

1. Assert: nonTerminal is a parsed grammar production.
2. If the source code match nonterminal is not strict code, then return false.
3. If nonTerminal is not contained within a FunctionBody, then return false.
4. Let body be the FunctionBody that most closely contains nonTerminal.

© Ecma International 2013
5. If `body` is the `FunctionBody` of a `GeneratorMethod`, `GeneratorDeclaration`, or a `GeneratorExpression`, then return `false`.

6. Let `has` be `HasProductionInTailPosition` of `body` with argument `nonterminal`.

**NOTE** Tail Position calls are only defined in strict mode code because of a common non-standard language extension (see 9.2.8) that enables observation of the chain of caller contexts.

### 14.6.2 Static Semantics: `HasProductionInTailPosition`

With parameter `nonTerminal`.

#### 14.6.2.1 Statement Rules

- **StatementList**: StatementList StatementListItem
  1. Let `has` be `HasProductionInTailPosition` of `StatementList` with argument `nonterminal`.
  2. If `has` is `true`, then return `true`.
  3. Return `HasProductionInTailPosition` of `StatementListItem` with argument `nonterminal`.

- **StatementListItem**: Declaration

- **Statement**: VariableStatement EmptyStatement ExpressionStatement ContinueStatement BreakStatement ThrowStatement DebuggerStatement

- **ReturnStatement**: `return`;

- **CaseBlock**: `{ }`

  1. Return `false`.

- **IfStatement**: `if` (Expression) Statement **else** Statement

  1. Let `has` be `HasProductionInTailPosition` of the first `Statement` with argument `nonterminal`.
  2. If `has` is `true`, then return `true`.
  3. Return `HasProductionInTailPosition` of the second `Statement` with argument `nonterminal`.

- **IfStatement**: `if` (Expression) Statement

- **IterationStatement**: `do` Statement `while` (Expression) `;`opt

  - `while` (Expression) `Statement`

  - `for` (Expression; Expression; Expression) `Statement`

  - `for` (LexicalDeclaration Expression; Expression) `Statement`

  - `for` (LeftHandSideExpression in Expression) `Statement`

  - `for` (ForDeclaration in Expression) `Statement`

  - `for` (LeftHandSideExpression of AssignmentExpression) `Statement`

  - `for` (ForDeclaration of AssignmentExpression) `Statement`

- **WithStatement**: `with` (Expression) `Statement`

- **LabelledStatement**: IdentifierReference : Statement

- **yield** : Statement
1. Return HasProductionInTailPosition of Statement with argument nonterminal.

ReturnStatement : return Expression ;

1. Return HasProductionInTailPosition of Expression with argument nonterminal.

SwitchStatement : switch ( Expression ) CaseBlock

1. Return HasProductionInTailPosition of CaseBlock with argument nonterminal.

CaseBlock : { CaseClauses DefaultClause CaseClausesopt }

1. Let has be false.
2. If the first CaseClauses is present, let has be HasProductionInTailPosition of the first CaseClauses with argument nonterminal.
3. If has is true, then return true.
4. Let has be HasProductionInTailPosition of the DefaultClause with argument nonterminal.
5. If has is true, then return true.
6. If the second CaseClauses is present, let has be HasProductionInTailPosition of the second CaseClauses with argument nonterminal.
7. Return has.

CaseClauses : CaseClauses CaseClause

1. Let has be HasProductionInTailPosition of CaseClauses with argument nonterminal.
2. If has is true, then return true.
3. Return HasProductionInTailPosition of CaseClause with argument nonterminal.

CaseClause : case Expression : StatementListopt
DefaultClause : default : StatementListopt

1. If StatementList is present, return HasProductionInTailPosition of StatementList with argument nonterminal.
2. Return false.

TryStatement : try Block Catch

1. Return HasProductionInTailPosition of Catch with argument nonterminal.

TryStatement : try Block Finally
TryStatement : try Block Catch Finally

1. Return HasProductionInTailPosition of Finally with argument nonterminal.

Catch : catch ( CatchParameter ) Block

1. Return HasProductionInTailPosition of Finally with argument nonterminal.

14.6.2.2 Expression Rules

NOTE A potential tail position call that is immediately followed by return GetValue of the call result is also a possible tail position call. Functional calls can not return reference values, so such a GetValue operation will always returns the same value as the actual function call result.
AssignmentExpression:
    ConditionalExpression
    YieldExpression
    ArrowFunction
    LeftHandSideExpression = AssignmentExpression
    LeftHandSideExpression AssignmentOperator AssignmentExpression

BitwiseANDExpression : BitwiseANDExpression & EqualityExpression
BitwiseXORExpression : BitwiseXORExpression ^ BitwiseANDExpression
BitwiseORExpression : BitwiseORExpression | BitwiseXORExpression

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

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

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

AdditiveExpression :
    AdditiveExpression + MultiplicativeExpression
    AdditiveExpression - MultiplicativeExpression

MultiplicativeExpression :
    MultiplicativeExpression * UnaryExpression
    MultiplicativeExpression / UnaryExpression
    MultiplicativeExpression % UnaryExpression

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

PostfixExpression :
    LeftHandSideExpression ++
    LeftHandSideExpression --

CallExpression :
    CallExpression [ Expression ]
    CallExpression . IdentifierName
MemberExpression :
  MemberExpression { Expression } 
  MemberExpression . IdentifierName

super [ Expression ]
super . IdentifierName

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

1. Return false.

Expression :
  AssignmentExpression
  Expression , AssignmentExpression
1. Return HasProductionInTailPosition of AssignmentExpression with argument nonterminal.

ConditionalExpression : LogicalORExpression ? AssignmentExpression : AssignmentExpression
1. Let has be HasProductionInTailPosition of the first AssignmentExpression with argument nonterminal.
2. If has is true, then return true.
3. Return HasProductionInTailPosition of the second AssignmentExpression with argument nonterminal.

LogicalANDExpression : LogicalANDExpression && BitwiseORExpression
1. Return HasProductionInTailPosition of BitwiseORExpression with argument nonterminal.

LogicalORExpression : LogicalORExpression || LogicalANDExpression
1. Return HasProductionInTailPosition of LogicalANDExpression with argument nonterminal.

CallExpression :
  MemberExpression Arguments
  super Arguments
  CallExpression Arguments
  CallExpression TemplateLiteral
1. If this CallExpression is nonTerminal, then return true.
2. Return undefined.

MemberExpression :
  MemberExpression TemplateLiteral
  new super Arguments
  new MemberExpression Arguments
1. If this MemberExpression is nonTerminal, then return true.
2. Return false.

NewExpression:
   new NewExpression
   1. If this NewExpression is nonTerminal, then return true.
   2. Return false.

PrimaryExpression: CoverParenthesisedExpressionAndArrowParameterList
   1. Let expr be CoveredParenthesisedExpression of CoverParenthesisedExpressionAndArrowParameterList.
   2. Return HasProductionInTailPosition of expr with argument nonterminal.

ParenthesisedExpression:
   ( Expression )
   1. Return HasProductionInTailPosition of Expression with argument nonterminal.

14.6.3 Runtime Semantics: PrepareForTailCall

The abstract operation PrepareForTailCall performs the following steps:
1. Let leafContext be the running execution context.
2. Suspend leafContext.
3. 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.
4. Assert: leafContext has no further use. It will never be activated as the running execution context.

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 activation 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.

15 ECMAScript Language: Modules and Scripts

This section is incomplete
15.1.0 Module Semantics

15.1.0.1 Static Semantics: Early Errors

ModuleBody : ModuleItemList
• It is a Syntax Error if the LexicallyDeclaredNames of ModuleItemList contains any duplicate entries.
• It is a Syntax Error if the ExportedBindings of ModuleItemList contains any duplicate entries.
• It is a Syntax Error if any element of the LexicallyDeclaredNames of ModuleItemList also occurs in the VarDeclaredNames of ModuleItemList.
• It is a Syntax Error if ModuleItemList Contains super.

NOTE Additional error conditions relating to conflicting or duplicate declarations are checked during module linking prior to evaluation of a Module. If any such errors are detected the Module is not evaluated.

15.1.0.2 Static Semantics: ExportedBindings

See also: 15.1.2.2.

ModuleItemList : [empty]
1. Return a new empty List.

ModuleItemList : ModuleItemList ModuleItem
1. Let names be ExportedBindings of ModuleItemList.
2. Append to names the elements of the ExportedBindings of ModuleItem.
3. Return names.

ModuleItem:
ImportDeclaration
StatementListItem
1. Return a new empty List.

15.1.0.3 Static Semantics: ExportEntries

See also: 15.1.2.3.

ModuleItemList : [empty]
1. Return a new empty List.

ModuleItemList : ModuleItemList ModuleItem
1. Let entries be ExportEntries of ModuleItemList.
2. Append to entries the elements of the ExportEntries of ModuleItem.
3. Return entries.

ModuleItem:
ImportDeclaration
StatementListItem
1. Return a new empty List.
15.1.0.4 Static Semantics: ImportedBindings

ModuleItemList : [empty]
1. Return a new empty List.

ModuleItemList : ModuleItemList ModuleItem

1. Let names be ImportedBindings of ModuleItemList.
2. Append to names the elements of the ImportedBindings of ModuleItem.
3. Return names.

ModuleItem: ImportDeclaration
1. Return the BoundNames of ImportDeclaration.

ModuleItem :
  ExportDeclaration
  StatementList
1. Return a new empty List.

15.1.0.5 Static Semantics: ImportEntries

See also: 15.1.1.3.

ModuleItemList : [empty]
1. Return a new empty List.

ModuleItemList : ModuleItemList ModuleItem

1. Let entries be ImportEntries of ModuleItemList.
2. Append to entries the elements of the ImportEntries of ModuleItem.
3. Return entries.

ModuleItem :
  ExportDeclaration
  StatementList
1. Return a new empty List.

15.1.0.6 Static Semantics: IsStrict

See also: 14.1.9, 15.2.2.

ModuleBody : ModuleItemList
1. Return true.

15.1.0.7 Static Semantics: KnownExportEntries

ModuleBody : ModuleItemList

1. Let allExports be ExportEntries of ModuleItemList.
2. Return a new List containing all the entries of allExports whose [[ImportName]] field is not all.

15.1.0.8 Static Semantics: ModuleRequests

See also: 15.1.1.5, 15.1.2.5.
ModuleItemList : [empty]
1. Return a new empty List.

ModuleItemList : ModuleItem
1. Return ModuleRequests of ModuleItemList.

ModuleItemList : ModuleItemList ModuleItem
1. Let moduleNames be ModuleRequests of ModuleItemList.
2. Let additionalNames be ModuleRequests of ModuleItem.
3. Append to moduleNames each element of additionalNames that is not already an element of moduleNames.
4. Return moduleNames.

ModuleItem : StatementListItem
1. Return a new empty List.

15.1.0.9 Static Semantics: LexicallyDeclaredNames
See also: 13.1.3, 13.11.3, 14.1.10, 14.2.7, 14.4.7, 14.5.8, 15.2.3.

ModuleItemList : [empty]
1. Return a new empty List.

ModuleItemList : ModuleItemList ModuleItem
1. Let names be LexicallyDeclaredNames of ModuleItemList.
2. Append to names the elements of the LexicallyDeclaredNames of ModuleItem.
3. Return names.

ModuleItem : ImportDeclaration
1. Return the BoundNames of ImportDeclaration.

ModuleItem : ExportDeclaration
1. Return the BoundNames of ExportDeclaration.

ModuleItem : StatementListItem
1. Return LexicallyDeclaredNames of StatementListItem.

NOTE At the top level of a Module, function declarations are treated like lexical declarations rather than like var declarations.

15.1.0.10 Static Semantics: LexicalDeclarations
See also: 13.1.2, 13.11.2.
1. Let declarations be LexicalDeclarations of ModuleItemList.
2. Append to declarations the elements of the LexicalDeclarations of ModuleItem.
3. Return declarations.

ModuleItem : ImportDeclaration
1. If the BoundNames of ImportDeclarations is empty, then return an empty List.
2. Return a new List containing ImportDeclaration.

ModuleItem : ExportDeclaration
1. If ExportDeclaration is export Declaration; then return a new List containing Declaration.
2. Return a new empty List.

15.1.0.11 Static Semantics: UnknownExportEntries

ModuleBody : ModuleItemList
1. Let allExports be ExportEntries of ModuleItemList.
2. Return a new List containing all the entries of allEntries whose [[ImportName]] field is all.

15.1.0.12 Static Semantics: VarDeclaredNames


ModuleItemList : ModuleItemList ModuleItem
1. Let names be VarDeclaredNames of ModuleItemList.
2. Append to names the elements of the VarDeclaredNames of ModuleItem.
3. Return names.

ModuleItem : ImportDeclaration
1. Return an empty List.

ModuleItem : ExportDeclaration
1. If ExportDeclaration is export VariableStatement; then return BoundNames of ExportDeclaration.
2. Return a new empty List.

15.1.0.13 Static Semantics: VarScopedDeclarations

See also: 13.1.9, 15.2.6.

ModuleItemList : [empty]
1. Return a new empty List.

ModuleItemList : ModuleItemList ModuleItem
1. Let declarations be VarScopedDeclarations of ModuleItemList.
2. Append to declarations the elements of the VarScopedDeclarations of ModuleItem.
3. Return declarations.

ModuleItem : ImportDeclaration
1. Return a new empty List.

ModuleItem : ExportDeclaration

1. If `ExportDeclaration` is `export VariableStatement` then return a new List containing `VariableStatement`.
2. Return a new empty List.

15.1.1 Imports

`ImportDeclaration`:
```
ModuleImport
  import ImportClause FromClause ;
  import ModuleSpecifier ;
```

`ModuleImport`:
```
module [no LineTerminator here] ImportedBinding FromClause ;
```

`FromClause`:
```
  from ModuleSpecifier
```

`ImportClause`:
```
  ImportedBinding
  ImportedBinding , NamedImports
  NamedImports
```

`NamedImports`:
```
  {  }
  { ImportsList }
  { ImportsList , }
```

`ImportsList`:
```
  ImportSpecifier
  ImportsList , ImportSpecifier
```

`ImportSpecifier`:
```
  ImportedBinding
  IdentifierName as ImportedBinding
```

`ModuleSpecifier`:
```
  StringLiteral
```

`ImportedBinding`:
```
  BindingIdentifier
```

15.1.1.1 Static Semantics: Early Errors

`ModuleItem` : `ImportDeclaration`
```
- It is a Syntax Error if the BoundNames of `ImportDeclaration` contains any duplicate entries.
```

15.1.1.2 Static Semantics: BoundNames

See also: 13.2.1.2, 13.2.2.1, 13.2.3.2, 13.6.4.2, 14.1.2, 14.2.2, 14.4.2, 14.5.2, 0.

`ImportDeclaration` : `import ImportClause FromClause` ;
```
1. Return the BoundNames of `ImportClause`.
```

`ImportDeclaration` : `import ModuleSpecifier` ;
1. Return a new empty List.

```
ModuleImport: module ImportedBinding FromClause;
```

1. Return the BoundNames of ImportedBinding.

```
ImportClause: ImportedBinding, NamedImports
```

1. Let names be the BoundNames of ImportedBinding.
2. Append to names the elements of the BoundNames of NamedImports.
3. Return names.

```
ImportsList: ImportsList, ImportSpecifier
```

1. Let names be the BoundNames of ImportsList.
2. Append to names the elements of the BoundNames of ImportSpecifier.
3. Return names.

```
ImportSpecifier: IdentifierName as ImportedBinding
```

1. Return the BoundNames of ImportedBinding.

### 15.1.1.3 Static Semantics: ImportEntries

See also: 15.1.0.5.

```
ImportDeclaration: import ImportClause FromClause;
```

1. Let module be the sole element of ModuleRequests of FromClause.
2. Return ImportEntriesForModule of ImportClause with argument module.

```
ImportDeclaration: import ModuleSpecifier;
```

1. Return a new empty List.

```
ModuleImport: module ImportedBinding FromClause;
```

1. Let module be ModuleRequests of FromClause.
2. Let localName be the StringValue of ImportedBinding.
3. Let entry be the Record { [[ModuleRequest]]: module, [[ImportName]]: "default", [[LocalName]]: localName }.
4. Return a new List containing entry.

### 15.1.1.4 Static Semantics: ImportEntriesForModule

With parameter module.

```
ImportClause: ImportedBinding, NamedImports
```

1. Let localName be the StringValue of ImportedBinding.
2. Let defaultEntry be the Record { [[ModuleRequest]]: module, [[ImportName]]: "default", [[LocalName]]: localName }.
3. Let entries be a new List containing defaultEntry.
4. Append to entries the elements of the ImportEntitiesForModule of NamedImports with argument module.
5. Return entries.

```
NamedImports: { }
```

1. Return a new empty List.
1. Let \( \text{module} \) be the \( \text{importEntitiesForModule} \) of \( \text{ImportsList} \) with argument \( \text{module} \).
2. Append to \( \text{specs} \) the elements of the \( \text{importEntitiesForModule} \) of \( \text{ImportSpecifier} \) with argument \( \text{module} \).
3. Return \( \text{specs} \).

**ImportSpecifier**

1. Let \( \text{localName} \) be the \( \text{StringValue} \) of \( \text{importedBinding} \).
2. Let \( \text{entry} \) be the Record `{ \[\text{ModuleName}\]: \text{module}, \[\text{ModuleName}\]: \text{localName}, \[\text{ModuleName}\]: \text{localName} }`.
3. Return a new List containing \( \text{entry} \).

**ImportSpecifier**

1. Let \( \text{importName} \) be the \( \text{StringValue} \) of \( \text{IdentifierName} \).
2. Let \( \text{localName} \) be the \( \text{StringValue} \) of \( \text{importedBinding} \).
3. Let \( \text{entry} \) be the Record `{ \[\text{ModuleName}\]: \text{module}, \[\text{ModuleName}\]: \text{importName}, \[\text{ModuleName}\]: \text{localName} }`.
4. Return a new List containing \( \text{entry} \).

### 15.1.1.5 Static Semantics: ModuleRequests

See also: 15.1.0.8, 15.1.2.5.

**ImportDeclaration**: `import \text{importClause} \text{fromClause} ;`

1. Return ModuleRequests of \( \text{fromClause} \).

**ModuleImport**: `module \text{importedBinding} \text{fromClause} ;`

1. Return ModuleRequests of \( \text{fromClause} \).

**ModuleSpecifier**: `\text{stringLiteral} \text{fromClause} ;`

1. Return a List containing the \( \text{SV} \) of \( \text{stringLiteral} \).

### 15.1.2 Exports

**ExportDeclaration**:

- `export * \text{fromClause} ;`
- `export \text{ExportClause} \text{fromClause} ;`
- `export \text{VariableStatement} ;`
- `export \text{Declaration} \text{fromClause} ;`
- `export \text{default AssignmentExpression} ;`

**ExportClauseElements**:

- `{ }`
- `{ \text{ExportsList} \text{fromClause} }`
- `{ \text{ExportsList} \text{fromClause} }`

**ExportsList**:

- `\text{ExportSpecifier} \text{fromClause} ;`
- `\text{ExportsList} \text{fromClause} , \text{ExportSpecifier} \text{fromClause} ;`

250 © Ecma International 2013
**NOTE** ExportSpecifier is used to export bindings from the enclosing module Module. ExportSpecifier is used to export bindings from a referenced Module. In that case IdentifierReference restrictions are not applied to the naming of the items too be exported because they are not used to create local bindings.

### 15.1.2.1 Static Semantics: BoundNames

See also: 13.2.1.2, 13.2.2.1, 13.2.3.2, 13.6.4.2, 14.1.2, 14.2.2, 14.4.2, 14.5.2, 15.1.1.2.

**ExportDeclaration:**

- `export * FromClause ;`
- `export ExportClause FromClause ;`
- `export ExportClause ;`

1. Return a new empty List.

**ExportDeclaration:** `export VariableStatement ;`

1. Return the BoundNames of VariableStatement.

**ExportDeclaration:** `export Declaration ;`

1. Return the BoundNames of Declaration.

**ExportDeclaration:** `export default AssignmentExpression ;`

1. Return a List containing "default".

### 15.1.2.2 Static Semantics: ExportedBindings

See also: 15.1.0.2.

**ExportDeclaration:** `export * FromClause ;`

1. Return a new empty List.

**ExportDeclaration:** `export ExportClause FromClause ;`
- `export ExportClause ;`

1. Return the ExportedBindings of this ExportClause.

**ExportDeclaration:**

- `export VariableStatement`
- `export Declaration[default]

1. Return the BoundNames of this ExportDeclaration.

**ExportDeclaration:** `export default AssignmentExpression ;`

1. Return a List containing "default".

**ExportClause:** `{ }`
1. Return a new empty List.

ExportsList : ExportList, ExportSpecifier
1. Let names be the ExportedBindings of ExportList.
2. Append to names the elements of the ExportedBindings of ExportSpecifier.
3. Return names.

ExportDeclaration : export ExportClause FromClauseopt ;
1. Return the ExportedBindings of ExportClause.

ExportSpecifier : IdentifierReference
1. Return a List containing the StringValue of IdentifierReference.

ExportSpecifier : IdentifierReference as IdentifierName
1. Return a List containing the StringValue of IdentifierName.

ExportSpecifier : IdentifierName
1. Return a List containing the StringValue of IdentifierName.

ExportSpecifier : IdentifierName as IdentifierName
1. Return a List containing the StringValue of the second IdentifierName.

15.1.2.3 Static Semantics: ExportEntries
See also:15.1.0.3.

ExportDeclaration : export * FromClause ;
1. Let module be the sole element of ModuleRequests of FromClause.
2. Let entry be the Record {[[ModuleRequest]]: module, [[ModuleName]]: all, [[LocalName]]: null,
   [[ExportName]]: null }.
3. Return a new List containing entry.

ExportDeclaration : export ExportClause FromClause ;
1. Let module be the sole element of ModuleRequests of FromClause.
2. Return ExportEntriesForModule of ExportClause with argument module.

ExportDeclaration : export ExportClause ;
1. Let module be the sole element of ModuleRequests of FromClause.
2. Return ExportEntriesForModule of ExportClause with argument null.

ExportDeclaration : export VariableStatement;
1. Let entries be a new empty List.
2. Let names be the BoundNames of VariableStatement.
3. For each name in names, do
   a. Append to entries the Record {[[ModuleRequest]]: null, [[ModuleName]]: null, [[LocalName]]:
      name, [[ExportName]]: name }.
4. Return `entries`.

```plaintext
ExportDeclaration: export Declaration;
```

1. Let `entries` be a new empty List.
2. Let `names` be the BoundNames of `Declaration`.
3. For each `name` in `names`, do
   a. Append to `entries` the Record `[[ModuleRequest]]: null, [[ImportName]]: null, [[LocalName]]: `name`, `[[ExportName]]: `name` `.`
4. Return `entries`.

```plaintext
ExportDeclaration: export default AssignmentExpression;
```

1. Let `entry` be the Record `[[ModuleRequest]]: null, [[ImportName]]: null, [[LocalName]]: "default", [[ExportName]]: "default"`.
2. Return a new List containing `entry`.

### 15.1.2.4 Static Semantics: ExportEntriesForModule

With parameter `module`.

```plaintext
ExportClause: { }
```

1. Return a new empty List.

```plaintext
ExportsList: ExportsList, ExportSpecifier
```

1. Let `specs` be the ExportEntriesForModule of `ExportsList` with argument `module`.
2. Append to `specs` the elements of the ExportEntitiesForModule of `ExportSpecifier` with argument `module`.
3. Return `specs`.

```plaintext
ExportSpecifier: IdentifierReference
```

1. Let `localName` be the StringValue of `IdentifierReference`.
2. Return a new List containing the Record `[[ModuleRequest]]: module, [[ImportName]]: null, [[LocalName]]: `localName`, `[[ExportName]]: `localName` `.`

```plaintext
ExportSpecifier: IdentifierReference as IdentifierName
```

1. Let `localName` be the StringValue of `IdentifierReference`.
2. Let `exportName` be the StringValue of `IdentifierName`.
3. Return a new List containing the Record `[[ModuleRequest]]: module, [[ImportName]]: null, [[LocalName]]: `localName`, `[[ExportName]]: `exportName` `.`

```plaintext
ExportSpecifier: IdentifierName
```

1. Let `sourceName` be the StringValue of `IdentifierName`.
2. Return a new List containing the Record `[[ModuleRequest]]: module, [[ImportName]]: `sourceName`, `[[LocalName]]: null, `[[ExportName]]: `sourceName` `.`

```plaintext
ExportSpecifier: IdentifierReference as IdentifierName
```

1. Let `sourceName` be the StringValue of the first `IdentifierReference`.
2. Let `exportName` be the StringValue of the second `IdentifierName`.
3. Return a new List containing the Record `[[ModuleRequest]]: module, [[ImportName]]: `sourceName`, `[[LocalName]]: null, `[[ExportName]]: `exportName` `.`

### 15.1.2.5 Static Semantics: ModuleRequests

See also: 15.1.0.8, 15.1.1.5.
ExportDeclaration : export ExportClause FromClause ;

1. Return the ModuleRequests of FromClause.

ExportDeclaration :
  export ExportClause ;
  export VariableStatement
  export Declaration
  export default AssignmentExpression ;

1. Return a new empty List.

15.2 Scripts

Syntax
Script :
  ScriptBodyopt
ScriptBody :
  StatementList

15.2.1 Static Semantics: Early Errors

ScriptBody : 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 ScriptItemList.
- It is a Syntax Error if ScriptItemList Contains super.

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.

15.2.2 Static Semantics: IsStrict

See also: 14.1.9, 15.1.0.2

ScriptBody : StatementList

1. If this ScriptBody is contained in strict code or if StatementList is strict code, then return true. Otherwise, return false.

15.2.3 Static Semantics: LexicallyDeclaredNames

See also: 13.1.3, 13.11.3, 14.1.10, 14.2.7, 14.4.7, 14.5.8, 15.1.0.7.

ScriptBody : StatementList

1. Return TopLevelLexicallyDeclaredNames of StatementList.

NOTE At the top level of a Script, function declarations are treated like var declarations rather than like lexical declarations.
15.2.4 Static Semantics: LexicallyScopedDeclarations

ScriptBody : StatementList
1. Return TopLevelLexicallyScopedDeclarations of StatementList.

15.2.5 Static Semantics: VarDeclaredNames


ScriptBody : StatementList
1. Return TopLevelVarDeclaredNames of StatementList.

15.2.6 Static Semantics: VarScopedDeclarations

ScriptBody : StatementList
1. Return TopLevelVarScopedDeclarations of StatementList.

15.2.7 Runtime Semantics: Script Evaluation

With argument realm and deletableBindings.

Script : ScriptBody
1. The code of this Script is strict mode code if the Directive Prologue (15.3) of its ScriptBody contains a Use Strict Directive or if any of the conditions of 10.2.1 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 GlobalDeclarationInstantiation as described in 15.2.8 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 env 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.

15.2.8 Runtime Semantics: GlobalDeclarationInstantiation

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.

GlobalDeclarationInstantiation 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 a `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` as the argument.
            b. If `vnDefinable` is `false`, throw a `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. Let `status` be the result of calling `env`’s `CreateGlobalFunctionBinding` concrete method passing `fn`, `fo`, and `deletableBindings` as the arguments.
   d. Return(AbruptStatus).
14. For each `String vn in declaredVarNames`, in list order do
   a. Let `status` be the result of calling `env`’s `CreateGlobalVarBinding` concrete method passing `vn` and `deletableBindings` as the argument.
   b. Return(AbruptStatus).
15. Let `lexDeclarations` be the `LexicallyScopedDeclarations of script`.
16. For each element `d` in `lexDeclarations` do
   a. `NOTE` Except for generator function declarations, lexically declared names are only instantiated here but not initialised.
   b. For each element `dn` of the `BoundNames of d` do
      i. If `IsConstantDeclaration of d` is `true`, then
         1. Let `status` be the result of calling `env`’s `CreateImmutableBinding` concrete method passing `dn` as the argument.
ii. Else,
   1. Let status be the result of calling env’s CreateMutableBinding concrete method passing dn and false as the arguments.
iii. Assert: status is never an abrupt completion for lexically declared names.
c. If d is a GeneratorDeclaration production, then
   i. Let fn be the sole element of the BoundNames of d.
   ii. Let fo be the result of performing InstantiateFunctionObject for d with argument env.
   iii. Let status be the result of calling env’s SetMutableBinding concrete method passing fn, fo, and false as the arguments.
   iv. ReturnIfAbrupt(status).

17. Return NormalCompletion(empty)

NOTE Early errors specified in 15.2.1 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 GlobalDeclarationInstantiation. 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.

15.3 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.

16 Error Handling and Language Extensions

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 as an early error any instance of an early error that is specified in a static

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.
17 ECMAScript Standard Built-in Objects

There are certain built-in objects available whenever an ECMAScript script 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 or indirectly as properties of accessible built-in objects.

Unless specified otherwise, a built-in object that is callable as a function is a Built-in Function object with the characteristics described in 9.3. Unless specified otherwise, the [[Extensible]] internal slot of a built-in object initially has the value true. Every built-in object has a [[Realm]] internal slot whose value is the code Realm for which the object was initially created.

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 built-in function or constructor 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 built-in function or constructor described 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.

Unless otherwise specified every built-in function and every built-in constructor has the Function prototype object, which is the initial value of the expression `Function.prototype` (19.2.3), as the value of its [[Prototype]] internal slot.

Unless otherwise specified every built-in prototype object has the Object prototype object, which is the initial value of the expression `Object.prototype` (19.1.3), as the value of its [[Prototype]] internal slot, except the Object prototype object itself.

Built-in function objects that are not identified as constructors do not implement the [[Construct]] internal method unless otherwise specified in the description of a particular function.

Unless otherwise specified, every built-in function defined in clauses 18 through 26 are created as if by calling the CreateBuiltinFunction abstract operation (9.3.1).

Every built-in Function object, including constructors, 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.

Unless otherwise specified, the length property of a built-in Function object has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }. 
Every built-in Function object, including constructors, that is not identified as an anonymous function has a name property whose value is a String. Unless otherwise specified, this value is the name that is given to the function in this specification. For functions that are specified as properties of objects, the name value is the property name string used to access the function. Functions that are specified as get or set accessor functions of built-in properties have "get" or "set" prepended to the property name string. The value of the name property is explicitly specified for each built-in functions whose property key is a symbols value.

Unless otherwise specified, the name property of a built-in Function object has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

Every other data property described in clauses 18 through 26 has the attributes { [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true } unless otherwise specified.

Every accessor property described in clauses 18 through 26 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 function is described, the get accessor function is the default value, undefined.

18 The Global Object

The unique global object is created before control enters any execution context.

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 slot 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.

18.1 Value Properties of the Global Object

18.1.1 Infinity

The value of Infinity is ±∞ (see 6.1.6). This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

18.1.2 NaN

The value of NaN is NaN (see 6.1.5.1). This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

18.1.3 undefined

The value of undefined is undefined (see 6.1.1). This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.
18.2 Function Properties of the Global Object

18.2.1 eval (x)

When the eval function is called with one argument x, the following steps are taken:

1. If Type(x) is not String, return x.
2. Let script be the ECMAScript code that is the result of parsing x, interpreted as UTF-16 encoded Unicode text as described in 10.1.1, for the goal symbol Script. If the parse fails or any early errors are detected, throw a SyntaxError exception (but see also clause 16).
3. If script Contains ScriptBody is false, return undefined.
4. Let strictScript be IsStrict of script.
5. If this is a direct call to eval (18.2.1.1), let direct be true, otherwise let direct be false.
6. 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.
7. 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.
8. Let evalRealm be ctx’s Realm.
9. If direct is false and strictScript is false, then
   a. Return the result of Script Evaluation for script with arguments evalRealm and true.
10. 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.
11. If direct is true, then
    a. If the code that made the direct call to eval is function code and ValidInFunction 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.
12. If direct is true, then
    a. Let lexEnv be ctx’s LexicalEnvironment.
    b. Let varEnv be ctx’s VariableEnvironment.
13. Else,
    a. Let lexEnv be evalRealm.([globalEnv]).
    b. Let varEnv be evalRealm.([globalEnv]).
14. If strictScript is true or if direct is true and strictCaller is true, then
    a. Let strictVarEnv be the result of calling NewDeclarativeEnvironment passing lexEnv as the argument.
    b. Let lexEnv be strictVarEnv.
    c. Let varEnv be strictVarEnv.
15. Let status be the result of performing Eval Declaration Instantiation as described in 18.2.1.2 with script, varEnv, and lexEnv.
16. ReturnAbrupt(status).
17. Let evalCtx be a new ECMAScript code execution context.
18. Set the evalCtx’s Realm to evalRealm.
19. Set the evalCtx’s VariableEnvironment to varEnv.
20. Set the evalCtx’s LexicalEnvironment to lexEnv.
21. If there is a currently running execution context, suspend it.
22. Push evalCtx on to the execution context stack; evalCtx is now the running execution context.
23. Let result be the result of evaluating script.
24. Resume evalCtx and remove it from the execution context stack.
25. Resume the context that is now on the top of the execution context stack as the running execution context.
26. 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.
18.2.1.1 Direct Call to Eval

A direct call to the eval function is one that is expressed as a `CallExpression` that meets all of the following 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".
- If the base value of the Reference has `true` as its `withEnvironment` value, then its binding object is an object that uses the ordinary definition of the `[[Call]]` internal method (9.3.1)
- The result of calling the abstract operation `GetValue` with that Reference as the argument is the standard built-in function defined in 18.2.1.

18.2.1.2 Eval Declaration Instantiation

18.2.2 `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 `num` is `NaN`, `+∞`, or `-∞`, return `false`.
4. Otherwise, return `true`.

18.2.3 `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`.

18.2.4 `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(`inputString`).
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 7.1.3.1), 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.
18.2.5 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, 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.

18.2.6 URI Handling Function

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 18.2.6.2, 18.2.6.3, 18.2.6.4 and 18.2.6.4.

Example: Many implementations of ECMAScript provide additional functions and methods that manipulate web pages; these functions are beyond the scope of this standard.

18.2.6.1 URI Syntax and Semantics

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".

18.2.6.1.1 Runtime Semantics: Encode Abstract Operation

The encoding and escaping process is described by the abstract operation Encode taking two String arguments string and unescapedSet.
Let $strLen$ be the number of characters in $string$.
Let $R$ be the empty String.
Let $k$ be 0.
Repeat
  a. If $k$ equals $strLen$, return $R$.
  b. Let $C$ be the character at position $k$ within $string$.
     i. If $C$ is not in $unescapedSet$, then
        1. Let $S$ be the string containing only the character $C$.
        2. Let $R$ be a new string value computed by concatenating the previous value of $R$ and $S$.
     ii. Else $C$ is not in $unescapedSet$,
        1. If the code unit value of $C$ is not less than 0xDC00 and not greater than 0xDFFF, throw a $URIError$ exception.
        2. If the code unit value of $C$ is less than 0xD800 or greater than 0xDBFF, then
           a. Let $V$ be the code unit value of $C$.
           i. Increase $k$ by 1.
           ii. If $k$ equals $strLen$, throw a $URIError$ exception.
           iii. Let $kChar$ be the code unit value of the character at position $k$ within $string$.
           iv. If $kChar$ is less than 0xDC00 or greater than 0xDFFF, throw a $URIError$ exception.
           v. Let $V$ be (((the code unit value of $C$) – 0xD800) * 0x400 + ($kChar$ – 0xDC00) + 0x10000).
           vi. Let $Octets$ be the array of octets resulting by applying the UTF-8 transformation to $V$, and let $L$ be the array size.
           vii. Let $j$ be 0.
           viii. 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.
              5. Increase $k$ by 1.

18.2.6.1.2 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 $string$.
      i. If $C$ is not ‘%’, then
         1. Let $S$ be the string containing only the character $C$.
      ii. Else $C$ is ‘%’,
         1. Let $start$ be $k$.
         2. If $k + 2$ is greater than or equal to $strLen$, throw a $URIError$ exception.
         3. If the characters at position ($k$+1) and ($k$ + 2) within $string$ do not represent hexadecimal digits, throw a $URIError$ exception.
         4. Let $B$ be the 8-bit value represented by the two hexadecimal digits at position ($k$ + 1) and ($k$ + 2).
         5. Increment $k$ by 2.
         6. 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
               1. Let $S$ be the string containing only the character $C$.
               2. Else $C$ is in $reservedSet$,
                  a. Let $S$ be the substring of $string$ from position $start$ to position $k$ included.
         7. Else the most significant bit in $B$ is 1,
            1. Let $n$ be the smallest non-negative number such that ($B << n$) & 0x80 is equal to 0.
2. If \( n \) equals 1 or \( n \) is greater than 4, throw a URIError exception.
3. Let Octets be an array of 8-bit integers of size \( n \).
4. Put \( B \) into Octets at position 0.
5. If \( k + (3 \times (n - 1)) \) is greater than or equal to \( \text{strLen} \), 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 \( \text{string} \) is not "%", throw a URIError exception.
   c. If the characters at position \( (k + 1) \) and \( (k + 2) \) within \( \text{string} \) 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 a URIError exception.
9. If \( V < 0x10000 \), then:
   a. Let \( C \) be the character with code unit value \( V \).
   b. If \( C \) is not in \( \text{reservedSet} \), then:
      i. Let \( S \) be the String containing only the character \( C \).
   c. Else \( C \) is in \( \text{reservedSet} \),
      i. Let \( S \) be the substring of \( \text{string} \) from position \( \text{start} \) to position \( k \) included.
10. Else \( V \geq 0x10000 \),
    a. Let \( L \) be \( ((V - 0x10000) \& 0x3FF) + 0xDC00) \).
    b. Let \( H \) be \( (((V - 0x10000) >> 10) \& 0x3FF) + 0xD800) \).
    c. Let \( S \) be the String containing the two characters with code unit values \( H \) and \( L \).
    e. Let \( R \) be a new String value computed by concatenating the previous value of \( R \) and \( S \).
    f. Increase \( k \) by 1.

**NOTE**
This syntax of Uniform Resource Identifiers is based upon RFC 2396 and does not reflect the more recent RFC 3986 which replaces RFC 2396. A formal description and implementation of UTF-8 is given in RFC 3629.

In UTF-8, characters are encoded using sequences of 1 to 6 octets. The only octet of a "sequence" of one has the higher-order bit set to 0, the remaining 7 bits being used to encode the character value. In a sequence of \( n \) octets, \( n = 1 \), 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 31.
Table 31 — UTF-8 Encodings

<table>
<thead>
<tr>
<th>Code Unit Value</th>
<th>Representation</th>
<th>1st Octet</th>
<th>2nd Octet</th>
<th>3rd Octet</th>
<th>4th Octet</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 – 0x007F</td>
<td>00000000 0zzzzzz</td>
<td>0zzzzzz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0080 – 0x07FF</td>
<td>0000yyyy yyyyyy 1111xxxx 10yyyyyy 10zzzzzz</td>
<td>110yyyyy 10zzzzzz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0800 – 0x0FFF</td>
<td>Followed by</td>
<td>110111yyyy yyyyyyy</td>
<td>11110uuu 10yyyyyy 10zzzzzz</td>
<td>110111yyyy yyyyyyy</td>
<td>11110uuu 10yyyyyy 10zzzzzz</td>
</tr>
<tr>
<td>0x0800 – 0x0FFF</td>
<td>not Followed by</td>
<td>causes URIError</td>
<td>11110uuu 10yyyyyy 10zzzzzz</td>
<td>11110uuu 10yyyyyy 10zzzzzz</td>
<td></td>
</tr>
<tr>
<td>0x8000 – 0xFFFF</td>
<td>causes URIError</td>
<td>11110uuu 10yyyyyy 10zzzzzz</td>
<td>11110uuu 10yyyyyy 10zzzzzz</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Where

\[
n_{uuuu} = vvvv + 1
\]

to account for the addition of 0x10000 as in Surrogates, section 3.7, of the Unicode Standard.

The range of code unit values 0xD800-0xDBFF 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.

18.2.6.2 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.

18.2.6.3 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)
18.2.6.4 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.

18.2.6.5 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)

18.3 Constructor Properties of the Global Object

18.3.1 Array (...)
See 22.1.1.

18.3.2 ArrayBuffer (...)
See 24.1.2.

18.3.3 Boolean (...)
See 19.3.1.

18.3.4 DataView (...)
See 24.2.2.

18.3.5 Date (...)
See 20.3.2.
18.3.6 Error ( . . . )
See 19.5.1.

18.3.7 EvalError ( . . . )
See 19.5.5.1.

18.3.8 Float32Array ( . . . )
See 22.2.4.

18.3.9 Float64Array ( . . . )
See 22.2.4.

18.3.10 Function ( . . . )
See 19.2.1.

18.3.11 Int8Array ( . . . )
See 22.2.4.

18.3.12 Int16Array ( . . . )
See 22.2.4.

18.3.13 Int32Array ( . . . )
See 22.2.4.

18.3.14 Map ( . . . )
See 23.1.1.

18.3.15 Number ( . . . )
See 20.1.1.

18.3.16 Object ( . . . )
See 19.1.1.

18.3.17 RangeError ( . . . )
See 19.5.5.2.

18.3.18 ReferenceError ( . . . )
See 19.5.5.3.

18.3.19 RegExp ( . . . )
See 21.2.4.
18.3.20 Set ( . . . )
See 23.2.1.

18.3.21 String ( . . . )
See 21.1.1.

18.3.22 Symbol ( . . . )
See 19.4.1.

18.3.23 SyntaxError ( . . . )
See 19.5.5.4.

18.3.24 TypeError ( . . . )
See 19.5.5.5.

18.3.25 Uint8Array ( . . . )
See 22.2.4.

18.3.26 Uint8ClampedArray ( . . . )
See 22.2.4.

18.3.27 Uint16Array ( . . . )
See 22.2.4.

18.3.28 Uint32Array ( . . . )
See 22.2.4.

18.3.29 URIError ( . . . )
See 19.5.5.6.

18.3.30 WeakMap ( . . . )
See 23.3.1.

18.3.31 WeakSet ( . . . )
See 23.4.

18.4 Other Properties of the Global Object

18.4.1 JSON
See 24.3.
18.4.2 Math
See 20.2.

18.4.3 Proxy(...)
See 26.2.1.

18.4.4 Reflect
See 26.1.

19 Fundamental Objects

19.1 Object Objects

19.1.1 The Object Constructor

The Object constructor is the `%Object% intrinsic object and the initial value of the `object` property of the global object. When `Object` is called as a function rather than as a constructor, it performs a type conversion.

The `Object` constructor is designed to be subclassable. It may be used as the value of an `extends` clause of a class declaration.

NOTE Subclass constructors that inherit from the Object constructor typically should not include a `super` call to `Object` as it performs no initialization action on its `this` value and does not return its `this` value as its value.

19.1.1.1 `Object([value])`

When `Object` function is called with optional argument `value`, the following steps are taken:

1. If `value` is `null`, `undefined` or not supplied, return the result of the abstract operation `ObjectCreate` with the intrinsic object `%ObjectPrototype%` as its argument.
2. Return `ToObject(value)`.

19.1.1.2 `new Object(...argumentsList)`

When `Object` is called as part of a `new` expression, it creates a new object:

1. Let `F` be the `Object` function object on which the `new` operator was applied.
2. Let `argumentsList` be the `argumentsList` argument of the `[[Construct]]` internal method that was invoked by the `new` operator.
3. Return the result of calling the `[[Call]]` internal method of `F`, providing `undefined` and `argumentsList` as the arguments.

The above steps defined the `[[Construct]]` internal method of the Object constructor. Object may not be implemented as an ECMAScript function object because this definition differs from the definition of `[[Construct]]` used by ECMAScript function objects.

19.1.2 Properties of the Object Constructor

The value of the `[[Prototype]]` internal slot 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:
19.1.2.1 Object.assign (target, source)

The assign function is used to copy the values of all of the enumerable own properties from a source object to a target object. When the assign function is called, the following steps are taken:

1. Let to be ToObject(target).
2. ReturnIfAbrupt(to).
3. Let from be ToObject(source).
4. ReturnIfAbrupt(from).
5. Let keys be the result of calling the [[OwnPropertyKeys]] internal method of source.
6. ReturnIfAbrupt(keys).
7. Let getAllNames be false.
8. Let pendingException be undefined.
9. Repeat while getAllNames is false,
   a. Let next be the result of IteratorStep(keys).
   b. ReturnIfAbrupt(next).
   c. If next is false, then let getAllNames be true.
   d. Else,
      i. Let nextKey be IteratorValue(next).
      ii. ReturnIfAbrupt(nextKey).
      iii. Let desc be the result of calling the [[GetOwnProperty]] internal method of from with argument nextKey.
      iv. If desc is an abrupt completion, then
         1. If pendingException is undefined, then set pendingException to desc.
      v. Else if desc is not undefined and desc.[[Enumerable]] is true, then
         1. Let propValue be Get(from, nextKey).
         2. If propValue is an abrupt completion, then
            a. If pendingException is undefined, then set pendingException to propValue.
         3. Else
            a. Let status be Put(to, nextKey, propValue, true);
            b. If status is an abrupt completion, then
               i. If pendingException is undefined, then set pendingException to status.
   10. If pendingException is not undefined, then return pendingException.
11. Return to.

19.1.2.2 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(obj, Properties).
4. Return obj.

19.1.2.3 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.
2. 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 a List containing the keys of each enumerable own property of `props`.
4. Let `descriptors` be an empty 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. ReturnIfAbrupt(`pendingException`).
7. Return `O`.

If an implementation defines a specific order of enumeration for the `for-in` statement, that same enumeration order must be used to order the list elements in step 3 of this algorithm.

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.

19.1.2.4 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 `key` be `ToPropertyKey(P)`.
3. ReturnIfAbrupt(`key`).
4. Let `desc` be the result of calling `ToPropertyDescriptor(Attributes)`.
5. ReturnIfAbrupt(`desc`).
6. Let `success` be the result of `DefinePropertyOrThrow(O, key, desc)`.
7. ReturnIfAbrupt(`success`).
8. Return `O`.

19.1.2.5 Object.freeze ( O )

When the `freeze` function is called, the following steps are taken:

1. If `Type(O)` is not `Object`, return `O`.
2. Let `status` be the result of `SetIntegrityLevel(O, "frozen")`.
3. ReturnIfAbrupt(`status`).
4. If `status` is false, throw a `TypeError` exception.
5. Return `O`.

19.1.2.6 Object.getOwnPropertyDescriptor ( O, P )

When the `getOwnPropertyDescriptor` function is called, the following steps are taken:

1. Let `obj` be `ToObject(O)`.
2. ReturnIfAbrupt(`obj`).
3. Let `key` be `ToPropertyKey(P)`.
4. ReturnIfAbrupt(`key`).
5. Let `desc` be the result of calling the `[GetOwnProperty]` internal method of `obj` with argument `key`.
6. ReturnIfAbrupt(`desc`).
7. Return the result of calling `FromPropertyDescriptor(desc)`.

19.1.2.7 **Object.getOwnPropertyNames ( O )**

When the `getOwnPropertyNames` function is called, the following steps are taken:

1. Return `GetOwnPropertyKeys(O, String)`.

19.1.2.8 **Object.getOwnPropertySymbols ( O )**

When the `getOwnPropertySymbols` function is called with argument `O`, the following steps are taken:

1. Return `GetOwnPropertyKeys(O, Symbol)`.

19.1.2.8.1 **GetOwnPropertyKeys ( O, Type ) Abstract Operation**

The abstract operation `GetOwnPropertyKeys` is called with arguments `O` and `Type` where `O` is an Object and `Type` is one of the ECMAScript specification types String or Symbol. The following steps are taken:

1. Let `obj` be `ToObject(O)`.
2. ReturnIfAbrupt(obj).
3. Let `keys` be the result of calling the `[[OwnPropertyKeys]]` internal method of `obj`.
4. ReturnIfAbrupt(keys).
5. Let `nameList` be a new empty List.
6. Let `gotAllNames` be `false`.
7. Repeat while `gotAllNames` is `false`:
   a. Let `next` be the result of `IteratorStep(keys)`.
   b. ReturnIfAbrupt(next).
   c. If `next` is `false`, then let `gotAllNames` be `true`.
   d. Else:
      i. Let `nextKey` be `IteratorValue(next)`.
      ii. ReturnIfAbrupt(nextKey).
      iii. If `Type(nextKey)` is `Type`, then
          1. Append `nextKey` as the last element of `nameList`.
8. Return `CreateArrayFromList(nameList)`.

19.1.2.9 **Object.getPrototypeOf ( O )**

When the `getPrototypeOf` function is called with argument `O`, the following steps are taken:

1. Let `obj` be `ToObject(O)`.
2. ReturnIfAbrupt(obj).
3. Return the result of calling the `[[GetPrototypeOf]]` internal method of `obj`.

19.1.2.10 **Object.is ( value1, value2 )**

When the `is` function is called with arguments `value1` and `value2` the following steps are taken:

1. Return `SameValue(value1, value2)`.

19.1.2.11 **Object.isExtensible ( O )**

When the `isExtensible` function is called with argument `O`, the following steps are taken:

1. If `Type(O)` is not Object, return `false`.
2. Return the result of `IsExtensible(O)`.
19.1.2.12 Object.isFrozen ( O )

When the `isFrozen` function is called with argument `O`, the following steps are taken:

1. If `Type(O)` is not Object, return `true`.
2. Return `TestIntegrityLevel(O, "frozen")`.

19.1.2.13 Object.isSealed ( O )

When the `isSealed` function is called with argument `O`, the following steps are taken:

1. If `Type(O)` is not Object, return `true`.
2. Return `TestIntegrityLevel(O, "sealed")`.

19.1.2.14 Object.keys ( O )

When the `keys` function is called with argument `O`, the following steps are taken:

1. Let `obj` be `ToObject(O)`.
2. ReturnIfAbrupt(`obj`).
3. Let `keys` be the result of calling the `[[OwnPropertyKeys]]` internal method of `obj`.
4. ReturnIfAbrupt(`keys`).
5. Let `nameList` be a new empty List.
6. Let `gotAllNames` be `false`.
7. Repeat while `gotAllNames` is `false`.
   a. Let `next` be the result of `IteratorStep(keys)`.
   b. ReturnIfAbrupt(`next`).
   c. If `next` is `false`, then let `gotAllNames` be `true`.
   d. Else,
      i. Let `nextKey` be `IteratorValue(next)`.
      ii. ReturnIfAbrupt(`nextKey`).
      iii. If `Type(nextKey)` is String, then
          1. Let `desc` be the result of calling the `[[GetProperty]]` internal method of `O` with argument `nextKey`.
          2. ReturnIfAbrupt(`desc`).
          3. If `desc` is not `undefined` and `desc.[[Enumerable]]` is `true`, then
             a. Append `nextKey` as the last element of `nameList`.
     8. Return `CreateArrayFromList(nameList)`.

If an implementation defines a specific order of enumeration for the for-in statement, the same order must be used for the elements of the array returned in step 8.

19.1.2.15 Object.mixin ( target, source )

The `mixin` function is used to copy the definitions of all of the own properties from a `source` object to a `target` object. When the `mixin` function is called, the following steps are taken:

1. Let `to` be `ToObject(target)`.
2. ReturnIfAbrupt(`to`).
3. Let `from` be `ToObject(source)`.
4. ReturnIfAbrupt(`from`).
5. Return the result of `MixinProperties(to, from)`.

19.1.2.15.1 MixinProperties( target, source )

The abstract operation `MixinProperties` copy the definitions of all of the own properties from a `source` object to a `target` object. If a property value is a function that is super bound to `source`, a new function is created that is super bound to `target` and the MixinProperties is recursively applied to the functions. When the MixinProperties abstract operations called, the following steps are taken.
1. Assert: Type(target) is Object.
2. Assert: Type(source) is Object.
3. Let keys be the result of calling the [[OwnPropertyKeys]] internal method of source.
4. ReturnIfAbrupt(keys).
5. Let gotAllNames be false.
6. Let pendingException be undefined.
7. Repeat while gotAllNames is false,
   a. Let next be the result of IteratorStep(keys).
   b. ReturnIfAbrupt(next).
   c. If next is false, then let gotAllNames be true.
   d. Else,
      i. Let nextKey be IteratorValue(next).
      ii. ReturnIfAbrupt(nextKey).
      iii. If nextKey is false, then let gotAllNames be true.
      iv. Else,
         i. Let nextKey be IteratorValue(next).
         ii. ReturnIfAbrupt(nextKey).
         iii. Let desc be the result of calling the [[GetOwnProperty]] internal method of source with argument nextKey.
         iv. If desc is an abrupt completion, then
            1. If pendingException is undefined, then set pendingException to desc.
            2. Else if desc is not undefined, then
               i. If IsDataDescriptor(desc), then
                  a. Let propValue be desc.[[Value]].
                  b. If SameValue(GetSuperBinding(propValue, source), then
                     i. Let newFunc be MixinProperties(RebindSuper(propValue, target), propValue).
                     ii. If newFunc is an abrupt completion, then
                        1. If pendingException is undefined, then set pendingException to newFunc.
                        2. Else, set desc.[[Value]] to newFunc.
                  i. Else, desc is an accessor descriptor
                     a. Let getter be desc.[[Get]].
                     b. If SameValue(GetSuperBinding(getter, source), then
                        i. Let newFunc be MixinProperties(RebindSuper(getter, target), getter).
                        ii. If newFunc is an abrupt completion, then
                           1. If pendingException is undefined, then set pendingException to newFunc.
                           2. Else, set desc.[[Get]] to newFunc.
                     i. Else, desc is not a well-formed accessor descriptor
                        a. Let setter be desc.[[Set]].
                        b. If SameValue(GetSuperBinding(setter, source), then
                           i. Let newFunc be MixinProperties(RebindSuper(setter, target), setter).
                           ii. If newFunc is an abrupt completion, then
                              1. If pendingException is undefined, then set pendingException to newFunc.
                              2. Else, set desc.[[Set]] to newFunc.
                        i. Else, desc is not a well-formed accessor descriptor
                           a. Let status be DefinePropertyOrThrow(target, nextKey, desc).
                           b. If status is an abrupt completion, then
                              i. If pendingException is not undefined, then set pendingException to status.
         i. Else, desc is an abrupt completion, then
            1. Let status be DefinePropertyOrThrow(target, nextKey, desc).
            2. If status is an abrupt completion, then
               i. If pendingException is undefined, then set pendingException to status.
     i. Else if desc is not undefined, then
        1. If IsDataDescriptor(desc), then
           a. Let propValue be desc.[[Value]].
           b. If SameValue(GetSuperBinding(propValue, source), then
              i. Let newFunc be MixinProperties(RebindSuper(propValue, target), propValue).
              ii. If newFunc is an abrupt completion, then
                 1. If pendingException is undefined, then set pendingException to newFunc.
                 2. Else, set desc.[[Value]] to newFunc.
           i. Else, desc is an accessor descriptor
              a. Let getter be desc.[[Get]].
              b. If SameValue(GetSuperBinding(getter, source), then
                 i. Let newFunc be MixinProperties(RebindSuper(getter, target), getter).
                 ii. If newFunc is an abrupt completion, then
                    1. If pendingException is undefined, then set pendingException to newFunc.
                    2. Else, set desc.[[Get]] to newFunc.
              i. Else, desc is not a well-formed accessor descriptor
                 a. Let setter be desc.[[Set]].
                 b. If SameValue(GetSuperBinding(setter, source), then
                    i. Let newFunc be MixinProperties(RebindSuper(setter, target), setter).
                    ii. If newFunc is an abrupt completion, then
                       1. If pendingException is undefined, then set pendingException to newFunc.
                       2. Else, set desc.[[Set]] to newFunc.
             i. Else, desc is an abrupt completion, then
                1. If pendingException is not undefined, then set pendingException to status.
         i. Else if desc is not undefined, then
            1. If IsDataDescriptor(desc), then
               a. Let propValue be desc.[[Value]].
               b. If SameValue(GetSuperBinding(propValue, source), then
                  i. Let newFunc be MixinProperties(RebindSuper(propValue, target), propValue).
                  ii. If newFunc is an abrupt completion, then
                     1. If pendingException is undefined, then set pendingException to newFunc.
                     2. Else, set desc.[[Value]] to newFunc.
               i. Else, desc is an accessor descriptor
                  a. Let getter be desc.[[Get]].
                  b. If SameValue(GetSuperBinding(getter, source), then
                     i. Let newFunc be MixinProperties(RebindSuper(getter, target), getter).
                     ii. If newFunc is an abrupt completion, then
                        1. If pendingException is undefined, then set pendingException to newFunc.
                        2. Else, set desc.[[Get]] to newFunc.
                  i. Else, desc is not a well-formed accessor descriptor
                     a. Let setter be desc.[[Set]].
                     b. If SameValue(GetSuperBinding(setter, source), then
                        i. Let newFunc be MixinProperties(RebindSuper(setter, target), setter).
                        ii. If newFunc is an abrupt completion, then
                           1. If pendingException is undefined, then set pendingException to newFunc.
                           2. Else, set desc.[[Set]] to newFunc.
             i. Else, desc is an abrupt completion, then
                1. If pendingException is not undefined, then set pendingException to status.
     i. Else, desc is an abrupt completion, then
        1. If pendingException is undefined, then set pendingException to desc.
4. Else if desc is not undefined, then
   1. If IsDataDescriptor(desc), then
      a. Let propValue be desc.[[Value]].
      b. If SameValue(GetSuperBinding(propValue, source), then
         i. Let newFunc be MixinProperties(RebindSuper(propValue, target), propValue).
         ii. If newFunc is an abrupt completion, then
            1. If pendingException is undefined, then set pendingException to newFunc.
            2. Else, set desc.[[Value]] to newFunc.
      i. Else, desc is an accessor descriptor
         a. Let getter be desc.[[Get]].
         b. If SameValue(GetSuperBinding(getter, source), then
            i. Let newFunc be MixinProperties(RebindSuper(getter, target), getter).
            ii. If newFunc is an abrupt completion, then
               1. If pendingException is undefined, then set pendingException to newFunc.
               2. Else, set desc.[[Get]] to newFunc.
         i. Else, desc is not a well-formed accessor descriptor
            a. Let setter be desc.[[Set]].
            b. If SameValue(GetSuperBinding(setter, source), then
               i. Let newFunc be MixinProperties(RebindSuper(setter, target), setter).
               ii. If newFunc is an abrupt completion, then
                  1. If pendingException is undefined, then set pendingException to newFunc.
                  2. Else, set desc.[[Set]] to newFunc.
         i. Else, desc is not a well-formed accessor descriptor
            a. Let status be DefinePropertyOrThrow(target, nextKey, desc).
            b. If status is an abrupt completion, then
               i. If pendingException is not undefined, then set pendingException to status.
        i. Else if desc is not undefined, then
           1. If IsDataDescriptor(desc), then
              a. Let propValue be desc.[[Value]].
              b. If SameValue(GetSuperBinding(propValue, source), then
                 i. Let newFunc be MixinProperties(RebindSuper(propValue, target), propValue).
                 ii. If newFunc is an abrupt completion, then
                    1. If pendingException is undefined, then set pendingException to newFunc.
                    2. Else, set desc.[[Value]] to newFunc.
              i. Else, desc is an accessor descriptor
                 a. Let getter be desc.[[Get]].
                 b. If SameValue(GetSuperBinding(getter, source), then
                    i. Let newFunc be MixinProperties(RebindSuper(getter, target), getter).
                    ii. If newFunc is an abrupt completion, then
                       1. If pendingException is undefined, then set pendingException to newFunc.
                       2. Else, set desc.[[Get]] to newFunc.
                 i. Else, desc is not a well-formed accessor descriptor
                    a. Let setter be desc.[[Set]].
                    b. If SameValue(GetSuperBinding(setter, source), then
                       i. Let newFunc be MixinProperties(RebindSuper(setter, target), setter).
                       ii. If newFunc is an abrupt completion, then
                          1. If pendingException is undefined, then set pendingException to newFunc.
                          2. Else, set desc.[[Set]] to newFunc.
                 i. Else, desc is not a well-formed accessor descriptor
                    a. Let status be DefinePropertyOrThrow(target, nextKey, desc).
                    b. If status is an abrupt completion, then
                       i. If pendingException is not undefined, then set pendingException to status.
           i. Else, desc is an abrupt completion, then
              1. If pendingException is undefined, then set pendingException to desc.
5. If pendingException is not undefined, then return pendingException.
6. Return target.

19.1.2.16 Object.preventExtensions ( O )

When the preventExtensions function is called, the following steps are taken:

1. If Type(O) is not Object, return O.
2. Let status be the result of calling the [[PreventExtensions]] internal method of O.
3. ReturnIfAbrupt(status).
4. If status is false, throw a TypeError exception.
5. Return O.

19.1.2.17 Object.prototype

The initial value of Object.prototype is the standard built-in Object prototype object (19.1.3).

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

19.1.2.18 Object.seal ( O )

When the `seal` function is called, the following steps are taken:

1. If `Type(O)` is not Object, return O.
2. Let `status` be the result of `SetIntegrityLevel(O, "sealed")`.
3. ReturnIfAbrupt(status).
4. If `status` is false, throw a `TypeError` exception.
5. Return O.

19.1.2.19 Object.setPrototypeOf ( O, proto )

When the `setPrototypeOf` function is called with arguments `O` and `proto`, the following steps are taken:

1. Let `O` be `CheckObjectCoercible(O)`.
2. ReturnIfAbrupt(O).
3. If `Type(proto)` is neither Object nor Null, then throw a `TypeError` exception.
4. If `Type(O)` is not Object, then return O.
5. Let `status` be the result of calling the `[[SetPrototypeOf]]` internal method of `O` with argument `proto`.
6. ReturnIfAbrupt(status).
7. If `status` is false, then throw a `TypeError` exception.
8. Return O.

19.1.3 Properties of the Object Prototype Object

The Object prototype object is an ordinary object.

The value of the `[[Prototype]]` internal slot of the Object prototype object is `null` and the initial value of the `[[Extensible]]` internal slot is `true`.

19.1.3.1 Object.prototype.constructor

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

19.1.3.2 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.
   a. ReturnIfAbrupt(O).
4. Return the result of `HasOwnProperty(O, P)`.

**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`.

19.1.3.3 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 `[[GetPrototypeOf]]` internal method of `V` with no arguments.
   b. if `V` is `null`, return `false`.
   c. If `SameValue(O, V)` is `true`, then return `true`.

NOTE The ordering of steps 1 and 2 preserves 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`.

19.1.3.4 `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`.

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`.

19.1.3.5 `Object.prototype.toLocaleString ( )`

When the `toLocaleString` method is called, the following steps are taken:

1. Let `O` be the `this` value.
2. 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.

19.1.3.6 `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 `Array` object, then let `builtinTag` be "Array".
5. Else, if `O` is an exotic `String` object, then let `builtinTag` be "String".
6. Else, if `O` is an exotic Proxy object, then let `builtinTag` be "Proxy".
7. Else, if `O` is an exotic arguments object, then let `builtinTag` be "Arguments".
8. Else, if `O` is an ECMAScript function object, a built-in function object, or a bound function exotic object, then let `builtinTag` be "Function".
9. Else, if `O` has an `[[ErrorData]]` internal slot, then let `builtinTag` be "Error".
10. Else, if \( O \) has a [[BooleanData]] internal slot, then let \( \text{builtinTag} \) be "Boolean".
11. Else, if \( O \) has a [[NumberData]] internal slot, then let \( \text{builtinTag} \) be "Number".
12. Else, if \( O \) has a [[DateValue]] internal slot, then let \( \text{builtinTag} \) be "Date".
13. Else, if \( O \) has a [[RegExpMatcher]] internal slot, then let \( \text{builtinTag} \) be "RegExp".
14. Else, let \( \text{builtinTag} \) be "Object".
15. Let \( \text{hasTag} \) be the result of HasProperty(\( O \), @toStringTag).
16. ReturnIfAbrupt(\( \text{hasTag} \)).
17. If \( \text{hasTag} \) is false, then let \( \text{tag} \) be \( \text{builtinTag} \).
18. Else,
   a. Let \( \text{tag} \) be the result of Get(\( O \), @toStringTag).
   b. If \( \text{tag} \) is an abrupt completion, let \( \text{tag} \) be NormalCompletion("???").
   c. Let \( \text{tag} \) be \( \text{tag} \)\[value\].
   d. If Type(\( \text{tag} \)) is not String, let \( \text{tag} \) be "???".
   e. If \( \text{tag} \) is any of "Arguments", "Array", "Boolean", "Date", "Error", "Function", "Number", "RegExp", or "String" and SameValue(\( \text{tag} \), \( \text{builtinTag} \)) is false, then let \( \text{tag} \) be the string value "~" concatenated with the current value of \( \text{tag} \).
19. Return the String value that is the result of concatenating the three Strings "[object ", \( \text{tag} \), and "]".

NOTE
Historically, this function was occasionally used to access the string value of the [[Class]] internal slot that was used in previous editions of this specification as a nominal type tag for various built-in objects. The above 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.

19.1.3.7 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 \( O \).

19.4 Properties of Object Instances

Object instances have no special properties beyond those inherited from the Object prototype object.

19.2 Function Objects

19.2.1 The Function Constructor

The Function constructor is the %Function% intrinsic object and the initial value of the Function property of the global object. 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. However, if the this value passed in the call is an Object with a [[Code]] internal slot whose value is undefined, it initialises the this value using the argument values. This permits Function to be used both as factory method and to perform constructor instance initialisation.

Function may be subclassed and subclass constructors may perform a super invocation of the Function constructor to initialise subclass instances. However, all syntactic forms for defining function objects create instances of Function. There is no syntactic means to create instances of Function subclasses except for the built-in Generator Function subclass.

19.2.1.1 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 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:

© Ecma International 2009 – All rights reserved
1. Let \(\text{argCount}\) be the total number of arguments passed to this function invocation.
2. Let \(P\) be the empty String.
3. If \(\text{argCount} = 0\), let \(\text{bodyText}\) be the empty String.
4. Else if \(\text{argCount} = 1\), let \(\text{bodyText}\) be that argument.
5. Else \(\text{argCount} > 1\),
   a. Let \(\text{firstArg}\) be the first argument.
   b. ReturnIfAbrupt(\(\text{firstArg}\)).
   c. Let \(P\) be ToString(\(\text{firstArg}\)).
   d. ReturnIfAbrupt(\(P\)).
   e. Let \(k\) be 2.
   f. Repeat, while \(k < \text{argCount}\)
      i. Let \(\text{nextArg}\) be the \(k\)'th argument.
      ii. Let \(\text{nextArgString}\) be ToString(\(\text{nextArg}\)).
      iii. ReturnIfAbrupt(\(\text{nextArgString}\)).
      iv. Let \(P\) be the result of concatenating the previous value of \(P\), the String "," (a comma), and \(\text{nextArgString}\).
      v. Increase \(k\) by 1.
6. Let \(\text{bodyText}\) be the \(k\)'th argument.
7. Let \(\text{bodyText}\) be ToString(\(\text{bodyText}\)).
8. ReturnIfAbrupt(\(\text{bodyText}\)).
9. Let \(\text{parameters}\) be the result of parsing \(P\), interpreted as UTF-16 encoded Unicode text as described in clause 10.1.1, using FormalParameters as the goal symbol. Throw a SyntaxError exception if the parse fails.
10. Let \(\text{body}\) be the result of parsing \(\text{bodyText}\), interpreted as UTF-16 encoded Unicode text as described in clause 10.1.1, using FunctionBody as the goal symbol. Throw a SyntaxError exception if the parse fails or if any static semantics errors are detected.
11. If IsSimpleParameterList of \(\text{parameters}\) is false and any element of the BoundNames of \(\text{parameters}\) also occurs in the VarDeclaredNames of \(\text{body}\), then throw a SyntaxError exception.
12. If any element of the BoundNames of \(\text{parameters}\) also occurs in the LexicallyDeclaredNames of \(\text{body}\), then throw a SyntaxError exception.
13. If \(\text{bodyText}\) is strict mode code (see 10.2.1) then let \(\text{strict}\) be true, else let \(\text{strict}\) be false.
14. Let \(\text{scope}\) be the Global Environment.
15. Let \(F\) be the this value.
16. If Type(\(F\)) is not Object or if \(F\) does not have a \([\text{Code}]\) internal slot or if the value of \([\text{Code}]\) is not undefined, then
17. Let \(F\) be the result of calling FunctionAllocate with arguments %FunctionPrototype% and \(\text{strict}\).
18. Else, set \(F\)'s \([\text{Strict}]\) internal slot to \(\text{strict}\).
19. If the value of \(F\)'s \([\text{FunctionKind}]\) internal slot is not "normal", then throw a TypeError exception.
20. Perform the FunctionInitialise abstract operation with arguments \(F\), Normal, \(\text{parameters}\), body, and \(\text{scope}\).
21. Let \(\text{status}\) be the result of MakeConstructor \(F\).
22. SetFunctionName(\(F\), "anonymous").
23. Return \(F\).

A prototype property is automatically created for every function created using the Function constructor, 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:

\[
\text{new Function(} "a", "b", "c", "return a+b+c")
\]

\[
\text{new Function(} a, b, c, "return a+b+c")
\]

\[
\text{new Function(}a, b", c", "return a+b+c")
\]

19.2.1.2 new Function( ... argumentsList)

When Function is called as part of a new expression, it initialises the newly created object.
1. Let \( F \) be the \texttt{Function} function object on which the \texttt{new} operator was applied.
2. Let \( argumentsList \) be the \texttt{argumentsList} argument of the \texttt{[[Construct]]} internal method that was invoked by the \texttt{new} operator.
3. Return the result of \texttt{OrdinaryConstr}(\( F \), \( argumentsList \)).

If \texttt{Function} is implemented as an ECMAScript function object, its \texttt{[[Construct]]} internal method will perform the above steps.

19.2.2 Properties of the Function Constructor

The Function constructor is itself a built-in Function object. The value of the \texttt{[[Prototype]]} internal slot of the Function constructor is \%FunctionPrototype\%, the intrinsic Function prototype object (19.2.3).

The value of the \texttt{[[Extensible]]} internal slot of the Function constructor is \texttt{true}.

The Function constructor has the following properties:

19.2.2.1 \texttt{Function.length}

This is a data property with a value of 1. This property has the attributes \{ \texttt{[[Writable]]}: \texttt{false}, \texttt{[[Enumerable]]}: \texttt{false}, \texttt{[[Configurable]]}: \texttt{true} \}.

19.2.2.2 \texttt{Function.prototype}

The value of \texttt{Function.prototype} is \%FunctionPrototype\%, the intrinsic Function prototype object (19.2.3).

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

19.2.2.3 \texttt{Function[@@create]} ()

The @@create method of an object \( F \) performs the following steps:

1. Let \( F \) be the \texttt{this} value.
2. Let \( proto \) be the result of \texttt{GetPrototypeFromConstructor}(\( F \), \%FunctionPrototype\%).
3. ReturnIfAbrupt(\( proto \)).
4. Let \( obj \) be the result of calling \texttt{FunctionAllocate} with arguments \( proto \) and \texttt{false}.
5. Return \( obj \).

The value of the \texttt{name} property of this function is \"\texttt{[Symbol.create]}\"\.

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

NOTE The Function @@create function passes \texttt{false} as the \texttt{strict} parameter to FunctionAllocate. This causes the allocated ECMAScript function object to have the internal methods of a non-strict function. The Function constructor may reset the functions \texttt{[[Strict]]} internal slot to \texttt{true}. It is up to the implementation whether this also changes the internal methods.

19.2.3 Properties of the Function Prototype Object

The Function prototype object is itself a Built-in Function object. When invoked, it accepts any arguments and returns \texttt{undefined}.

NOTE The Function prototype object is specified to be a function object to ensure compatibility with ECMAScript code that was created prior to the 6th Edition of this specification.

The value of the \texttt{[[Prototype]]} internal slot of the Function prototype object is the intrinsic object \%ObjectPrototype\% (19.1.3). The initial value of the \texttt{[[Extensible]]} internal slot of the Function prototype object is \texttt{true}. 
The Function prototype object does not have a `prototype` property.

The value of the `length` property of the Function prototype object is 0.

The value of the `name` property of the Function prototype object is the empty String.

### 19.2.3.1 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. Let `argList` be the result of `CreateListFromArrayLike(argArray)`.
4. Perform the `PrepareForTailCall` abstract operation.
5. 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 an `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.

### 19.2.3.2 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) 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 `BoundFunctionCreate` with arguments `Target`, `thisArg`, and `A`.
5. If `Target` has a `[[BoundTargetFunction]]` internal slot, 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]]: true`.
8. Perform the `AddRestrictedFunctionProperties` abstract operation with argument `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.

### 19.2.3.3 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. Perform the PrepareForTailCall abstract operation.
5. 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 an 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.

19.2.3.4 Function.prototype.constructor

The initial value of Function.prototype.constructor is the intrinsic object %Function%.

19.2.3.5 Function.prototype.toString ()

An implementation-dependent String source code representation of the this object is returned. This representation has the syntax of a FunctionDeclaration FunctionExpression, GeneratorDeclaration, GeneratorExpression, ClassDeclaration, ClassExpression, ArrowFunction, MethodDefinition, or GeneratorMethod depending upon the actual characteristics of the object. In particular that the use and placement of white space, line terminators, and semicolons within the representation String is implementation-dependent.

If the object was defined using ECMAScript code and the returned string representation is in the form of a FunctionDeclaration FunctionExpression, GeneratorDeclaration, GeneratorExpression, ClassDeclaration, ClassExpression, or ArrowFunction then the representation must be such that if the string is evaluated, using eval in a lexical context that is equivalent to the lexical context used to create the original object, it will result in a new functionally equivalent object. The returned source code must not mention freely any variables that were not mentioned freely by the original function's source code, even if these "extra" names were originally in scope. If the source code string does meet these criteria then it must be a string for which eval will throw a SyntaxError exception.

The toString function is not generic; it throws a TypeError exception if its this value does not have a [[Call]] internal method. Therefore, it cannot be transferred to other kinds of objects for use as a method.

19.2.3.6 Function.prototype[@@create] ()

The @@create method of an object F performs the following steps:
1. Return the result of calling OrdinaryCreateFromConstructor(F, "%ObjectPrototype%").

The value of the name property of this function is "[Symbol.create]".

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

NOTE This is the default @@create method that is inherited by all ordinary constructor functions that do not explicitly override it.

19.2.3.7 Function.prototype[@@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).

The value of the name property of this function is "[Symbol.hasInstance]".
This property has the attributes \{ [\[Writable\]]: false, [\[Enumerable\]]: false, [\[Configurable\]]: false \}.

NOTE This is the default implementation of @@hasInstance that most functions inherit. @@hasInstance is called by the instanceof operator to determine whether a value is an instance of a specific constructor. An expression such as

\[
v \text{ instanceof } F
\]

evaluates as

\[
F[@@hasInstance](v)
\]

A constructor function can control which objects are recognised as its instances by instanceof by exposing a different @@hasInstance method on the function.

This property is non-writable and non-configurable to prevent tampering that could be used to globally expose the target function of a bound function.

19.2.4 Function Instances

Every function instance is an ECMA-Script function object and has the internal slots listed in Table 24.

Function instances that correspond to strict mode functions and function instances created using the Function.prototype.bind method (19.2.3.2) have properties named caller and arguments that throw a TypeError exception. An ECMA-Script implementation must not associate any implementation specific behaviour with accesses of these properties from strict mode function code.

The Function instances have the following properties:

19.2.4.1 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, [\[Configurable\]]: true \}.

19.2.4.2 name

The value of the name property is a String that is descriptive of the function. The name has no semantic significance but is typically a variable or property name that is used to refer to the function at its point of definition in ECMA-Script code. This property has the attributes \{ [\[Writable\]]: false, [\[Enumerable\]]: false, [\[Configurable\]]: true \}.

Anonymous functions objects that do not have a contextual name associated with them by this specification do not have a name own property but inherit the name property of %FunctionPrototype%.

Function objects created using Function.prototype.bind do not have a name own property.

19.2.4.3 prototype

Function instances that can be used as a constructor have a prototype property. Whenever such a function instance is created another ordinary object is also created and is the initial value of the function’s prototype property. Unless otherwise specified, the value of the prototype property is used to initialise the [[Prototype]] internal slot of a newly created ordinary object before the Function object is invoked as a constructor for that newly created object.

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

Commented [AWB2091]: TODO, need to confirm.
19.3 Boolean Objects

19.3.1 The Boolean Constructor

The Boolean constructor is the %Boolean% intrinsic object and the initial value of the Boolean property of the global object. When Boolean is called as a function rather than as a constructor, it performs a type conversion. However, if the this value passed in the call is an Object with an uninitialised [[BooleanData]] internal slot, it initialises the this value using the argument value. This permits Boolean to be used both to perform type conversion and to perform constructor instance initialisation.

The Boolean constructor is designed to be subclassable. It may be used as the value of an extends clause of a class declaration. Subclass constructors that intended to inherit the specified Boolean behaviour must include a super call to the Boolean constructor to initialise the [[BooleanData]] state of subclass instances.

19.3.1.1 Boolean (value)

When Boolean is called with argument value, the following steps are taken:

1. Let O be the this value.
2. Let b be ToBoolean(value).
3. If Type(O) is Object and O has a [[BooleanData]] internal slot and the value of [[BooleanData]] is undefined, then
   a. Set the value of O’s [[BooleanData]] internal slot to b.
   b. Return O.
4. Return b.

19.3.1.2 new Boolean (... argumentsList)

When Boolean is called as part of a new expression, it initialises a newly created object:

1. Let F be the Boolean function object on which the new operator was applied.
2. Let argumentsList be the argumentsList argument of the [[Construct]] internal method that was invoked by the new operator.
3. Return the result of OrdinaryConstruct (F, argumentsList).

If Boolean is implemented as an ECMAscript function object, its [[Construct]] internal method will perform the above steps.

19.3.2 Properties of the Boolean Constructor

The value of the [[Prototype]] internal slot of the Boolean constructor is the Function prototype object (19.2.3). Besides the length property (whose value is 1), the Boolean constructor has the following properties:

19.3.2.1 Boolean.prototype

The initial value of Boolean.prototype is the Boolean prototype object (19.3.3).

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

19.3.2.2 Boolean[@@create] ()

The @@create method of an object F performs the following steps:

1. Let F be the this value.
2. Let \( \text{obj} \) be the result of calling OrdinaryCreateFromConstructor(F, "BooleanPrototype", ([[BooleanData]])).
3. Return \( \text{obj} \).

The value of the \texttt{name} property of this function is "\texttt{Symbol.create}".

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

\textbf{NOTE} \([\text{BooleanData}]\) is initially assigned the value \texttt{undefined} as a flag to indicate that the instance has not yet been initialised by the Boolean constructor. This flag value is never directly exposed to ECMAScript code; hence implementations may choose to encode the flag in some other manner.

\textbf{19.3.3 Properties of the Boolean Prototype Object}

The Boolean prototype object is an ordinary object. It is not a Boolean instance and does not have a \texttt{[[BooleanData]]} internal slot.

The value of the \texttt{[[Prototype]]} internal slot of the Boolean prototype object is the standard built-in Object prototype object (19.1.3).

The abstract operation \texttt{thisBooleanValue(value)} performs the following steps:

1. If \texttt{Type(value)} is Boolean, return \texttt{value}.
2. If \texttt{Type(value)} is Object and \texttt{value} has a \texttt{[[BooleanData]]} internal slot, then
   a. Let \( b \) be the value of \texttt{value}'s \texttt{[[BooleanData]]} internal slot.
   b. If \( b \) is not \texttt{undefined}, then return \( b \).
3. Throw a \texttt{TypeError} exception.

\textbf{19.3.3.1 Boolean.prototype.constructor}

The initial value of \texttt{Boolean.prototype.constructor} is the built-in \texttt{Boolean} constructor.

\textbf{19.3.3.2 Boolean.prototype.toString ()}

The following steps are taken:

1. Let \( b \) be this\texttt{BooleanValue(this value)}.
2. ReturnIfAbrupt(\( b \)).
3. If \( b \) is \texttt{true}, then return "\texttt{true}"; else return "\texttt{false}".

\textbf{19.3.3.3 Boolean.prototype.valueOf ()}

The following steps are taken:

1. Return this\texttt{BooleanValue(this value)}.

\textbf{19.3.4 Properties of Boolean Instances}

Boolean instances are ordinary objects that inherit properties from the Boolean prototype object. Boolean instances have a \texttt{[[BooleanData]]} internal slot. The \texttt{[[BooleanData]]} internal slot is the Boolean value represented by this Boolean object.
19.4 Symbol Objects

19.4.1 The Symbol Constructor

The Symbol constructor is the %Symbol% intrinsic object and the initial value of the Symbol property of the global object. When Symbol is called as a function rather than as a constructor, it returns a new Symbol value.

The Symbol constructor is not intended to be used with the new operator or to be subclassed. It may be used as the value of an extends clause of a class declaration but a super call to the Symbol constructor will not initialise the state of subclass instances.

19.4.1.1 Symbol (description=undefined)

When Symbol is called with argument description, the following steps are taken:

1. If description is undefined, then let descString be undefined.
2. Else, let descString be ToString(description).
3. ReturnIfAbrupt(descString).
4. Return a new unique Symbol value whose [[Description]] is descString.

19.4.1.2 new Symbol (...argumentsList)

When Symbol is called as part of a new expression, it initialises a newly created object:

1. Let F be the Symbol function object on which the new operator was applied.
2. Let argumentsList be the argumentsList argument of the [[Construct]] internal method that was invoked by the new operator.
3. Return the result of OrdinaryConstruct (F, argumentsList).

If Symbol is implemented as an ECMAScript function object, its [[Construct]] internal method will perform the above steps.

NOTE Symbol has ordinary [[Construct]] behaviour but the definition of its @@create method causes new Symbol to throw a TypeError exception.

19.4.2 Properties of the Symbol Constructor

The value of the [[Prototype]] internal slot of the Symbol constructor is the Function prototype object (19.2.3). Besides the length property (whose value is 0), the Symbol constructor has the following properties:

19.4.2.1 Symbol.create

The initial value of Symbol.create is the well known symbol @@create (Table 1).
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

19.4.2.2 Symbol.hasInstance

The initial value of Symbol.hasInstance is the well known symbol @@hasInstance (Table 1).
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

19.4.2.3 Symbol.isConcatSpreadable

The initial value of Symbol.isConcatSpreadable is the well known symbol @@isConcatSpreadable (Table 1).
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

19.4.2.4 Symbol.isRegExp

The initial value of Symbol.isRegExp is the well known symbol @@isConcatSpreadableisRegExp (Table 1).

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

19.4.2.5 Symbol.iterator

The initial value of Symbol.iterator is the well known symbol @@iterator (Table 1).

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

19.4.2.6 Symbol.prototype

The initial value of Symbol.prototype is the Symbol prototype object (19.4.3).

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

19.4.2.7 Symbol.toPrimitive

The initial value of Symbol.toPrimitive is the well known symbol @@toPrimitive (Table 1).

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

19.4.2.8 Symbol.toStringTag

The initial value of Symbol.toStringTag is the well known symbol @@toStringTag (Table 1).

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

19.4.2.9 Symbol.unscopables

The initial value of Symbol.unscopables is the well known symbol @@unscopables (Table 1).

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

19.4.2.10 Symbol.create

The @@create method of a Symbol object F performs the following steps:

1. Throw a TypeError exception.

The value of the name property of this function is "[Symbol.create]".

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.
19.4.3 Properties of the Symbol Prototype Object

The Symbol prototype object is an ordinary object. It is not a Symbol instance and does not have a 
[[SymbolData]] internal slot.

The value of the [[Prototype]] internal slot of the Symbol prototype object is the standard built-in Object
prototype object (19.1.3).

19.4.3.1 Symbol.prototype.constructor

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

19.4.3.2 Symbol.prototype.toString()

The following steps are taken:
1. Let s be the this value.
2. If s does not have a [[SymbolData]] internal slot, then throw a TypeError exception.
3. Let sym be the value of s’s [[SymbolData]] internal slot.
4. Let desc be the value of sym’s [[Description]] attribute.
5. If desc is undefined, then let desc be the empty string.
6. Assert: Type(desc) is String.
7. Let result be the result of concatenating the strings “Symbol”, desc, and “”.
8. Return result.

19.4.3.3 Symbol.prototype.valueOf()

The following steps are taken:
1. Let s be the this value.
2. If s does not have a [[SymbolData]] internal slot, then throw a TypeError exception.
3. Return the value of s’s [[SymbolData]] internal slot.

19.4.3.4 Symbol.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”. Implicit conversion of Symbol objects to
primitive values is not allowed.

When the @@toPrimitive method is called with argument hint, the following steps are taken:
1. Throw a TypeError exception.

The value of the name property of this function is “Symbol.toPrimitive”.

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

19.4.3.5 Symbol.prototype[@@toStringTag]

The initial value of the @@toStringTag property is the string value “Symbol”.

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

19.4.4 Properties of Symbol Instances

Symbol instances are ordinary objects that inherit properties from the Symbol prototype object. Symbol
instances have a [[SymbolData]] internal slot. The [[SymbolData]] internal slot is the Symbol value
represented by this Symbol object.
19.5 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.

19.5.1 The Error Constructor

The Error constructor is the %Error% intrinsic object and the initial value of the Error property of the global object. 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. However, if the this value passed in the call is an Object with an uninitialised [[ErrorData]] internal slot, it initialises the this value using the argument value rather than creating a new object. This permits Error to be used both as a factory method and to perform constructor instance initialisation.

The Error constructor is designed to be subclassable. It may be used as the value of an extends clause of a class declaration. Subclass constructors that intended to inherit the specified Error behaviour should include a super call to the Error constructor to initialise subclass instances.

19.5.1.1 Error (message)

When the Error function is called with argument message the following steps are taken:

1. Let func be this Error function object.
2. Let O be the this value.
3. If Type(O) is not Object or Type(O) is Object and O does not have an [[ErrorData]] internal slot or Type(O) is Object and O has an [[ErrorData]] internal slot and the value of [[ErrorData]] is not undefined, then:
   a. Let O be the result of calling OrdinaryCreateFromConstructor(func, "%ErrorPrototype", ([[ErrorData]])).
   b. ReturnIfAbrupt(O).
4. Assert: Type(O) is Object.
5. Set the value of O’s [[ErrorData]] internal slot to any value other than undefined.
6. If message is not undefined, then:
   a. Let msg be ToString(message).
   b. ReturnIfAbrupt(msg).
   c. Let msgDesc be the PropertyDescriptor{[[Value]]: msg, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true}.
   d. Let status be the result of DefinePropertyOrThrow(O, "message", msgDesc).
   e. ReturnIfAbrupt(status).
7. Return O.

19.5.1.2 new Error(... argumentsList)

When Error called as part of a new expression with argument list argumentsList it performs the following steps:

1. Let F be the Error function object on which the new operator was applied.
2. Let argumentsList be the argumentsList argument of the [[Construct]] internal method that was invoked by the new operator.
3. Return the result of OrdinaryConstruct (F, argumentsList).

If Error is implemented as an ECMAScript function object, its [[Construct]] internal method will perform the above steps.
19.5.2 Properties of the Error Constructor

The value of the [[Prototype]] internal slot of the Error constructor is the Function prototype object (19.2.3).

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

19.5.2.1 Error.prototype

The initial value of Error.prototype is the Error prototype object (19.5.3).

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

19.5.2.2 Error @@create ()

The @@create method of an object F performs the following steps:

1. Let F be the this value.
2. Let obj be the result of calling OrdinaryCreateFromConstructor(F, "%ErrorPrototype%, ([ErrorData])).
3. Return obj.

The value of the name property of this function is "[Symbol.create]".

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

NOTE [[ErrorData]] is initially assigned the value undefined as a flag to indicate that the instance has not yet been initialised by the Error constructor. This flag value is never directly exposed to ECMAScript code; hence implementations may choose to encode the flag in some other manner.

19.5.3 Properties of the Error Prototype Object

The Error prototype object is an ordinary object. It is not an Error instance and does not have an [[ErrorData]] internal slot.

The value of the [[Prototype]] internal slot of the Error prototype object is the standard built-in Object prototype object (19.1.3).

19.5.3.1 Error.prototype.constructor

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

19.5.3.2 Error.prototype.message

The initial value of Error.prototype.message is the empty String.

19.5.3.3 Error.prototype.name

The initial value of Error.prototype.name is "Error".

19.5.3.4 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`.

19.5.4 Properties of Error Instances

Error instances are ordinary objects that inherit properties from the Error prototype object and have an [[ErrorData]] internal slot whose initial value is `undefined`. The only specified uses of `[[ErrorData]]` is to flag whether or not an Error instance has been initialised by the `Error` constructor and to identify them as `Error` objects within `Object.prototype.toString`.

19.5.5 Native Error Types Used in This Standard

A new instance of one of the `NativeError` objects below is thrown when a runtime error is detected. All of these objects share the same structure, as described in 19.5.6.

19.5.5.1 EvalError

This exception is not currently used within this specification. This object remains for compatibility with previous editions of this specification.

19.5.5.2 RangeError

Indicates a value that is not in the set or range of allowable values. 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.

19.5.5.3 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.

19.5.5.4 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.

19.5.5.5 TypeError

Indicates that 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, 11.4.6, 11.8.7, 11.8.8, 11.8.9, 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, 15.3.3.2, 15.3.3.3, 15.3.3.4, 15.3.3.5, 15.3.3.6, 15.3.3.7, 15.3.3.8, 15.3.3.9, 15.3.3.10, 15.3.3.11, 15.3.3.12, 15.3.3.13, 15.3.3.14, 15.3.3.15, 15.3.3.16, 15.3.3.17, 15.3.3.18, 15.3.3.19, 15.3.3.20, 15.3.3.21, 15.3.3.22, 15.4.5.1, 15.5.4.3, 15.5.4.3, 15.6.4.2, 15.6.4.3, 15.7.14, 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.

19.5.5.6 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.

Commented [AWB1093]: Section references have not yet been updated to reflect ES6
19.5.6 NativeError Object Structure

When an ECMAScript implementation detects a runtime error, it throws a new instance of one of the NativeError objects defined in 19.5.5. 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 19.5.5.

19.5.6.1 NativeError Constructors

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. However, if the this value passed in the call is an Object with an uninitialised [[ErrorData]] internal slot, it initialises the this value using the argument value. This permits a NativeError to be used both as a factory method and to perform constructor instance initialisation.

The NativeError constructor is designed to be subclassable. It may be used as the value of an extends clause of a class declaration. Subclass constructors that intended to inherit the specified NativeError behaviour should include a super call to the NativeError constructor to initialise subclass instances.

19.5.6.1.1 NativeError (message)

When a NativeError function is called with argument message the following steps are taken:

1. Let func be this NativeError function object.
2. Let O be the this value.
3. If Type(O) is not Object or Type(O) is Object and O does not have an [[ErrorData]] internal slot or Type(O) is Object and O has an [[ErrorData]] internal slot and the value of [[ErrorData]] is not undefined, then:
   a. Let O be the result of calling OrdinaryCreateFromConstructor(func, "%NativeErrorPrototype", ([[ErrorData]])).
   b. ReturnIfAbrupt(O).
4. Assert: Type(O) is Object.
5. Set the value of O's [[ErrorData]] internal slot to any value other than undefined.
6. If message is not undefined, then:
   a. Let msg be ToString(message).
   b. Let msgDesc be the PropertyDescriptor{ [[Value]]: msg, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true }.
   c. Let status be the result of DefinePropertyOrThrow(O, "message", msgDesc).
   d. ReturnIfAbrupt(status).
7. Return O.

The actual value of the string passed in step 3.a is either "%EvalErrorPrototype", "%RangeErrorPrototype", "%ReferenceErrorPrototype", "%TypeErrorPrototype", or "%URIErrorPrototype" corresponding to which NativeError constructor is being defined.

19.5.6.1.2 new NativeError(... argumentsList )

When a NativeError constructor is called as part of a new expression with argument list argumentsList it performs the following steps:

1. Let F be this NativeError function object on which the new operator was applied.
2. Let argumentsList be the argumentsList argument of the [[Construct]] internal method that was invoked by the new operator.
3. Return the result of OrdinaryConstruct (F, argumentsList).
19.5.6.2 Properties of the `NativeError` Constructors

The value of the `[[Prototype]]` internal slot of a `NativeError` constructor is the `Error` constructor object (19.5.1). Besides the `length` property (whose value is 1), each `NativeError` constructor has the following properties:

19.5.6.2.1 `NativeError.prototype`

The initial value of `NativeError.prototype` is a `NativeError` prototype object (19.5.6.3). Each `NativeError` constructor has a separate prototype object.

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

19.5.6.2.2 `NativeError[@@create]()`

The `@@create` method of an object `F` performs the following steps:

1. Let `F` be the `this` value.
2. Let `obj` be the result of calling `OrdinaryCreateFromConstructor(F, NativeErrorPrototype, ([ErrorData]))`.
3. Return `obj`.

The actual value passed as `NativeErrorPrototype` in step 2 is either `%EvalErrorPrototype%`, `%RangeErrorPrototype%`, `%ReferenceErrorPrototype%`, `%SyntaxErrorPrototype%`, `%TypeErrorPrototype%`, or `%URIErrorPrototype%` corresponding to which `NativeError` constructor is being defined.

The value of the `name` property of this function is `"[Symbol.create]"`.

This property has the attributes `[[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true`.

NOTE `[[ErrorData]]` is initially assigned the value `undefined` as a flag to indicate that the instance has not yet been initialised by the `NativeError` constructor. This flag value is never directly exposed to ECMAScript code; hence implementations may choose to encode the flag in some other manner.

19.5.6.3 Properties of the `NativeError` Prototype Objects

Each `NativeError` prototype object is an ordinary object. It is not an `Error` instance and does not have an `[[ErrorData]]` internal slot.

The value of the `[[Prototype]]` internal slot of each `NativeError` prototype object is the standard built-in `Error` prototype object (19.5.3).

19.5.6.3.1 `NativeError.prototype.constructor`

The initial value of the `constructor` property of the prototype for a given `NativeError` constructor is the `NativeError` constructor function itself (19.5.6.1).

19.5.6.3.2 `NativeError.prototype.message`

The initial value of the `message` property of the prototype for a given `NativeError` constructor is the empty String.
19.5.6.3.3  NativeError.prototype.name

The initial value of the name property of the prototype for a given NativeError constructor is a string consisting of the name of the constructor (the name used instead of NativeError).

19.5.6.4  Properties of NativeError Instances

NativeError instances are ordinary objects that inherit properties from their NativeError prototype object and have an [[ErrorData]] internal slot whose initial value is undefined. The only specified use of [[ErrorData]] is to flag whether or not an Error or NativeError instance has been initialised by its constructor.

20  Numbers and Dates

20.1  Number Objects

20.1.1  The Number Constructor

The Number constructor is the %Number% intrinsic object and the initial value of the Number property of the global object. When Number is called as a function rather than as a constructor, it performs a type conversion. However, if the this value passed in the call is an Object with an uninitialised [[NumberData]] internal slot, it initialises the this value using the argument value. This permits Number to be used both to perform type conversion and to perform constructor instance initialisation.

The Number constructor is designed to be subclassable. It may be used as the value of an extends clause of a class declaration. Subclass constructors that intended to inherit the specified Number behaviour must include a super call to the Number constructor to initialise the [[NumberData]] state of subclass instances.

20.1.1.1  Number ( [ value ] )

When Number is called with argument number, the following steps are taken:

1. Let O be the this value.
2. If no arguments were passed to this function invocation, then let n be +0.
3. Else, let n be ToNumber(value).
4. ReturnIfAbrupt(n).
5. If Type(O) is Object and O has a [[NumberData]] internal slot and the value of [[NumberData]] is undefined, then
   a. Set the value of O’s [[NumberData]] internal slot to n.
   b. Return O.
6. Return n.

20.1.1.2  new Number ( ...argumentsList )

When Number is called as part of a new expression with argument list argumentsList it performs the following steps:

1. Let F be the Number function object on which the new operator was applied.
2. Let argumentsList be the argumentsList argument of the [[Construct]] internal method that was invoked by the new operator.
3. Return the result of OrdinaryConstruct (F, argumentsList).

If Number is implemented as an ECMAScript function object, its [[Construct]] internal method will perform the above steps.

20.1.2  Properties of the Number Constructor

The value of the [[Prototype]] internal slot of the Number constructor is the Function prototype object (19.2.3).
Besides the `length` property (whose value is 1), the Number constructor has the following properties:

### 20.1.2.1 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 \(2.220446049250313084729533361816 \times 10^{-16}\).

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

### 20.1.2.2 Number.isFinite (number)

When the `Number.isFinite` is called with one argument `number`, the following steps are taken:

1. If `Type(number)` is not Number, return false.
2. If `number` is NaN, +\(\infty\), or -\(\infty\), return false.
3. Otherwise, return true.

### 20.1.2.3 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. If `number` is NaN, +\(\infty\), or -\(\infty\), return false.
3. Let `integer` be `ToInteger(number)`.
4. If `integer` is not equal to `number`, return false.
5. Otherwise, return true.

### 20.1.2.4 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 (18.2.3) is that it does not convert its argument to a Number before determining whether it is NaN.

### 20.1.2.5 Number.isSafeInteger (number)

When the `Number.isSafeInteger` 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.

### 20.1.2.6 Number.MAX_SAFE_INTEGER

The value of Number.MAX_SAFE_INTEGER is 9007199254740991 (\(2^{53} - 1\)).

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.
20.1.2.7 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 \}.

20.1.2.8 Number.NaN

The value of Number.NaN is NaN.

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

20.1.2.9 Number.NEGATIVE_INFINITY

The value of Number.NEGATIVE_INFINITY is $-\infty$.

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

20.1.2.10 Number.MIN_SAFE_INTEGER

The value of Number.MIN_SAFE_INTEGER is $-9007199254740991 (-2^{53}-1)$.

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

20.1.2.11 Number.MIN_VALUE

The value of Number.MIN_VALUE is the smallest positive value of the Number type, which is approximately $5 \times 10^{-324}$.

In the IEEE-754 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 \}.

20.1.2.12 Number.parseFloat (string)

The value of the Number.parseFloat data property is the same built-in function object that is the value of the parseFloat property of the global object defined in 18.2.4.

20.1.2.13 Number.parseInt (string, radix)

The value of the Number.parseInt data property is the same built-in function object that is the value of the parseInt property of the global object defined in 18.2.5.

20.1.2.14 Number.POSITIVE_INFINITY

The value of Number.POSITIVE_INFINITY is $+\infty$.

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

20.1.2.15 Number.prototype

The initial value of Number.prototype is the Number prototype object (20.1.3).

This property has the attributes \{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false \}. 
20.1.2.16 Number[@create] ()

The @@create method of an object F performs the following steps:

1. Let F be the this value.
2. Let obj be the result of calling OrdinaryCreateFromConstructor(F, "%NumberPrototype%", ([NumberData])).
3. Return obj.

The value of the name property of this function is "[Symbol.create]".

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

NOTE [[NumberData]] is initially assigned the value undefined as a flag to indicate that the instance has not yet been initialised by the Number constructor. This flag value is never directly exposed to ECMAScript code, hence implementations may choose to encode the flag in some other manner.

20.1.3 Properties of the Number Prototype Object

The Number prototype object is an ordinary object. It is not a Number instance and does not have a [[NumberData]] internal slot.

The value of the [[Prototype]] internal slot of the Number prototype object is the standard built-in Object prototype object (19.1.3).

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 slot that has been initialised to a Number value.

The abstract operation thisNumberValue(value) performs the following steps:

1. If Type(value) is Number, return value.
2. If Type(value) is Object and value has a [[NumberData]] internal slot, then
   a. Let n be the value of value’s [[NumberData]] internal slot.
   b. If n is not undefined, then return n.
3. Throw a TypeError exception.

The phrase "this Number value" within the specification of a method refers to the result returned by calling the abstract operation thisNumberValue with the this value of the method invocation passed as the argument.

20.1.3.1 Number.prototype.clz() When Number.prototype.clz is called with one argument number, the following steps are taken:

1. Let x be thisNumberValue(this 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.

20.1.3.2 Number.prototype.constructor

The initial value of Number.prototype.constructor is the built-in Number constructor.
20.1.3.3 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 thisNumberValue(this value).
2. ReturnIfAbrupt(x).
3. Let f be ToInteger(fractionDigits).
4. Assert: f is 0, when fractionDigits is undefined.
5. ReturnIfAbrupt(f).
6. If x is NaN, return the String "NaN".
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 f < 0 or f > 20, throw a RangeError exception.
10. If x = 0, then
    a. Let m be the String consisting of f+1 occurrences of the code unit 0x0030.
    b. Let e = 0.
11. Else x ≠ 0, then
    a. If fractionDigits is not undefined, then
       i. Let e and n be integers such that 10^f ≤ n < 10^{f+1} and for which the exact mathematical value
          of n \times 10^{-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^{-f} is larger.
       b. Else fractionDigits is undefined, then
          i. Let e, n, and f be integers such that f ≥ 0, 10^f ≤ n < 10^{f+1}, the number value for n \times 10^{-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. Else x ≠ 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. If e < 0, then
        a. Let c = "-".
        b. Else e = 0,
           i. Let c = "-".
           ii. Let e = –e.
        c. Let d be the String consisting of the digits of the decimal representation of e (in order, with no
leading zeroes).
    15. Else
        a. If e > 0, then let c = "+".
        b. Else e < 0,
           i. Let c = "-".
           ii. Let e = –e.
        c. Let d be the String consisting of the digits of the decimal representation of e (in order, with no
leading zeroes).
    16. Let m be the concatenation of the four Strings m, "\e", c, and d.
    17. 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 17).
An implementation is permitted to extend the behaviour of \texttt{toExponential} for values of \texttt{fractionDigits} less than 0 or greater than 20. In this case \texttt{toExponential} would not necessarily throw \texttt{RangeError} for such values.

\textbf{NOTE}  For implementations that provide more accurate conversions than required by the rules above, it is recommended that the following alternative version of step 12.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 $n \times 10^{-f}$ is $x$, and $f/n$ as small as possible. If there are multiple possibilities for $n$, choose the value of $n$ for which $n \times 10^{-f}$ is closest in value to $x$. If there are two such possible values of $n$, choose the one that is even.

20.1.3.4 \texttt{Number.prototype.toFixed} (\texttt{fractionDigits})

\textbf{Note} \texttt{toFixed} returns a String containing this Number value represented in decimal fixed-point notation with \texttt{fractionDigits} digits after the decimal point. If \texttt{fractionDigits} is \texttt{undefined}, 0 is assumed.

The following steps are performed:

1.  Let $x$ be this\texttt{NumberValue}(\texttt{this} value).
2.  ReturnIfAbrupt($x$).
3.  Let $f$ be To\texttt{Integer}(\texttt{fractionDigits}). (If \texttt{fractionDigits} is \texttt{undefined}, this step produces the value 0).
4.  ReturnIfAbrupt($f$).
5.  If $f < 0$ or $f > 20$, throw a \texttt{RangeError} exception.
6.  If $x$ is \texttt{NaN}, return the String "\texttt{NaN}".
7.  Let $s$ be the empty String.
8.  If $x < 0$, then
   a.  Let $s$ be "-".
   b.  Let $x = -x$.
9.  If $x \geq 10^{21}$, then
   a.  Let $m = \text{ToString}(x)$.
10. Else $x < 10^{21}$,
    a.  Let $n$ be an integer for which the exact mathematical value of $n + 10^f - x$ 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 \texttt{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 $b$.
11. Return the concatenation of the Strings $s$ and $m$.

The \texttt{length} property of the \texttt{toFixed} method is 1.

If the \texttt{toFixed} method is called with more than one argument, then the behaviour is undefined (see clause 17).

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

\textbf{NOTE}  The output of \texttt{toFixed} may be more precise than \texttt{toString} for some values because \texttt{toString} only prints enough significant digits to distinguish the number from adjacent number values. For example,


20.1.3.5 Number.prototype.toLocaleString(reserved1=undefined, reserved2=undefined)

An ECMAScript implementation that includes the ECMA-402 International API must implement the `Number.prototype.toLocaleString` method as specified in the ECMA-402 specification. If an ECMAScript implementation does not include the ECMA-402 API the following specification of the `toLocaleString` method is used.

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`.

The meaning of the optional parameters to this method are defined in the ECMA-402 specification; implementations that do not include ECMA-402 support must not use those parameter position for anything else.

The `length` property of the `toLocaleString` method is 0.

20.1.3.6 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` (7.1.9) instead.

Specifically, perform the following steps:

1. Let `x` be this `Number` value (this value).
2. ReturnIfAbrupt(x).
3. If `precision` is `undefined`, return `toString(x)`.
4. ReturnIfAbrupt(precision).
5. Let `p` be `ToInteger(precision)`.
6. ReturnIfAbrupt(p).
7. Let `x` be the empty string.
8. If `x < 0`, then
   a. Let `s` be `“-“`.
   b. Let `x` be `–x`.
9. If `x = +∞`, 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 ≠ 0`,
   a. Let `e` and `n` be integers such that `10^{e−1} ≤ n < 10^e` and for which the exact mathematical value of `n × 10^{-p+1}` 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 × 10^{-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 ≥ p`, then
      i. Assert: `e ≠ 0`.
      ii. Let `a` be the first element of `m`, and let `b` be the remaining `p−1` elements of `m`.
      iii. Let `m` be the concatenation of the three strings `a`, “.”, and `b`.
      iv. If `e > 0`, then
          1. Let `e = “+“`.
          2. Else `e < 0`, then
             1. Let `e = “−“`.
             2. Let `e = −e`.  

© Ecma International 2009 – All rights reserved
vi. Let \( d \) be the String consisting of the digits of the decimal representation of \( e \) (in order, with no leading zeroes).

vii. Return the concatenation of the five Strings \( s, m, \cdot, c, \) and \( \cdot d \).

13. If \( e = p - 1 \), then return the concatenation of the Strings \( s \) and \( m \).

14. If \( e \geq 0 \), then
   a. Let \( m \) be the concatenation of the first \( e + 1 \) 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 \texttt{Length} property of the \texttt{toPrecision} method is 1.

If the \texttt{toPrecision} method is called with more than one argument, then the behaviour is undefined (see clause 17).

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

20.1.3.7 \texttt{Number.prototype.toString ([ radix ])}

The optional \texttt{radix} should be an integer value in the inclusive range 2 to 36. If \texttt{radix} not present or is \texttt{undefined} the Number 10 is used as the value of \texttt{radix}. If \texttt{ToInteger(radix)} is the Number 10 then this Number value is given as an argument to the \texttt{ToString} abstract operation; the resulting String value is returned.

If \texttt{ToInteger(radix)} is not an integer between 2 and 36 inclusive throw a \texttt{RangeError} exception. If \texttt{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 \texttt{a}-\texttt{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 7.1.12.1.

The \texttt{toString} function is not generic; it throws a \texttt{TypeError} exception if its \texttt{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.

20.1.3.8 \texttt{Number.prototype.valueOf ()}

1. Let \( x \) be this\texttt{NumberValue} (this value).

2. Return \( x \).

20.1.4 Properties of Number Instances

Number instances are ordinary objects that inherit properties from the Number prototype object. Number instances also have a \([\text{NumberData}]\) internal slot. The \([\text{NumberData}]\) internal slot is the Number value represented by this Number object.

20.2 The Math Object

The Math object is a single ordinary object.

The value of the \([\text{Prototype}]\) internal slot of the Math object is the standard built-in Object prototype object (19.1.3).
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 also 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 6.1.6.

### 20.2.1 Value Properties of the Math Object

#### 20.2.1.1 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 }.

#### 20.2.1.2 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 }.

#### 20.2.1.3 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.

#### 20.2.1.4 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 }.

#### 20.2.1.5 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.

#### 20.2.1.6 Math.PI

The Number value for $\pi$, the ratio of the circumference of a circle to its diameter, which is approximately 3.1415926535897932.

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

#### 20.2.1.7 Math.SQRT1_2

The Number value for the square root of $\frac{1}{2}$, which is approximately 0.7071067811865476.

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

**NOTE** The value of Math.SQRT1_2 is approximately the reciprocal of the value of Math.SQRT2.
20.2.1.8 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 )

20.2.1.9 Math[@@toStringTag]

The initial value of the @@toStringTag property is the string value "Math".

This property has the attributes ( [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true )

20.2.2 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, the function 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 6.1.6.

NOTE The behaviour of the functions acos, acosh, asin, asinh, atan, atanh, atan2, cbrt, cos, cosh, exp, hypot, log, log1p, log2, log10, pow, sin, sinh, sqrt, tan, and tanh 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).

20.2.2.1 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 +∞.

20.2.2.2 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.

20.2.2.3 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 +\( \infty \), the result is +\( \infty \).

20.2.2.4 Math.asin (x)

Returns an implementation-dependent approximation to the arc sine of \( x \). The result is expressed in radians and ranges from \(-\pi/2\) to +\( \pi/2\).

• 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 \(-0\).

20.2.2.5 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 +\( \infty \), the result is +\( \infty \).
• If \( x \) is \(-\infty \), the result is \(-\infty \).

20.2.2.6 Math.atan (x)

Returns an implementation-dependent approximation to the arc tangent of \( x \). The result is expressed in radians and ranges from \(-\pi/2\) to +\( \pi/2\).

• 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 +\( \infty \), the result is an implementation-dependent approximation to +\( \pi/2\).
• If \( x \) is \(-\infty \), the result is an implementation-dependent approximation to \(-\pi/2\).

20.2.2.7 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 \(-\infty\).
• If \( x \) is +1, the result is +\( \infty \).
• If \( x \) is +0, the result is +0.
• If \( x \) is \(-0\), the result is \(-0\).

20.2.2.8 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 \(-\pi\) to +\( \pi\).

• If either \( y \) or \( x \) is NaN, the result is NaN.
20.2.2.11 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 \(+0\), the result is NaN.
• If \( x \) is \(-\infty\), the result is NaN.

20.2.2.12 Math.cosh(\( x \))

Returns an implementation-dependent approximation to the hyperbolic \( \cos \) of \( x \).

- If \( x \) is NaN, the result is NaN.
- If \( x \) is \(+0\), the result is 1.
- If \( x \) is \(-0\), the result is 1.
- If \( x \) is \(+\infty\), the result is \(+\infty\).
- If \( x \) is \(-\infty\), the result is NaN.

\NOTE\ The value of \( \cosh(x) \) is the same as \((\exp(x) + \exp(-x))/2\).

20.2.2.13 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 \(+\infty\), the result is \(+\infty\).
- If \( x \) is \(-\infty\), the result is \(+0\).

\NOTE\ The value of \( \exp(x) \) is the same as \((\exp(x) + \exp(-x))/2\).

20.2.2.14 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 to 0.

- If \( x \) is NaN, the result is NaN.
- If \( x \) is \(+0\), the result is \(+0\).
- If \( x \) is \(-0\), the result is \(-0\).
- If \( x \) is \(+\infty\), the result is \(+\infty\).
- If \( x \) is \(-\infty\), the result is \(-1\).

\NOTE\ The value of \( \expm1(x) \) is the same as \(-\exp(-x)\).

20.2.2.15 Math.floor(\( x \))

Returns the greatest (closest to \(+\infty\)) Number value that is not greater than \( x \) and is equal to a mathematical integer. If \( x \) is already an integer, the result is \( x \).

- If \( x \) is NaN, the result is NaN.
- If \( x \) is \(+0\), the result is \(+0\).
- If \( x \) is \(-0\), the result is \(-0\).
- If \( x \) is \(+\infty\), the result is \(+\infty\).
- If \( x \) is \(-\infty\), the result is \(-\infty\).
- If \( x \) is greater than 0 but less than 1, the result is \(+0\).

\NOTE\ The value of \( \text{Math.floor}(x) \) is the same as the value of \( \text{Math.ceil}(-x) \).

20.2.2.16 Math.fround(\( x \))

When \( \text{Math.fround} \) is called with argument \( x \) the following steps are taken:

1. If \( x \) is NaN, return NaN.
2. If \( x \) is one of \( +0, -0, +\infty, -\infty \), then return \( x \).
3. Let \( x_{32} \) be the result of converting \( x \) to a value in IEEE-754-2005 binary32 format using roundTiesToEven.
4. Let \( x_{64} \) be the result of converting \( x_{32} \) to a value in IEEE-754-2005 binary64 format.
5. Return the ECMAScript Number value corresponding to \( x_{64} \).

20.2.2.17 \texttt{Math.hypot([ value1 [, value2 [, ... ]] ])}

\texttt{Math.hypot} returns an implementation-dependent approximation of the square root of the sum of squares of its arguments.

- If no arguments are passed, the result is \(+0\).
- If any argument is \(+\infty\), the result is \(+\infty\).
- If any argument is \(-\infty\), the result is \(+\infty\).
- If no argument is \(+\infty\) or \(-\infty\), and any argument is NaN, the result is NaN.
- If all arguments are either \(+0\) or \(-0\), the result is \(+0\).
- The implementation must avoid underflow.
- The implementation must avoid overflow, where possible.
- The implementation must minimise rounding errors.

The length property of the \texttt{hypot} function is 2.

20.2.2.18 \texttt{Math.imul(x, y)}

When the \texttt{Math.imul} is called with arguments \( x \) and \( y \) the following steps are taken:

1. Let \( a \) be ToUint32(\( x \)).
2. ReturnIfAbrupt(\( a \)).
3. Let \( b \) be ToUint32(\( y \)).
4. ReturnIfAbrupt(\( b \)).
5. Let \( \text{product} \) be \((a \times b) \mod 2^{32}\).
6. If \( \text{product} \geq 2^{31} \), return \( \text{product} - 2^{31} \), otherwise return \( \text{product} \).

20.2.2.19 \texttt{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 \(-\infty\).
- If \( x \) is 1, the result is \(+0\).
- If \( x \) is \(+\infty\), the result is \(+\infty\).

20.2.2.20 \texttt{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 \(-\infty\).
- If \( x \) is \(+0\), the result is \(+0\).
- If \( x \) is \(-0\), the result is \(-0\).
- If \( x \) is \(+\infty\), the result is \(+\infty\).
20.2.2.21 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 \(-\infty\). 
- If x is \(-0\), the result is \(-\infty\).
- If x is 1, the result is +0.
- If x is +\(\infty\), the result is +\(\infty\).

20.2.2.22 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 \(-\infty\).
- If x is \(-0\), the result is \(-\infty\).
- If x is 1, the result is +0.
- If x is +\(\infty\), the result is +\(\infty\).

20.2.2.23 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 \(-\infty\).
- If any value is NaN, the result is NaN.
- The comparison of values to determine the largest value is done using the Abstract Relational Comparison algorithm (7.2.8) except that +0 is considered to be larger than –0.

The length property of the max method is 2.

20.2.2.24 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 +\(\infty\).
- If any value is NaN, the result is NaN.
- The comparison of values to determine the smallest value is done using the Abstract Relational Comparison algorithm (7.2.8) except that +0 is considered to be larger than –0.

The length property of the min method is 2.

20.2.2.25 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 +\(\infty\), the result is +\(\infty\).
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 abs(x) < 1 and y is +∞, the result is NaN.
• If abs(x) > 1 and y is +∞, the result is NaN.
• If abs(x) < 1 and y is +∞, the result is NaN.
• If abs(x) > 1 and y is +∞, the result is NaN.

20.2.2.26 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.

Each Math.random function created for distinct code Realms must produce a distinct sequence of values from successive calls.

20.2.2.27 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.

20.2.2.28 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.

20.2.2.29 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 +\( \infty \) or −\( \infty \), the result is NaN.

20.2.2.30 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 +\( \infty \), the result is +\( \infty \).
• If \( x \) is −\( \infty \), the result is −\( \infty \).

NOTE The value of \( \sinh(x) \) is the same as \((\exp(x) - \exp(-x))/2\).

20.2.2.31 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 +\( \infty \), the result is +\( \infty \).

20.2.2.32 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 +\( \infty \) or −\( \infty \), the result is NaN.

20.2.2.33 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 +\( \infty \), the result is +1.
• If \( x \) is −\( \infty \), the result is −1.

NOTE The value of \( \tanh(x) \) is the same as \((\exp(x) - \exp(-x))/\exp(x) + \exp(-x))\).
20.2.2.34 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 +\( \infty \), the result is +\( \infty \).
- If \( x \) is -\( \infty \), the result is -\( \infty \).
- If \( x \) is greater than 0 but less than 1, the result is +0.
- If \( x \) is less than 0 but greater than -1, the result is -0.

20.3 Date Objects

20.3.1 Overview of Date Objects and Definitions of Abstract Operations

The following functions are abstract operations that operate on time values (defined in 20.3.1.1). Note that, in every case, if any argument to one of these functions is NaN, the result will be NaN.

20.3.1.1 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.

20.3.1.2 Day Number and Time within Day

A given time value \( t \) belongs to day number

\[ \text{Day}(t) = \text{floor}(t/\text{msPerDay}) \]

where the number of milliseconds per day is

\[ \text{msPerDay} = 86400000 \]

The remainder is called the time within the day:

\[ \text{TimeWithinDay}(t) = t \mod \text{msPerDay} \]

20.3.1.3 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
A date number is identified by an integer in the range 01 January, 1970.

A month value of

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:

The leap-year function is 1 for a time within a leap year and otherwise is zero:

The time value of the start of a year is:

A time value determines a year by:

Where

DayFromYear(t) = Day(t)−DayFromYearFromTime(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.

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:

where
20.3.1.6 Week Day

The weekday for a particular time value \( t \) is defined as

\[
\text{WeekDay}(t) = (\text{Day}(t) + 4) \mod 7
\]

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.

20.3.1.7 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.

20.3.1.8 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. An implementation of ECMAScript is expected to make its best effort to determine the local daylight saving time adjustment.

20.3.1.9 Local Time

Conversion from UTC to local time is defined by

\[
\text{LocalTime}(t) = t + \text{LocalTZA} + \text{DaylightSavingTA}(t)
\]

Conversion from local time to UTC is defined by

\[
\text{UTC}(t) = t - \text{LocalTZA} - \text{DaylightSavingTA}(t) - \text{LocalTZA}
\]

Note that UTC(LocalTime(\( t \))) is not necessarily always equal to \( t \).

20.3.1.10 Hours, Minutes, Second, and Milliseconds

The following functions are useful in decomposing time values:

\[
\begin{align*}
\text{HourFromTime}(t) & = \text{floor}(t / \text{msPerHour}) \mod \text{HoursPerDay} \\
\text{MinFromTime}(t) & = \text{floor}(t / \text{msPerMinute}) \mod \text{MinutesPerHour} \\
\text{SecFromTime}(t) & = \text{floor}(t / \text{msPerSecond}) \mod \text{SecondsPerMinute} \\
\text{msFromTime}(t) & = t \mod \text{msPerSecond}
\end{align*}
\]

where

\[
\begin{align*}
\text{HoursPerDay} & = 24 \\
\text{MinutesPerHour} & = 60 \\
\text{SecondsPerMinute} & = 60 \\
\text{msPerSecond} & = 1000 \\
\text{msPerMinute} & = 60000 = \text{msPerSecond} \times \text{SecondsPerMinute}
\end{align*}
\]
20.3.1.11 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 \( h \times \text{msPerHour} + m \times \text{msPerMinute} + s \times \text{msPerSecond} + \text{milli} \), performing the arithmetic according to IEEE 754 rules (that is, as if using the ECMAScript operators * and +).
7. Return t.

20.3.1.12 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.

20.3.1.13 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.

20.3.1.14 TimeClip (time)

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

1. If time is not finite, return NaN.
2. If abs(time) > 8.64 \times 10^{15}, return NaN.
3. Return 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.

20.3.1.15 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.

20.3.1.15.1 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.
20.3.2 The Date Constructor

The Date constructor is the %Date% intrinsic object and the initial value of the Date property of the global object. When Date is called as a function rather than as a constructor, it returns a String representing the current time (UTC). However, if the this value passed in the call is an Object with an uninitialised [[DateValue]] internal slot, Date initialises the this object using the argument value. This permits Date to be used both as a function for creating data strings and to perform constructor instance initialisation.

The Date constructor is designed to be subclassable. It may be used as the value of an extends clause of a class declaration. Subclass constructors that intended to inherit the specified Date behaviour must include a super call to the Date constructor to initialise the [[DateValue]] state of subclass instances.

20.3.2.1 Date (year, month,, date, hours, minutes, seconds, ms)

This description applies only if the Date constructor is called with at least two arguments.

When the Date function is called the following steps are taken:

1. Let numberOfArgs be the number of arguments passed to this constructor call.
3. Let O be the this value.
4. If Type(O) is Object and O has a [[DateValue]] internal slot and the value of [[DateValue]] is undefined, then
   a. Let y be ToNumber(year).
   b. ReturnIfAbrupt(year).
   c. Let m be ToNumber(month).
   d. ReturnIfAbrupt(month).
   e. If date is supplied then let dt be ToNumber(date); else let dt be 1.
   f. ReturnIfAbrupt(date).
   g. If hours is supplied then let h be ToNumber(hours); else let h be 0.
   h. ReturnIfAbrupt(hours).
   i. If minutes is supplied then let min be ToNumber(minutes); else let min be 0.
   j. ReturnIfAbrupt(minutes).
   k. If seconds is supplied then let s be ToNumber(seconds); else let s be 0.
   l. ReturnIfAbrupt(seconds).
   m. If ms is supplied then let milli be ToNumber(ms); else let milli be 0.
   n. ReturnIfAbrupt(milliseconds).
   o. If y is not NaN and 0 ≤ ToInteger(y) ≤ 99, then let yr be 1900+ToInteger(y); otherwise, let yr be y.
   p. Let finalDate be MakeDate(MakeDay(yr, m, dt), MakeTime(h, min, s, milli)).
   q. Set the [[DateValue]] internal slot of O to TimeClip(UTC(finalDate)).
   r. Return O.
5. Else,
   a. Return the result computed as if by the expression (new Date()).toString() where Date is this function and toString is the standard built-in method Date.prototype.toString.

20.3.2.2 Date (value)

This description applies only if the Date constructor is called with exactly one argument.
When the `Date` function is called the following steps are taken:

1. Let `numberOfArgs` be the number of arguments passed to this constructor call.
3. Let `O` be the `this` value.
4. If `Type(O)` is `Object` and `O` has a `[[DateValue]]` internal slot and the value of `[[DateValue]]` is `undefined`, then
   a. If `Type(value)` is `Object` and `value` has a `[[DateValue]]` internal slot, then
      i. Let `tv` be `thisTimeValue(value)`.
   b. Else, i. Let `v` be `ToPrimitive(value)`. ii. If `Type(v)` is `String`, then
      1. Let `tv` be the result of parsing `v` as a date, in exactly the same manner as for the `parse` method (20.3.3.2). If the parse resulted in an abrupt completion, `tv` is the Completion Record. iii. Else, 1. Let `tv` be `ToNumber(v)`.
   c. ReturnIfAbrupt(`tv`). d. Set the `[[DateValue]]` internal slot of `O` to `TimeClip(tv)`. e. Return `O`.
5. Else, a. Return the result computed as if by the expression `(new Date()).toString()` where `Date` is this function and `toString` is the standard built-in method `Date.prototype.toString`.

20.3.2.3 `Date()`

This description applies only if the `Date` constructor is called with no arguments.

When the `Date` function is called the following steps are taken:

1. Let `numberOfArgs` be the number of arguments passed to this constructor call.
2. Assert: `numberOfArgs` = 0.
3. Let `O` be the `this` value.
4. If `Type(O)` is `Object` and `O` has a `[[DateValue]]` internal slot and the value of `[[DateValue]]` is `undefined`, then
   a. Set the `[[DateValue]]` internal slot of `O` to the time value (UTC) identifying the current time.
   b. Return `O`.
5. Else, a. Return the result computed as if by the expression `(new Date()).toString()` where `Date` is this function and `toString` is the standard built-in method `Date.prototype.toString`.

20.3.2.4 new `Date(...) argumentsList)`

When `Date` is called as part of a new expression with argument list `argumentsList` it performs the following steps:

1. Let `F` be the `Date` function object on which the `new` operator was applied.
2. Let `argumentsList` be the `argumentsList` argument of the `[[Construct]]` internal method that was invoked by the `new` operator.
3. Return the result of OrdinaryConstruct (`F`, `argumentsList`).

If `Date` is implemented as an ECMAScript function object, its `[[Construct]]` internal method will perform the above steps.
20.3.3 Properties of the Date Constructor

The value of the [[Prototype]] internal slot of the Date constructor is the Function prototype object (19.2.3).

Besides the length property (whose value is 7), the Date constructor has the following properties:

20.3.3.1 Date.now()

The now function returns a Number value that is the time value designating the UTC date and time of the occurrence of the call to now.

20.3.3.2 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 (including extended years) called out in Date Time String Format (20.3.1.15). 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 (20.3.1.15) and that could not be produced in that implementation by the toString or toUTCString method.

20.3.3.3 Date.prototype

The initial value of Date.prototype is the built-in Date prototype object (20.3.4).

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

20.3.3.4 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.

20.3.3.5 Date[@create] ()
The @@create method of an object F performs the following steps:
1. Let obj be the result of calling OrdinaryCreateFromConstructor(F, "%DatePrototype%", ([DateValue])).
2. Return obj.

The value of the name property of this function is "[Symbol.create]".
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

NOTE [[DateValue]] is initially assigned the value undefined as a flag to indicate that the instance has not yet been initialised by the Date constructor. This flag value is never directly exposed to ECMAScript code; hence implementations may choose to encode the flag in some other manner.

20.3.4 Properties of the Date Prototype Object
The Date prototype object is itself an ordinary object. It is not a Date instance and does not have a [[DateValue]] internal slot.

The value of the [[Prototype]] internal slot of the Date prototype object is the standard built-in Object prototype object (20.3.4).

Unless explicitly defined otherwise, the methods of the Date prototype object defined below are not generic and the this value passed to them must be an object that has a [[DateValue]] internal slot that has been initialised to a time value.

The abstract operation thisTimeValue(value) performs the following steps:
1. If Type(value) is Object and value has a [[DateValue]] internal slot, then
   a. Let n be the Number that is the value of value’s [[DateValue]] internal slot.
   b. If n is not undefined, then return n.
2. Throw a TypeError exception.

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. The phrase “this time value” within the specification of a method refers to the result returned by calling the abstract operation thisTimeValue with the this value of the method invocation passed as the argument.
20.3.4.1 `Date.prototype.constructor`

The initial value of `Date.prototype.constructor` is the built-in `Date` constructor.

20.3.4.2 `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))`.

20.3.4.3 `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))`.

20.3.4.4 `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))`.

20.3.4.5 `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))`.

20.3.4.6 `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))`.

20.3.4.7 `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))`.

20.3.4.8 `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))`.

20.3.4.9 `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))`. 
20.3.4.10 Date.prototype.getTime ( )
1. Return this time value.

20.3.4.11 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.

20.3.4.12 Date.prototype.getUTCDate ( )
1. Let t be this time value.
2. ReturnIfAbrupt(t).
3. If t is NaN, return NaN.
4. Return DateFromTime(t).

20.3.4.13 Date.prototype.getUTCDay ( )
1. Let t be this time value.
2. ReturnIfAbrupt(t).
3. If t is NaN, return NaN.
4. Return WeekDay(t).

20.3.4.14 Date.prototype.getUTCFullYear ( )
1. Let t be this time value.
2. ReturnIfAbrupt(t).
3. If t is NaN, return NaN.
4. Return YearFromTime(t).

20.3.4.15 Date.prototype.getUTCHours ( )
1. Let t be this time value.
2. ReturnIfAbrupt(t).
3. If t is NaN, return NaN.
4. Return HourFromTime(t).

20.3.4.16 Date.prototype.getUTCMilliseconds ( )
1. Let t be this time value.
2. ReturnIfAbrupt(t).
3. If t is NaN, return NaN.
4. Return msFromTime(t).

20.3.4.17 Date.prototype.getUTCMilliseconds ( )
1. Let t be this time value.
2. ReturnIfAbrupt(t).
3. If t is NaN, return NaN.
4. Return MinFromTime(t).
20.3.4.18 Date.prototype.getUTCMonth()

1. Let \( t \) be this time value.
2. ReturnIfAbrupt(\( t \)).
3. If \( t \) is NaN, return NaN.
4. Return MonthFromTime(\( t \)).

20.3.4.19 Date.prototype.getUTCSeconds()

1. Let \( t \) be this time value.
2. ReturnIfAbrupt(\( t \)).
3. If \( t \) is NaN, return NaN.
4. Return SecFromTime(\( t \)).

20.3.4.20 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 slot of this Date object to \( u \).
6. Return \( u \).

20.3.4.21 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 slot of this Date object to \( u \).
8. Return \( u \).

The length property of the setFullYear method is 3.

20.3.4.22 Date.prototype.setHours(hour[, min[, sec[, ms]]])

If min is not specified, this behaves as if min were specified with the value getMinutes().
If sec is not specified, this behaves as if sec were specified with the value getSeconds().
If ms is not specified, this behaves as if ms were specified with the value getMilliseconds().

1. Let \( t \) be the result of LocalTime(this time value).
2. Let \( h \) be ToNumber(hour).
3. If min is not specified, then let \( m \) be MinFromTime(\( t \)); otherwise, let \( m \) be ToNumber(min).
4. If sec is not specified, then let \( s \) be SecFromTime(\( t \)); otherwise, let \( s \) be ToNumber(sec).
5. If ms is not specified, then let \( milli \) be msFromTime(\( t \)); otherwise, let \( milli \) be ToNumber(ms).
6. Let date be MakeDate(Day(\( t \)), MakeTime(h, m, s, milli)).
7. Let \( u \) be TimeClip(UTC(\( date \))).
8. Set the [[DateValue]] internal slot of this Date object to \( u \).
9. Return \( u \).

The length property of the setHours method is 4.
20.3.4.23 Date.prototype.setMilliseconds (ms)

1. Let \( t \) be the result of \( \text{LocalTime}(\text{this time value}) \).
2. Let \( \text{time} \) be \( \text{MakeTime}((\text{HourFromTime}(t), \text{MinFromTime}(t), \text{SecFromTime}(t), \text{ToNumber}(ms))) \).
3. Let \( u \) be \( \text{TimeClip}(\text{UTC}(\text{MakeDate}(\text{Day}(t), \text{time}))) \).
4. Set the \([[\text{DateValue}]]\) internal slot of this Date object to \( u \).
5. Return \( u \).

20.3.4.24 Date.prototype.setMinutes (min [, sec [, ms ]])

If \( \text{sec} \) is not specified, this behaves as if \( \text{sec} \) were specified with the value \( \text{getSeconds()} \).

If \( \text{ms} \) is not specified, this behaves as if \( \text{ms} \) were specified with the value \( \text{getMilliseconds()} \).

1. Let \( t \) be the result of \( \text{LocalTime}(\text{this time value}) \).
2. Let \( m \) be \( \text{ToNumber}(\text{min}) \).
3. If \( \text{sec} \) is not specified, then let \( s \) be \( \text{SecFromTime}(t) \); otherwise, let \( s \) be \( \text{ToNumber}(\text{sec}) \).
4. If \( \text{ms} \) is not specified, then let \( \text{milli} \) be \( \text{msFromTime}(t) \); otherwise, let \( \text{milli} \) be \( \text{ToNumber}(\text{ms}) \).
5. Let \( \text{date} \) be \( \text{MakeDate}(\text{YearFromTime}(t), \text{m}, s, \text{milli}) \).
6. Let \( u \) be \( \text{TimeClip}(\text{UTC}((\text{date})) \).
7. Set the \([[\text{DateValue}]]\) internal slot of this Date object to \( u \).
8. Return \( u \).

The \text{length} property of the \text{setMinutes} method is \( 3 \).

20.3.4.25 Date.prototype.setMonth (month [, date ])

If \( \text{date} \) is not specified, this behaves as if \( \text{date} \) were specified with the value \( \text{getDate()} \).

1. Let \( t \) be the result of \( \text{LocalTime}(\text{this time value}) \).
2. Let \( m \) be \( \text{ToNumber}(\text{month}) \).
3. If \( \text{date} \) is not specified, then let \( \text{dt} \) be \( \text{DateFromTime}(t) \); otherwise, let \( \text{dt} \) be \( \text{ToNumber}(\text{date}) \).
4. Let \( \text{newDate} \) be \( \text{MakeDate}(\text{YearFromTime}(t), m, \text{dt}, \text{TimeWithinDay}(t)) \).
5. Let \( u \) be \( \text{TimeClip}(\text{UTC}((\text{newDate})) \).
6. Set the \([[\text{DateValue}]]\) internal slot of this Date object to \( u \).
7. Return \( u \).

The \text{length} property of the \text{setMonth} method is \( 2 \).

20.3.4.26 Date.prototype.setSeconds (sec [, ms ])

If \( \text{ms} \) is not specified, this behaves as if \( \text{ms} \) were specified with the value \( \text{getMilliseconds()} \).

1. Let \( t \) be the result of \( \text{LocalTime}(\text{this time value}) \).
2. Let \( s \) be \( \text{ToNumber}(\text{sec}) \).
3. If \( \text{ms} \) is not specified, then let \( \text{milli} \) be \( \text{msFromTime}(t) \); otherwise, let \( \text{milli} \) be \( \text{ToNumber}(\text{ms}) \).
4. Let \( \text{date} \) be \( \text{MakeDate}(\text{YearFromTime}(t), \text{MinFromTime}(t), s, \text{milli}) \).
5. Let \( u \) be \( \text{TimeClip}(\text{UTC}((\text{date})) \).
6. Set the \([[\text{DateValue}]]\) internal slot of this Date object to \( u \).
7. Return \( u \).

The \text{length} property of the \text{setSeconds} method is \( 2 \).
20.3.4.27 Date.prototype.setTime (time)

1. Let v be TimeClip(ToNumber(time)).
2. ReturnIfAbrupt(v).
3. Set the [[DateValue]] internal slot of this Date object to v.
4. Return v.

20.3.4.28 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 slot of this Date object to v.
7. Return v.

20.3.4.29 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 slot of this Date object to v.

The length property of the setUTCFullYear method is 3.

20.3.4.30 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 slot of this Date object to v.
10. Return v.

The length property of the setUTCHours method is 4.

20.3.4.31 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 slot of this Date object to v.
6. Return v.

20.3.4.32 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 slot of this Date object to v.

The length property of the setUTCMinutes method is 3.

20.3.4.33 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 slot of this Date object to v.
8. Return v.

The length property of the setUTCMonth method is 2.

20.3.4.34 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(Date(Day(t), MakeTime(HourFromTime(t), MinFromTime(t), s, milli))).
6. Let v be TimeClip(date).
7. Set the [[DateValue]] internal slot of this Date object to v.
8. Return v.

The length property of the setUTCSeconds method is 2.
20.3.4.35 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.

20.3.4.36 Date.prototype.toISOString ( )

This function returns a String value representing the instance in time corresponding to this time value. The format of the String is the Date Time string format defined in 20.3.1.15. All fields are present in the String. The time zone is always UTC, denoted by the suffix Z. If this time value is not a finite Number or if the year is not a value that can be represented in that format (if necessary using extended year format), a RangeError exception is thrown.

20.3.4.37 Date.prototype.toJSON ( key )

This function provides a String representation of a Date object for use by JSON.stringify (24.3.2).

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.

20.3.4.38 Date.prototype.toLocaleDateString (reserved1=undefined, reserved2=undefined)

An ECMAScript implementation that includes the ECMA-402 International API must implement the Date.prototype.toLocaleDateString method as specified in the ECMA-402 specification. If an ECMAScript implementation does not include the ECMA-402 API the following specification of the toLocaleDateString method is used.

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.

The meaning of the optional parameters to this method are defined in the ECMA-402 specification; implementations that do not include ECMA-402 support must not use those parameter position for anything else.

The length property of the toLocaleDateString method is 0.

20.3.4.39 Date.prototype.toLocaleString (reserved1=undefined, reserved2=undefined)

An ECMAScript implementation that includes the ECMA-402 International API must implement the Date.prototype.toLocaleDateString method as specified in the ECMA-402 specification. If an ECMAScript implementation does not include the ECMA-402 API the following specification of the toLocaleDateString method is used.
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.

The meaning of the optional parameters to this method are defined in the ECMA-402 specification; implementations that do not include ECMA-402 support must not use those parameter position for anything else.

The length property of the `toLocaleString` method is 0.

### 20.3.4.40 `Date.prototype.toLocaleTimeString (reserved1=undefined, reserved2=undefined)`

An ECMAScript implementation that includes the ECMA-402 International API must implement the `Date.prototype.toLocaleTimeString` method as specified in the ECMA-402 specification. If an ECMAScript implementation does not include the ECMA-402 API the following specification of the `toLocaleTimeString` method is used.

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.

The meaning of the optional parameters to this method are defined in the ECMA-402 specification; implementations that do not include ECMA-402 support must not use those parameter position for anything else.

The length property of the `toLocaleTimeString` method is 0.

### 20.3.4.41 `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 whose milliseconds amount is zero, the result of `Date.parse(d.toString())` is equal to `d.valueOf()`. See 20.3.3.2.

### 20.3.4.42 `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.

### 20.3.4.43 `Date.prototype.toUTCString ()`

This function returns a String value. The contents of the String are implementation-dependent, but are intended to represent this time value 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 20.3.1.15. 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 20.3.1.15 but with a space rather than a “;” used to separate the date and time elements.

### 20.3.4.44 `Date.prototype.valueOf ()`

The `valueOf` function returns a Number, which is this time value.
20.3.4.45 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 objects 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 value.
2. If Type(O) is not Object, then throw a TypeError exception.
3. If hint is the string value "string" or the string value "default", then
   a. Let tryFirst be "string".
4. Else if hint is the string value "number", then
   a. Let tryFirst be "number".
5. Else, throw a TypeError exception.
6. Return the result of OrdinaryToPrimitive(O, tryFirst).

The value of the name property of this function is "[[Symbol.toPrimitive]]".

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

20.3.5 Properties of Date Instances

Date instances are ordinary objects that inherit properties from the Date prototype object. Date instances also have a [[DateValue]] internal slot. The [[DateValue]] internal slot is the time value represented by this Date object.

21 Text Processing

21.1 String Objects

21.1.1 The String Constructor

The String constructor is the %String% intrinsic object and the initial value of the String property of the global object. When String is called as a function rather than as a constructor, it performs a type conversion. However, if the this value passed in the call is an Object with an uninitialised [[StringData]] internal slot, it initialises the this value using the argument value. This permits String to be used both to perform type conversion and to perform constructor instance initialisation.

The String constructor is designed to be subclassable. It may be used as the value of an extends clause of a class declaration. Subclass constructors that intended to inherit the specified String behaviour must include a super call to the String constructor to initialise the [[StringData]] state of subclass instances.

21.1.1.1 String (value = "")

When String is called with argument value, the following steps are taken:

1. Let O be the this value.
2. If no arguments were passed to this function invocation, then let s be " ".
3. Else, let s be ToString(value).
4. ReturnIfAbrupt(s).
5. If Type(O) is Object and O has a [[StringData]] internal slot and the value of [[StringData]] is undefined, then
   a. Let length be the number of code unit elements in s.
   b. Let status be the result of DefinePropertyOrThrow(O, "length", PropertyDescriptor{[[Value]]: length, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}).
   c. ReturnIfAbrupt(status).
d. Set the value of O's [[StringData]] internal slot to s.
e. Return O.

6. Return s.

The length property of the String function is 1.

21.1.1.2 new String ( ... argumentsList )

When String is called as part of a new expression, it initialises a newly created exotic String object:

1. Let F be the String function object on which the new operator was applied.
2. Let argumentsList be the argumentsList argument of the [[Construct]] internal method that was invoked by the new operator.
3. Return the result of OrdinaryConstruct (F, argumentsList).

If String is implemented as an ECMAScript function object, its [[Construct]] internal method will perform the above steps.

21.1.2 Properties of the String Constructor

The value of the [[Prototype]] internal slot of the String constructor is the standard built-in Function prototype object (19.2.3).

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

21.1.2.1 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.

21.1.2.2 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

Commented [AWB9102]: Should we provide fromCodeUnit as an alias for this property and label fromCharCode as obsolete.
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 (10.1.1) 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.

21.1.2.3 String.prototype

The initial value of the String.prototype is the standard built-in String prototype object (21.1.3).

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

21.1.2.4 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 ToLength(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
   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 (12.2.7). 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.

21.1.2.5 String[ @@create ]()

The @@create method of an object performs the following steps:
1. Let \( F \) be the \( \text{this} \) value.
2. Let \( \text{proto} \) be the result of \text{GetPrototypeFromConstructor}(\( F \), "\%StringPrototype\%").
3. ReturnIfAbrupt(\( \text{proto} \)).
4. Let \( \text{obj} \) be the result of calling \text{StringCreate}(\( \text{proto} \)).
5. Return \( \text{obj} \).

The value of the \text{name} property of this function is "\%Symbol.create\%".

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

**NOTE**

[[StringData]] is initially assigned the value \text{undefined} as a flag to indicate that the instance has not yet been initialised by the String constructor. This flag value is never directly exposed to ECMA-Script code; hence implementations may choose to encode the flag in some other manner.

### 21.1.3 Properties of the String Prototype Object

The String prototype object is itself an ordinary object. It is not a String instance and does not have a [[StringData]] internal slot.

The value of the [[Prototype]] internal slot of the String prototype object is the standard built-in Object prototype object (19.1.3).

Unless explicitly stated otherwise, the methods of the String prototype object defined below are not generic and the this value passed to them must be either a String value or an object that has a [[StringData]] internal slot that has been initialised to a String value.

The abstract operation \text{thisStringValue(}value\text{)} performs the following steps:

1. If \text{Type(}value\text{)} is String, return \( \text{value} \).
2. If \text{Type(}value\text{)} is Object and \( \text{value} \) has a [[StringData]] internal slot, then
   a. Let \( s \) be the value of \( \text{value} \)'s [[StringData]] internal slot.
   b. If \( s \) is not \text{undefined}, then return \( s \).
3. Throw a \text{TypeError} exception.

The phrase "this String value" within the specification of a method refers to the result returned by calling the abstract operation \text{thisStringValue(}\text{this} \text{value of the method invocation}\text{)} passed as the argument.

#### 21.1.3.1 String.prototype.charAt(pos)

**NOTE**

Returns a single element String containing the code unit at element position \( \text{pos} \) 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 \( \text{pos} \) is a value of Number type that is an integer, then the result of \( x . \text{charAt(}\text{pos}\text{)} \) is equal to the result of \( x . \text{substring(}\text{pos, pos+1}\text{)} \).

When the charAt method is called with one argument \( \text{pos} \), the following steps are taken:

1. Let \( O \) be CheckObjectCoercible(\text{this value}).
2. Let \( S \) be ToString(\( O \)).
3. ReturnIfAbrupt(\( S \)).
4. Let \( \text{position} \) be ToInteger(\( \text{pos} \)).
5. ReturnIfAbrupt(\( \text{position} \)).
6. Let \( \text{size} \) be the number of elements in \( S \).
7. If \( \text{position} < 0 \) or \( \text{position} \geq \text{size} \), 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.

21.1.3.2 String.prototype.charAt (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 charAt method is called with one argument pos, the following steps are taken:

1. Let O be CheckObjectCoercible(this value).
2. Let S be ToString(O).
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 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.

21.1.3.3 String.prototype.codePointAt (pos)

NOTE Returns a nonnegative integer Number less than 1114112 (0x110000) 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 undefined. 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. Let O be CheckObjectCoercible(this value).
2. Let S be ToString(O).
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.

21.1.3.4 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.

Commented [AWB9105]: Do we also need UnicodeCharAt that returns a string of length 1 or 2?
The following steps are taken:

1. Assert: args is a well-formed rest parameter object.
2. Let O be CheckObjectCoercible(this value).
3. Let S be ToString(O).
4. ReturnIfAbrupt(S).
5. Let args be a 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 1 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.

21.1.3.5 String.prototype.constructor

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

21.1.3.6 String.prototype.contains (searchString, position = 0 )

The contains method takes two arguments, searchString and position, and performs the following steps:

1. Let O be CheckObjectCoercible(this value).
2. Let S be ToString(O).
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 elements 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 searchString, 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, return 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.

21.1.3.7 String.prototype.endsWith (searchString [, endPosition] )

The following steps are taken:
0. Let \( O \) be `CheckObjectCoercible(this value).`
1. Let \( S \) be `ToString(\( O \)).`
2. ReturnIfAbrupt(\( S \)).
3. If Type(\( searchString \)) is `Object` and HasProperty(\( searchString, @@isRegExp \)) is `true`, then
   a. Throw a `TypeError` exception.
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 \( searchStr \).  
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 \( searchStr \), 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, return `false`.

NOTE 2 Throwing an exception if the first argument is a RegExp is specified in order to allow future editions to define `extends` that allow such argument values.

NOTE 3 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.

21.1.3.8 `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. Let \( O \) be `CheckObjectCoercible(this value).`
2. Let \( S \) be `ToString(\( O \)).`
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 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.
21.1.3.9 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. Let O be CheckObjectCoercible(this value).
2. Let S be ToString(O).
3. ReturnIfAbrupt(S).
4. Let searchString 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 searchString.
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 searchString; 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.

21.1.3.10 String.prototype.localeCompare (that, reserved1=undefined, reserved2=undefined)

An ECMAScript implementation that includes the ECMA-402 International API must implement the localeCompare method as specified in the ECMA-402 specificat.
The `localeCompare` method, if considered as a function of two arguments `this` and `that`, is a consistent comparison function (as defined in 22.1.3.24) 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. If the implementation performs language-sensitive comparisons it must 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.

The `localeCompare` method itself is not directly suitable as an argument to `Array.prototype.sort` because the latter requires a function of two arguments.

The `length` property of the `localeCompare` method is 1.

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. This function must treat Strings that are canonically equivalent according to the Unicode standard as identical. It is also recommended that this function not honour Unicode compatibility equivalences or decompositions. For a definition and discussion of canonical equivalence see the Unicode Standard, chapters 2 and 3, as well as Unicode Annex #15, Unicode Normalization Forms and Unicode Technical Note #5 Canonical Equivalence in Applications. Also see Unicode Technical Standard #10, Unicode Collation Algorithm.

NOTE 3 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.

21.1.3.11 `String.prototype.match (regexp)`

When the `match` method is called with argument `regexp`, the following steps are taken:

1. Let `O` be `CheckObjectCoercible(this value).`
2. Let `S` be `ToString(O).`
3. ReturnIfAbrupt(S).
4. If `Type(regexexp) is Object and HasProperty(regexexp, @@isRegExp) is true`, then let `rx` be `regexexp`.
5. Else, let `rx` be the result of the abstract operation `RegExpCreate (21.2.3.3)` with arguments `regexp` and `undefined`.
6. ReturnIfAbrupt(rx).
7. Return the result of `Invoke(rx, "match", (S)).`.

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.

21.1.3.12 `String.prototype.normalize (form = "NFC")`

When the `normalize` method is called with one argument `form`, the following steps are taken:

1. Let `O` be `CheckObjectCoercible(this value).`
2. Let `S` be `ToString(O).`
3. ReturnIfAbrupt(S).
4. If `form` is not provided or `undefined` let `form` be "NFC".
5. Let `f` be `ToString(form).`
6. ReturnIfAbrupt(f).
7. If `f` is not one of "NFC", "NFD", "NFKC", or "NFKD", then throw a `RangeError` Exception.
8. Let `ns` be the String value is the result of normalizing `S` into the normalization form named by `f` as specified in `Unicode Standard Annex #15, UnicodeNormalization Forms`.
9. Return `ns`.

The `length` property of the `normalize` method is 0.
NOTE The `normalize` 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.

21.1.3.13 String.prototype.repeat (count)

The following steps are taken:

1. Let O be CheckObjectCoercible(this value).
2. Let S be ToString(O).
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 +\infty, then throw a RangeError exception.
8. Let T be a String value that is made from n copies of S appended together. If n is 0, T is the empty String.
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.

21.1.3.14 String.prototype.replace (searchValue, replaceValue)

When the replace method is called with arguments searchValue and replaceValue the following steps are taken:

1. Let O be CheckObjectCoercible(this value).
2. Let string be ToString(O).
3. ReturnIfAbrupt(string).
4. If Type(searchValue) is Object and HasProperty(searchValue, @@isRegExp) is true, then
   a. Return the result of Invoke(searchValue, "replace", string, replaceValue).
5. Let searchString be ToString(searchValue).
6. ReturnIfAbrupt(searchString).
7. Search string for the first occurrence of searchString and let pos be the index position within string of the first code unit of the matched substring and let matched be searchString. If no occurrences of searchString were found, return string.
8. If IsCallable(replaceValue) is true, then
   a. Let replValue be the result of calling the [[Call]] internal method of replaceValue passing undefined as the this value and a List containing matched, pos, and string as the argument list.
   b. Let replStr be ToString(replValue).
   c. ReturnIfAbrupt(replStr).
9. Else,
   a. Let captures be an empty List.
   b. Let replStr be the result of the abstract operation GetReplaceSubstitution(matched, string, pos, captures).
10. Let tailPos be pos + the number of code units in matched.
11. Let newString be the String formed by concatenating the first pos code units of string, replStr, and the trailing substring of string starting at index tailPos. If pos is 0, the first element of the concatenation will be the empty String.
12. Return newString.

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.
21.1.3.14.1 Runtime Semantics: GetReplaceSubstitution Abstract Operation

The abstract operation GetReplaceSubstitution(matched, string, position, captures) performs the following steps:

1. Assert: Type(matched) is String.
2. Let matchLength be the number of code units in matched.
3. Assert: Type(string) is String.
4. Let stringLength be the number of code units in string.
5. Assert: position is a non-negative integer.
7. Assert: captures is a possibly empty List of Strings.
8. Let tailPos be position + matchLength.
9. Let m be the number of elements in captures.
10. Let result be a String value derived from matched by replacing code unit elements in matched 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.
11. Return result.

Table 32 — Replacement Text Symbol Substitutions

<table>
<thead>
<tr>
<th>Code units</th>
<th>Unicode Characters</th>
<th>Replacement text</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0024, 0x0024</td>
<td>$</td>
<td>matched</td>
</tr>
<tr>
<td>0x0024, 0x0026</td>
<td>%</td>
<td>position is 0, the replacement is the empty String. Otherwise the replacement is the substring of string that starts at index 0 and whose last code unit is at index position + 1.</td>
</tr>
<tr>
<td>0x0024, 0x0027</td>
<td>^</td>
<td>tailPos ≤ stringLength, the replacement is the empty String. Otherwise the replacement is the substring of string that starts at index tailPos and contains to the end of string.</td>
</tr>
<tr>
<td>0x0024, N where $0x0031 ≤ N ≤ 0x0039</td>
<td>\n</td>
<td>\n</td>
</tr>
<tr>
<td>0x0024, N where $0x0030 ≤ N ≤ 0x0039</td>
<td>\n</td>
<td>\n</td>
</tr>
<tr>
<td>0x0024</td>
<td>\n</td>
<td>\n</td>
</tr>
</tbody>
</table>

21.1.3.15 String.prototype.search (regexp)

When the search method is called with argument regexp, the following steps are taken:

1. Let O be CheckObjectCoercible(this value).
2. Let string be ToString(O).
3. ReturnIfAbrupt(string).
4. If Type(regexp) is Object and HasProperty(regexp, @@isRegExp) is true, then
   a. Let rx be regexp;
5. Else,
   a. Let rx be the result of the abstract operation RegExpCreate (21.2.3.3) with arguments regexp and undefined.
6. ReturnIfAbrupt(rx).
7. Return the result of Invoke(rx, "search", (string)).

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.
21.1.3.16 String.prototype.slice (start, end)

The \texttt{slice} method takes two arguments, \texttt{start} and \texttt{end}, and returns a substring of the result of converting this object to a String, starting from element position \texttt{start} and running to, but not including, element position \texttt{end} (or through the end of the String if \texttt{end} is \texttt{undefined}). If \texttt{start} is negative, it is treated as \texttt{sourceLength}+\texttt{start} where \texttt{sourceLength} is the length of the String. If \texttt{end} is negative, it is treated as \texttt{sourceLength}+\texttt{end} where \texttt{sourceLength} is the length of the String. The result is a String value, not a String object. The following steps are taken:

1. Let \( O \) be \texttt{CheckObjectCoercible} this value.
2. Let \( S \) be \texttt{ToString}(\( O \)).
3. ReturnIfAbrupt(\( S \)).
4. Let \( \text{len} \) be the number of elements in \( S \).
5. Let \( \text{intStart} \) be \texttt{ToInteger} \( \text{start} \).
6. If \( \text{end} \) is \texttt{undefined}, let \( \text{intEnd} \) be \( \text{len} \); else let \( \text{intEnd} \) be \texttt{ToInteger} \( \text{end} \).
7. If \( \text{intStart} \) is negative, let \( \text{from} \) be \( \max(\text{len}+\text{intStart},0) \); else let \( \text{from} \) be \( \min(\text{intStart},\text{len}) \).
8. If \( \text{intEnd} \) is negative, let \( \text{to} \) be \( \max(\text{len}+\text{intEnd},0) \); else let \( \text{to} \) be \( \min(\text{intEnd},\text{len}) \).
9. Let \( \text{span} \) be \( \max(\text{to}–\text{from},0) \).
10. Return a String value containing \( \text{span} \) consecutive elements from \( S \) beginning with the element at position \( \text{from} \).

The \texttt{length} property of the \texttt{slice} method is 2.

**NOTE** The \texttt{slice} function is intentionally generic; it does not require that its \texttt{this} value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

21.1.3.17 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 \texttt{separator}; these occurrences are not part of any substring in the returned array, but serve to divide up the String value. The value of \texttt{separator} may be a String of any length or it may be a RegExp object.

When the \texttt{split} method is called, the following steps are taken:

1. Let \( O \) be \texttt{CheckObjectCoercible} this value.
2. ReturnIfAbrupt(\( O \)).
3. If Type(\( separator \)) is Object and HasProperty(\( separator \), \texttt{@isRegExp}) is \texttt{true}, then:
   a. Return the result of Invoke(\( separator \), \texttt{"split"}, (\( O \), \( limit \))
4. Let \( S \) be \texttt{ToString}(\( O \)).
5. ReturnIfAbrupt(\( S \)).
6. Let \( A \) be the result of the abstract operation ArrayCreate with argument 0.
7. Let \( \text{lengthA} \) be 0.
8. If \( \text{limit} \) is \texttt{undefined}, let \( \text{lim} = 2^{32}–1 \); else let \( \text{lim} = \text{ToLength} \( \text{limit} \).
9. Let \( s \) be the number of elements in \( S \).
10. Let \( p = 0 \).
11. Let \( R \) be \texttt{ToString} \( \text{separator} \).
12. ReturnIfAbrupt(\( R \)).
13. If \( \text{lim} = 0 \), return \( A \).
14. If \( \text{separator} \) is \texttt{undefined}, then:
   a. Call CreateDataProperty(\( A \), \texttt{"0"}, \( S \)).
   b. Assert: The above call will never result in an abrupt completion.
   c. Return \( A \).
15. If \( z = 0 \), then:
   a. Let \( z \) be the result of SplitMatch(\( S \), 0, \( R \)).
   b. If \( z \) is not \texttt{false}, return \( A \).
c. Call CreateDataProperty(A, "0", S).
d. Assert: The above call will never result in an abrupt completion.
e. Return A.

16. Let q = p.
17. Repeat, while q ≠ s
   a. Let e be the result of SplitMatch(S, q, R).
   b. If e is false, then let q = q+1.
   c. Else e is an integer index into S.
      i. If e = p, then let q = q+1.
      ii. Else e ≠ p,
           1. Let T be a String value equal to the substring of S consisting of the code units at positions p (inclusive) through q (exclusive).
           2. Call CreateDataProperty(A, ToString(lengthA), T).
           3. Assert: The above call will never result in an abrupt completion.
           4. Increment lengthA by 1.
           5. If lengthA = lim, return A.
           7. Let q = p.

18. Let T be a String value equal to the substring of S consisting of the code units at positions p (inclusive) through s (exclusive).
19. Call CreateDataProperty(A, ToString(lengthA), T).
20. Assert: The above call will never result in an abrupt completion.
21. Return A.

NOTE 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 ["ab","in"]. 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\<\=\bold\>B\+\"and\"\<\=\<\CODE\>\"\>\"\CODE\>\".split(/<\(/)??[^<>]+</>)/

evaluates to the array


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.

21.1.3.17.1 Runtime Semantics: SplitMatch Abstract Operation

The abstract operation SplitMatch takes three parameters, a String S, an integer q, and a String R, and performs the following in order to return either false or the end index of a match:

1. Type(R) must be String. Let r be the number of code units in R.
2. Let s be the number of code units in S.
3. If q+r > s then return false.
4. 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 false.
5. Return q+r.
The `length` property of the `split` method is 2.

**NOTE** 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.

## 21.1.3.18 String.prototype.startsWith (searchString [, position ])

The following steps are taken:

1. Let `O` be `CheckObjectCoercible(this value).`
2. Let `S` be `ToString(O).`
3. ReturnIfAbrupt(`S`).
4. If `Type(searchString)` is Object and HasProperty(`searchString`, `@@isRegExp`) is `true`, then
   1. Throw a `TypeError` exception.
5. Let `searchStr` be `ToString(searchString).`
6. ReturnIfAbrupt(`searchStr`).
7. Let `pos` be `ToInteger(position).` (If `position` is `undefined`, this step produces the value `0`).
8. ReturnIfAbrupt(`pos`).
9. Let `len` be the number of elements in `S`.
10. Let `start` be `min(max(pos, 0), len)`. (If `pos` is `undefined`, this step produces the value `0`).
11. Let `searchLength` be the number of elements in `searchStr`.
12. If `searchLength + start` is greater than `len`, return `false`.
13. If the `searchLength` sequence of elements of `S` starting at `start` is the same as the full element sequence of `searchStr`, return `true`.
14. 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** Throwing an exception if the first argument is a RegExp is specified in order to allow future editions to define extends that allow such argument values.

**NOTE 3** 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.

## 21.1.3.19 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. Let `O` be `CheckObjectCoercible(this value).`
2. Let `S` be `ToString(O).`
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 `max(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.

21.1.3.20 `String.prototype.toLocaleLowerCase()`

This function interprets a string value as a sequence of code points, as described in 6.1.4.

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.

21.1.3.21 `String.prototype.toLocaleUpperCase()`

This function interprets a string value as a sequence of code points, as described in 6.1.4.

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.

21.1.3.22 `String.prototype.toLowerCase()`

This function interprets a string value as a sequence of code points, as described in 6.1.4. The following steps are taken:

1. Let `O` be `CheckObjectCoercible(this value)`.
2. Let `S` be `ToString(O)`.
3. ReturnIfAbrupt(S).
4. Let `cpList` be a List containing in order the code points as defined in 6.1.4 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 (10.1.1) of `c`.
8. Let `L` be a String whose elements are, in order, the elements of `cuList`.

© Ecma International 2009 – All rights reserved
The result must be derived according to the locale-insensitive case mappings in the Unicode Character Database (this explicitly includes not only the UnicodeData.txt file, but also all locale-insensitive mappings in 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 \texttt{toUpperCase} and \texttt{toLowerCase} have context-sensitive behaviour, the functions are not symmetrical. In other words, \texttt{s.toUpperCase().toLowerCase()} is not necessarily equal to \texttt{s.toLowerCase()}.

NOTE 2 The \texttt{toLowerCase} function is intentionally generic; it does not require that its \texttt{this} value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

21.1.3.23 \texttt{String.prototype.toString ( )}

When the \texttt{toString} method is called, the following steps are taken:

1. Let \texttt{s} be \texttt{this} \texttt{StringValue} (\texttt{this} value).
2. Return \texttt{s}.

NOTE For a String object, the \texttt{toString} method happens to return the same thing as the \texttt{valueOf} method.

21.1.3.24 \texttt{String.prototype.toUpperCase ( )}

This function interprets a string value as a sequence of code points, as described in 6.1.4.

This function behaves in exactly the same way as \texttt{String.prototype.toLowerCase}, except that code points are mapped to their uppercase equivalents as specified in the Unicode Character Database.

NOTE The \texttt{toUpperCase} function is intentionally generic; it does not require that its \texttt{this} value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

21.1.3.25 \texttt{String.prototype.trim ( )}

This function interprets a string value as a sequence of code points, as described in 6.1.4.

The following steps are taken:

1. Let \texttt{O} be \texttt{CheckObjectCoercible} (\texttt{this} value).
2. Let \texttt{S} be \texttt{ToString} (\texttt{O}).
3. ReturnIfAbrupt (\texttt{S}).
4. Let \texttt{T} be a String value that is a copy of \texttt{S} with both leading and trailing white space removed. The definition of white space is the union of \texttt{WhiteSpace} and \texttt{LineTerminator}. When determining whether a Unicode code point is in Unicode general category "Zs", code unit sequences are interpreted as UTF-16 encoded code point sequences as specified in 6.1.4.
5. Return \texttt{T}.

NOTE The \texttt{trim} function is intentionally generic; it does not require that its \texttt{this} value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

21.1.3.26 \texttt{String.prototype.valueOf ( )}

When the \texttt{valueOf} method is called, the following steps are taken:

1. Let \texttt{s} be \texttt{this} \texttt{StringValue} (\texttt{this} value).
2. Return \texttt{s}.
21.1.3.27 String.prototype [ @@iterator ]

When the @@iterator method is called it returns an Iterator object (25.1.2) that iterates over the code points of a String value, returning each code point as a String value. The following steps are taken:

The following steps are taken:

1. Let O be CheckObjectCoercible(this value).
2. Let S be ToString(O).
3. ReturnIfAbrupt(S).
4. Return the result of calling the CreateStringIterator abstract operation with argument S.

The value of the name property of this function is "[[Symbol.iterator]]".

21.1.4 Properties of String Instances

String instances are String exotic objects and have the internal methods specified for such objects. String instances inherit properties from the String prototype object. String instances also have a [[StringData]] internal slot.

String instances have a length property, and a set of enumerable properties with integer indexed names.

21.1.4.1 length

The number of elements in the String value represented by this String object.

Once a String object is initialised, this property is unchanging. It has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

21.1.5 String Iterator Objects

An String Iterator is an object, that represents a specific iteration over some specific String instance object. There is not a named constructor for String Iterator objects. Instead, String iterator objects are created by calling certain methods of String instance objects.

21.1.5.1 CreateStringIterator Abstract Operation

Several methods of String objects return Iterator objects. The abstract operation CreateStringIterator with argument string is used to create such iterator objects. It performs the following steps:

1. Let s be the result of calling ToString(string).
2. ReturnIfAbrupt(s).
3. Let iterator be the result of ObjectCreate(%StringIteratorPrototype%, ("[[Iterated]]", [[StringIteratorNextIndex]])).
4. Set iterator’s [[IteratedString]] internal slot to s.
5. Set iterator’s [[StringIteratorNextIndex]] internal slot to 0.
6. Return iterator.

21.1.5.2 The %StringIteratorPrototype% Object

All String Iterator Objects inherit properties from the %StringIteratorPrototype% intrinsic object. The %StringIteratorPrototype% object is an ordinary object and its [[Prototype]] internal slot is the %ObjectPrototype% intrinsic object. In addition, %StringIteratorPrototype% has the following properties:

21.1.5.2.1 %StringIteratorPrototype%.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 slots of a String Iterator Instance \( 21.1.5.3 \), throw a TypeError exception.
4. Let \( s \) be the value of the \([\text{Iterated String}]\) internal slot of \( O \).
5. If \( s \) is undefined, then return CreateIterResultObject(undefined, true).
6. Let position be the value of the \([\text{String Iterator Next Index}]\) internal slot of \( O \).
7. Let len be the number of elements in \( s \).
8. If position \( \geq \) len, then
   a. Set the value of the \([\text{Iterated String}]\) internal slot of \( O \) to undefined.
   b. Return CreateIterResultObject(undefined, true).
9. Let first be the code unit value of the element at index position in \( s \).
10. If first < 0xD800 or first > 0xDBFF or position+1 = len, then let resultString be the string consisting of the single code unit first.
11. Else,
    a. Let second be the code unit value of the element at index position+1 in the String \( S \).
    b. If second < 0xDC00 or second > 0xDFFF, then let resultString be the string consisting of the single code unit first.
    c. Else, let resultString be the string consisting of the code unit first followed by the code unit second.
12. Let resultSize be the number of code units in resultString.
13. Set the value of the \([\text{String Iterator Next Index}]\) internal slot of \( O \) to position+ resultSize.
14. Return CreateIterResultObject(resultString, false).

21.1.5.2.2 \%StringIteratorPrototype\%[@@iterator]()

The following steps are taken:
1. Return the this value.

The value of the name property of this function is "[Symbol.iterator]".

21.1.5.2.3 \%StringIteratorPrototype\%[@@toStringTag]

The initial value of the @@toStringTag property is the string value "String Iterator".

21.1.5.3 Properties of String Iterator Instances

String Iterator instances are ordinary objects that inherit properties from the \%StringIteratorPrototype\% intrinsic object. String Iterator instances are initially created with the internal slots listed in Table 35.

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Iterated String]</td>
<td>The String value whose elements are being iterated.</td>
</tr>
<tr>
<td>[String Iterator Next Index]</td>
<td>The integer index of the next string index to be examined by this iteration.</td>
</tr>
</tbody>
</table>

21.2 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.
21.2.1 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

\[ \text{Pattern} \rightarrow \text{Disjunction} \]
\[ \text{Disjunction} \rightarrow \text{Alternative} \text{Disjunction} \]
\[ \text{Alternative} \rightarrow \text{Term} \text{Alternative} | \text{Disjunction} \text{Alternative} | \text{Term} \]
\[ \text{Term} \rightarrow \text{Assertion} \text{Term} \]
\[ \text{Assertion} \rightarrow \text{Atom} \text{Quantifier} \]
\[ \text{Quantifier} \rightarrow \text{QuantifierPrefix} \text{QuantifierPrefix} ? \]
\[ \text{QuantifierPrefix} \rightarrow \ast | \plus | \? \text{DecimalDigits} | \text{DecimalDigits} , \text{DecimalDigits} \]
\[ \text{Atom} \rightarrow \text{PatternCharacter} \]
\[ \\text{PatternCharacter} \rightarrow \text{SourceCharacter} \] but not SyntaxCharacter

\[ \\text{SyntaxCharacter} \rightarrow \text{one of}: ^ \ \$ \ \. \ \* \ \+ \ \? \ ( \ ) \ [ ] \ { } \ | \]

PatternCharacter :: SourceCharacter but not SyntaxCharacter
AtomEscape::
  DecimalEscape
  CharacterEscape
  CharacterClassEscape

CharacterEscape::
  ControlEscape
  ControlLetter
  HexEscapeSequence
  RegExpUnicodeEscapeSequence
  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

RegExpUnicodeEscapeSequence::
  \u{Hex4Digits}

LeadSurrogate::
  Hex4Digits [match only if the CV of Hex4Digits is in the inclusive range 0xD800 to 0xDBFF]

TailSurrogate::
  Hex4Digits [match only if the CV of Hex4Digits is in the inclusive range 0xDC00 to 0xDFFF]

IdentityEscape::
  \u{SourceCharacter} but not IdentifierPart
  \u{<ZWJ>}
  \u{<ZWNJ>}

DecimalEscape::
  DecimalIntegerLiteral [lookahead \d DecimalDigit]

CharacterClassEscape::
  one of
    d D s S w W

CharacterClass::
  \[ lookahead \s ClassRange \]
  \^ ClassRange

ClassRange::
  [empty]
  NonemptyClassRange

NonemptyClassRange::
  ClassAtom
  ClassAtom NonemptyClassRangeNoDash
  ClassAtom ClassRange
  ClassAtom ClassRange ClassRange

348 © Ecma International 2013
NonemptyClassRangesNoDash;::
ClassAtom
ClassAtomNoDash
NonemptyClassRangesNoDash
ClassAtomNoDash - ClassAtom
ClassRanges

ClassAtomNoDash;::
-
ClassAtomNoDash

ClassAtomNoDash;::
SourceCharacter but not one of \ or } or -
\ ClassEscape

ClassEscape;::
DecimalEscape
CharacterEscape
CharacterClassEscape

21.2.2 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 [[RegExpMatcher]] internal slot.

A Pattern is either a BMP pattern or a Unicode pattern depending upon whether or not its associated flags contain a "u". A BMP pattern matches against a String interpreted as consisting of a sequence of Unicode code units. A Unicode pattern matches against a String interpreted as consisting of Unicode code points encoded using UTF-16. In the context of describing the behaviour of a BMP pattern “character” means a Unicode code unit. In the context of describing the behaviour of a Unicode pattern “character” means a Unicode code point. In either context, “character value” means the numeric value of the code unit or code point.

The semantics of Pattern is defined as if a Pattern was a List of SourceCharacter values. Each SourceCharacter corresponding to a Unicode code point.

Non-BMP characters are UTF-16 encoded. The single character MUSICAL SYMBOL G CLEF pattern, expressed as a string value, is a String of length 2 whose elements were the code units 0xD834 and 0xDDE. So no further translation of the string would be necessary to process it as a BMP pattern consisting of two pattern characters. However, to process it as a Unicode pattern the string value must treated as if it was UTF-16 decoded into a List consisting of a single pattern character, the code unit U+1D11E.

An implementation may not actually perform such translations to or from UTF-16, but the semantics of this specification requires that the result of pattern matching be as if such translations were performed.

21.2.2.1 Notation

The descriptions below use the following variables:

- **Input** is a List consisting of all of the characters, in order, of the String being matched by the regular expression pattern. Each character is either a code units or a code points, depending upon the kind of pattern involved. The notation input[n] means the nᵗʰ character of input, where n can range between 0 (inclusive) and InputLength (exclusive).
- `InputLength` is the number of characters in `Input`.
- `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 `true` if the RegEx object's `[[OriginalFlags]]` internal slot contains "i" and otherwise is `false`.
- `Multiline` is `true` if the RegEx object's `[[OriginalFlags]]` internal slot contains "m" and otherwise is `false`.
- `Unicode` is `true` if the RegEx object's `[[OriginalFlags]]` internal slot contains "u" and otherwise is `false`.

Furthermore, the descriptions below use the following internal data structures:

- A ` CharSet` is a mathematical set of characters, either code units or code points depending up the state of the `Unicode` flag. "All characters" means either all code unit values or all code point values also depending upon the state if `Unicode`.
- A `State` is an ordered pair \((endIndex, captures)\) where \(endIndex\) is an integer and \(captures\) is a List 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 List 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 refers to 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 `Input`, 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 `Input`, 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 `Input` 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.
21.2.2 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. If Unicode is true, then let Input be a List of consisting of the sequence of code points of str interpreted as a UTF-16 encoded Unicode string. Otherwise, let Input be a List of consisting the sequence of code units that are the elements of str. Input will be used throughout the algorithms in 21.2.2. Each element of Input is considered to be a character.
   2. Let listIndex be the index into Input of the character that was obtained from element index of str.
   3. Let InputLength be the number of character contained in Input. This variable will be used throughout the algorithms in 21.2.2.
   4. Let r be a Continuation that always returns its State argument as a successful MatchResult.
   5. Let cap be a List of NcapturingParens undefined values, indexed 1 through NcapturingParens.
   6. Let x be the State (listIndex, cap).
   7. 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 21.2.2 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).

21.2.2.3 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,

/\bab\/.exec("abc")

returns the result "a" and not "ab". Moreover,

/\(a|\ab\)\(\{c\}\(bc\)\)/.exec("abc")

returns the array

["abc", "a", "ab", undefined, "bc", undefined, "bc"]

and not

["abc", "ab", undefined, "ab", "c", "c", undefined]

© Ecma International 2009 – All rights reserved
21.2.2.4 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 Input. 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`.

21.2.2.5 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.

21.2.2.5.1 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 \( \text{min} \) is zero then let \( \text{min}^2 \) be zero; otherwise let \( \text{min}^2 \) be \( \text{min} - 1 \).
3. If \( \text{max} = \infty \), then let \( \text{max}^2 \) be \( \infty \); otherwise let \( \text{max}^2 \) be \( \text{max} - 1 \).
4. Call RepeatMatcher\( (m, \text{min}^2, \text{max}^2, \text{greedy}, y, c, \text{parenIndex}, \text{parenCount}) \) and return its result.

Let \( \text{cap} \) be a fresh copy of \( x \)'s captures List.

For every integer \( k \) that satisfies \( \text{parenIndex} < k \) and \( k \leq \text{parenIndex} + \text{parenCount} \), set \( \text{cap}[k] \) to undefined.

Let \( e \) be \( x \)'s endIndex.

Let \( \text{xr} \) be the State \((e, \text{cap})\).

If \( \text{min} \) is not zero, then call \( m(\text{xr}, d) \) and return its result.

If \( \text{greedy} \) is false, then

a. Call \( c(x) \) and let \( z \) be its result.

b. If \( z \) is not failure, return \( z \).

c. Call \( m(\text{xr}, d) \) and return its result.

9. Call \( m(\text{xr}, d) \) 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 character sequence 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 \((n)\)st repetition of Atom are tried before moving on to the next choice in the \((n-1)\)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("abcdefghijklmnopqrstuvwxyz")
```

which returns "abcdefg" with

```
/a[a-z][2,4]/.exec("abcdefghijklmnopqrstuvwxyz")
```

which returns "abc".

Consider also

```
/aa|aab|ba|bc*/.exec("aabac")
```

which, by the choice point ordering above, returns the array

`["aab", "ba"]`

and not any of:

`["aabac", "aabac"]`

`["aabac", "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:

```
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa".replace(/\1\2/, "/1+$/).substring(0,1)
```

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

```
/(a+)(b+)?(c+)?/a[aacbbbcac]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 closure states that, once the minimum number of repetitions has been satisfied, any more expansions of Atom that match the empty character sequence 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([^\s]+)*/.exec("baaaaac")
```

which returns the array `['b', '']`

21.2.2.6 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`.

NOTE Even when the y flag is used with a pattern, `^` always matches only at the beginning of `Input`, or (if Multiline is `true`) at the beginning of a line.

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 :: \A` 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 List.
   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.

21.2.2.6.1 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.
   
   \[
   \begin{align*}
   \text{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, _}
   \end{align*}
   \]
4. Return false.

21.2.2.7 Quantifier

The production \( Quantifier ::= QuantifierPrefix \) evaluates as follows:
1. Evaluate \( QuantifierPrefix \) to obtain the two results: an integer \( min \) and an integer (or \( \infty \)) \( max \).
2. Return the three results \( min, max, \) and true.

The production \( Quantifier ::= QuantifierPrefix ? \) evaluates as follows:
1. Evaluate \( QuantifierPrefix \) to obtain the two results: an integer \( min \) and an integer (or \( \infty \)) \( max \).
2. Return the three results \( min, max, \) and false.

The production \( QuantifierPrefix ::= \{ \ DecimalDigits \} \) evaluates as follows:
1. Let \( i \) be the MV of \( DecimalDigits \) (see 11.8.3).
2. Return the two results \( i \) and \( i \).

The production \( QuantifierPrefix ::= \{ \ DecimalDigits \, \} \) evaluates as follows:
1. Let $i$ be the MV of `DecimalDigits`.
2. Return the two results $i$ and $\infty$.

The production `QuantifierPrefix` 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 $j$.

### 21.2.2.8 Atom

The production `Atom` 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` 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` 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` production is expanded prior to this production's `Atom` plus the total number of `Atom` 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 List.
      2. Let $xe$ be $x$'s endIndex.
      3. Let $ye$ be $y$'s endIndex.
      4. Let $s$ be a fresh List 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` evaluates by evaluating `Disjunction` to obtain a Matcher and returning that Matcher.
21.2.2.8.1 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 ce be the result of Canonicalize(ch).
   5. If invert is false, then
   6. If there does not exist a member a of set A such that Canonicalize(a) is ce, return failure.
   7. Else invert is true,
   8. If there exists a member a of set A such that Canonicalize(a) is ce, return failure.
   9. Let cap be x's captures List.
   10. Let y be the State (e+1, cap).
   11. Call c(y) and return its result.

21.2.2.8.2 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. If the file CaseFolding.txt of the Unicode Character Database does not provide a simple or common case folding mapping for ch, return ch.
3. Else, Let u be the result of apply that mapping to ch.
4. If Unicode is true, return u.
5. If u's code point > 65535, return ch
6. If ch's code unit value ≥ 128 and u's code unit value < 128, then return ch.
7. Return u.

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,

```
/(.*?)(?1)\1(.*?)/.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 \1) and a c. The second \1 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-sensitive matches all characters are implicitly case-folded using the simple mapping provided by the Unicode standard immediately before they are compared. The simple mapping always maps to a single code point, so it does not map, for example, “ß” (U+00DF) into “SS”. It may however map a code point outside the Basic Latin range to a character within, for example, “Γ” (U+017F) to “s”. Such characters are not mapped if `unicode` is false. This prevents Unicode code points such as U+0131 and U+017F from matching regular expressions such as `/[a-z]/i`, but they will match `/[a-z]/u`

### 21.2.2.9 `AtomEscape`

The production `AtomEscape :: DecimalEscape` evaluates as follows:

1. Evaluate `DecimalEscape` to obtain an `EscapeValue` E.
2. If E is a character, then
3. Let ch be E's character.
4. Let A be a one-element CharSet containing the character ch.
5. Call `CharacterSetMatcher(A, false)` and return its Matcher result.
6. E must be an integer. Let n be that integer,
7. If n=0 or n>NcapturingParens then throw a `SyntaxError` exception.
8. Return an internal Matcher closure that takes two arguments, a State x and a Continuation c, and performs the following:
   1. Let cap be x's captures List.
   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 e be e+len.
   7. If `InputLength`>e then 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 value as Canonicalize(Input[e+i]), then return `failure`.
   9. Let y be the State (e, 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.
4. 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.

### `AtomEscape :: CharacterClassEscape`

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 0). 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.
21.2.2.10 CharacterEscape

The production `CharacterEscape` evaluates by returning the character according to Table 34.

Table 34 — ControlEscape Character Values

<table>
<thead>
<tr>
<th>ControlEscape</th>
<th>Character Value</th>
<th>Code Point</th>
<th>Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>9</td>
<td>U+0009</td>
<td>horizontal tab</td>
<td>&lt;HT&gt;</td>
</tr>
<tr>
<td>n</td>
<td>10</td>
<td>U+000A</td>
<td>line feed (new line)</td>
<td>&lt;LF&gt;</td>
</tr>
<tr>
<td>v</td>
<td>11</td>
<td>U+000B</td>
<td>vertical tab</td>
<td>&lt;VT&gt;</td>
</tr>
<tr>
<td>f</td>
<td>12</td>
<td>U+000C</td>
<td>form feed</td>
<td>&lt;FF&gt;</td>
</tr>
<tr>
<td>r</td>
<td>13</td>
<td>U+000D</td>
<td>carriage return</td>
<td>&lt;CR&gt;</td>
</tr>
</tbody>
</table>

The production `CharacterEscape` evaluates as follows:

1. Let `ch` be the character represented by `ControlLetter`.
2. Let `i` be `ch`'s character value.
3. Let `j` be the remainder of dividing `i` by 32.
4. Return the character whose character value is `j`.

The production `CharacterEscape` evaluates by evaluating the `CV` of the `HexEscapeSequence` (see 11.8.4) and returning its character result.

The production `CharacterEscape` evaluates by returning the result of evaluating `RegExpUnicodeEscapeSequence`.

The production `RegExpUnicodeEscapeSequence` evaluates as follows:

1. Let `lead` be the result of evaluating `LeadSurrogate \u TrailSurrogate`.
2. Let `trail` be the result of evaluating `TrailSurrogate`.
3. Let `cp` be the UTF16Decode(`lead`, `trail`).
4. Return the character whose character value is `cp`.

The production `RegExpUnicodeEscapeSequence` evaluates by evaluating the `CV` of `HexDigits` (see 11.8.4.2) and returning its character result.

The production `RegExpUnicodeEscapeSequence` evaluates by evaluating the `MV` of the `HexDigits` (see 11.8.3.1) and returning its character result.

The production `LeadSurrogate \u HexDigits` evaluates by evaluating the `CV` of `HexDigits` (see 11.8.4.2) and returning its character result.

The production `TailSurrogate \u HexDigits` evaluates by evaluating the `CV` of `HexDigits` (see 11.8.4.2) and returning its character result.

21.2.2.11 DecimalEscape

The production `DecimalEscape` evaluates as follows:

1. Let `i` be the `MV` of `DecimalIntegerLiteral`.
2. If `i` is zero, return the `EscapeValue` consisting of the character U+0000 (NULL).
3. Return the `EscapeValue` consisting of the integer `i`.

The definition of “the `MV` of `DecimalIntegerLiteral`” is in 11.8.3.
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. \( \backslash 0 \) represents the \(<NUL>\) character and cannot be followed by a decimal digit.

21.2.2.12 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 (11.2)` or `LineTerminator (11.3)` productions.

The production `CharacterClassEscape :: 8` 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:

```
  abcdefghijklmnopqrstuvwxyz
  ABCDEFGHIJKLMNOPQRSTUVWXYZ
  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`.

21.2.2.13 CharacterClass

The production `CharacterClass :: [ 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.

21.2.2.14 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.

21.2.2.15 NonemptyClassRanges

The production `NonemptyClassRanges :: ClassAtom` evaluates by evaluating `ClassAtom` to obtain a CharSet and returning that CharSet.

The production `NonemptyClassRanges :: ClassAtom NonemptyClassRanges NoDash` evaluates as follows:

1. Evaluate `ClassAtom` to obtain a CharSet \( A \).
2. Evaluate `NonemptyClassRanges NoDash` 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`.

### 21.2.2.15.1 Runtime Semantics: CharacterRange Abstract Operation

The abstract operation `CharacterRange` takes two CharSet parameters `A` and `B` and performs the following:

- If `A` does not contain exactly one character or `B` does not contain exactly one character then throw a `SyntaxError` exception.
- Let `a` be the one character in CharSet `A`.
- Let `b` be the one character in CharSet `B`.
- Let `i` be the character value of character `a`.
- Let `j` be the character value of character `b`.
- If `i > j` then throw a `SyntaxError` exception.
- Return the set containing all characters numbered `i` through `j`, inclusive.

### 21.2.2.16 NonemptyClassRangesNoDash

The production `NonemptyClassRangesNoDash :: ClassAtom` evaluates by evaluating `ClassAtom` to obtain a CharSet and returning that CharSet.

The production `NonemptyClassRangesNoDash :: ClassAtomNonDash NonemptyClassRangesNoDash` evaluates as follows:

1. Evaluate `ClassAtomNonDash` 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 `ClassAtom`s and/or ranges of two `ClassAtom`s 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 character value is greater than the second `ClassAtom`'s character 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]A/` matches all upper and lower-case letters in the Unicode Basic Latin block as well as the symbols [, \, 1, "_, "_, 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.

### 21.2.2.17 ClassAtom

The production `ClassAtom :: -` evaluates by returning the CharSet containing the one character `-`. 

© Ecma International 2009 – All rights reserved
The production \texttt{ClassAtom} :: \texttt{ClassAtomNoDash} evaluates by evaluating \texttt{ClassAtomNoDash} to obtain a CharSet and returning that CharSet.

21.2.2.18 \texttt{ClassAtomNoDash}

The production \texttt{ClassAtomNoDash} :: \texttt{SourceCharacter} but not one of \texttt{\ or \} or \_} evaluates by returning a one-element CharSet containing the character represented by \texttt{SourceCharacter}.

The production \texttt{ClassAtomNoDash} :: \texttt{\ ClassEscape} evaluates by evaluating \texttt{ClassEscape} to obtain a CharSet and returning that CharSet.

21.2.2.19 \texttt{ClassEscape}

The production \texttt{ClassEscape} :: \texttt{DecimalEscape} evaluates as follows:

1. Evaluate \texttt{DecimalEscape} to obtain an \texttt{EscapeValue} \texttt{E}.
2. If \texttt{E} is not a character then throw a \texttt{SyntaxError} exception.
3. Let \texttt{ch} be \texttt{E}'s character.
4. Return the one-element CharSet containing the character \texttt{ch}.

The production \texttt{ClassEscape} :: \texttt{\ b} evaluates by returning the CharSet containing the one character <BS> U+0008 (BACKSPACE).

The production \texttt{ClassEscape} :: \texttt{CharacterEscape} evaluates by evaluating \texttt{CharacterEscape} to obtain a character and returning a one-element CharSet containing that character.

The production \texttt{ClassEscape} :: \texttt{CharacterClassEscape} evaluates by evaluating \texttt{CharacterClassEscape} to obtain a CharSet and returning that CharSet.

NOTE A \texttt{ClassAtom} can use any of the escape sequences that are allowed in the rest of the regular expression except for \texttt{\ b}, \texttt{\ B}, and backreferences. Inside a \texttt{CharacterClass}, \texttt{\ b} means the backspace character, while \texttt{\ B} and backreferences raise errors. Using a backreference inside a \texttt{ClassAtom} causes an error.

21.2.3 The RegExp Constructor

The RegExp constructor is the %RegExp% intrinsic object and the initial value of the RegExp property of the global object. When RegExp is called as a function rather than as a constructor, it creates and initialises a new RegExp object. Thus the function call \texttt{RegExp(\_\_\_\_\_\_) is equivalent to the object creation expression new RegExp(\_\_\_\_\_\_) with the same arguments. However, if the \texttt{this} value passed in the call is an Object with a [[RegExpMatcher]] internal slot whose value is \texttt{undefined}, it initialises the \texttt{this} value using the argument values. This permits RegExp to be used both as factory method and to perform constructor instance initialisation.

The RegExp constructor is designed to be subclassable. It may be used as the value of an extends clause of a class declaration. Subclass constructors that intended to inherit the specified RegExp behaviour must include a \texttt{super} call to the RegExp constructor to initialise subclass instances.

21.2.3.1RegExp(pattern, flags)

The following steps are taken:

1. Let \texttt{func} be this RegExp function object.
2. Let \texttt{O} be the \texttt{this} value.
3. If Type(\(O\)) is not Object or Type(\(O\)) is Object and \(O\) has a \([\text{RegExpMatcher}]\) internal slot and the value of \([\text{RegExpMatcher}]\) is not undefined, then
   a. If Type(\(pattern\)) is Object and \(O\) has a \([\text{RegExpMatcher}]\) internal slot and flags is undefined, then
   b. Let \(O\) be the result of calling the abstract operation RegExpAlloc with argument func.
   c. ReturnIfAbrupt(\(O\)).
4. If Type(\(pattern\)) is Object and \(pattern\) has a \([\text{RegExpMatcher}]\) internal slot, then
   a. If the value of \(pattern\)'s \([\text{RegExpMatcher}]\) internal slot is undefined, then throw a TypeError exception.
   b. If flags is not undefined, then throw a TypeError exception.
   c. Let \(P\) be the value of \(pattern\)'s \([\text{OriginalSource}]\) internal slot.
   d. Let \(F\) be the value of \(pattern\)'s \([\text{OriginalFlags}]\) internal slot.
5. Else,
   a. Let \(P\) be \(pattern\).
   b. Let \(F\) be flags.
6. Return the result of the abstract operation RegExpInitialise with arguments \(O\), \(P\), and \(F\).

NOTE If pattern is supplied using 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 \(\backslash\) characters must be escaped within the StringLiteral to prevent them being removed when the contents of the StringLiteral are formed.

21.2.3.2 new RegExp(...argumentsList)

When RegExp is called as part of a new expression with argument list argumentsList it performs the following steps:

1. Let \(F\) be the RegExp function object on which the new operator was applied.
2. Let argumentsList be the argumentsList argument of the [[Construct]] internal method that was invoked by the new operator.
3. Return the result of OrdinaryConstruct (\(F\), argumentsList).

If RegExp is implemented as an ECMAScript function object, its [[Construct]] internal method will perform the above steps.

21.2.3.3 Abstract Operations for the RegExp Constructor

21.2.3.3.1 Runtime Semantics: RegExpAlloc Abstract Operation

When the abstract operation RegExpAlloc with argument constructor is called, the following steps are taken:

1. Let \(obj\) be the result of calling OrdinaryCreateFromConstructor(constructor, "RegExpPrototype", ([RegExpMatcher], [OriginalSource], [OriginalFlags])).
2. Let status be the result of DefinePropertyOrThrow(obj, "lastIndex", PropertyDescriptor {[[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false}).
3. ReturnIfAbrupt(status).
4. Return \(obj\).

NOTE \([\text{RegExpMatcher}]\) is initially assigned the value undefined as a flag to indicate that the instance has not yet been initialised by the RegExp constructor. This flag value is never directly exposed to ECMAScript code; hence implementations may choose to encode the flag in some other manner.

21.2.3.3.2 Runtime Semantics: RegExpInitialise Abstract Operation

When the abstract operation RegExpInitialise with arguments obj, pattern, and flags is called, the following steps are taken:

1. If pattern is undefined, then let \(P\) be the empty String.
2. Else, let \( P \) be ToString(pattern).
3. ReturnIfAbrupt(P).
4. If flags is undefined, then let \( F \) be the empty String.
5. Else, let \( F \) be ToString(flags).
6. ReturnIfAbrupt(F).
7. If \( F \) contains any character other than "g", "i", "m", "u", or "y" or if it contains the same character more than once, then throw a SyntaxError exception.
8. Parse \( P \) interpreted as UTF-16 encoded Unicode code points using the grammars in 21.2.1. If \( F \) contains *u* the goal symbol for the parse is Pattern. Otherwise the goal symbol for the parse is Pattern. Throw a SyntaxError exception if \( P \) did not conform to the grammar or if all characters of \( P \) where not matched by the parse.
9. Set obj's [[RegExpMatcher]] internal slot to the internal procedure obtained by evaluating (‘compiling’) the step 3’s parse of \( P \) and applying the semantics provided in 21.2.2.
10. Set the value of obj's [[OriginalSource]] internal slot to \( P \).
11. Set the value of obj's [[OriginalFlags]] internal slot to \( F \).
12. Let putStatus be the result of Put(obj, "lastIndex", 0, true).
13. ReturnIfAbrupt(putStatus).

21.2.3.3 Runtime Semantics: RegExpCreate Abstract Operation

When the abstract operation RegExpCreate with arguments \( P \) and \( F \) is called, the following steps are taken:

1. Let obj be the result of calling the abstract operation RegExpAlloc with argument %RegExp%.
2. ReturnIfAbrupt(obj).
3. Return the result of the abstract operation RegExpInitialise with arguments obj, \( P \), and \( F \).

21.2.3.4 Runtime Semantics: EscapeRegExpPattern Abstract Operation

When the abstract operation EscapeRegExpPattern with arguments \( P \) and \( F \) is called, the following occurs:

Let \( S \) be a String in the form of a Pattern (PatternPS) if \( F \) contains *u* equivalent to \( P \) interpreted as UTF-16 encoded Unicode code points, in which certain code points are escaped as described below. \( S \) may or may not be identical to \( P \); however, the internal procedure that would result from evaluating \( S \) as a Pattern (PatternPS) if \( F \) contains *u* must behave identically to the internal procedure given by the constructed object's [[RegExpMatcher]] internal slot. Separate calls to this abstract operation using the same values for \( P \) and \( F \) must produce identical results.

The characters / or any LineTerminator occurring in the pattern shall be escaped in \( S \) as necessary to ensure that the String value formed by (concatenating the Strings "/", \( S \), "/", and \( F \) and can be parsed in an (appropriate lexical context) as a RegularExpressionLiteral that behaves identically to the constructed regular expression. For example, if \( P \) is "/f", then \( S \) could be "\"f\" or "\ux002F", among other possibilities but not "f", because /f/ 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 " (?::+) ".

Return \( S \).

21.2.4 Properties of the RegExp Constructor

The value of the [[Prototype]] internal slot of the RegExp constructor is the standard built-in Function prototype object (19.2.3).

Besides the length property (whose value is 2), the RegExp constructor has the following properties:

Commented [AWB14110]: Why is this underspecified? Why not specify an required escaping? Do different implementation differ in their results?
21.2.4.1 RegExp.prototype

The initial value of `RegExp.prototype` is the RegExp prototype object (21.2.5).
This property shall have the attributes {[[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}.

21.2.4.2 `RegExp @@create()`

The `@@create` method of an object `F` performs the following:
1. Return the result of calling the abstract operation `RegExpAlloc` with argument `F`.

The value of the `name` property of this function is “`[Symbol.create]`”.
This property has the attributes {[[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true}.

21.2.5 Properties of the RegExp Prototype Object

The RegExp prototype object is an ordinary object. It is not a RegExp instance and does not have a `[[RegExpMatcher]]` internal slot or any of the other internal slots of RegExp instance objects.

The value of the `[[Prototype]]` internal slot of the RegExp prototype object is the standard built-in Object prototype object (19.1.3).

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

21.2.5.1 RegExp.prototype.constructor

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

21.2.5.2 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 the this value.
2. If `typeof R` is not `Object`, then throw a `TypeError` exception.
3. If `R` does not have a `[[RegExpMatcher]]` internal slot, then throw a `TypeError` exception.
4. If the value of `R`'s `[[RegExpMatcher]]` internal slot is `undefined`, then throw a `TypeError` exception.
5. Let `S` be the value of `ToString(string)`.
6. ReturnIfAbrupt(`S`).
7. Return the result of the `RegExpExec` abstract operation with arguments `R` and `S`.

21.2.5.2.1 Runtime Semantics: RegExpExec Abstract Operation

The abstract operation `RegExpExec` with arguments `R` (an object) and `S` (a string) performs the following steps:

1. Assert: `R` is an initialised RegExp instance.
2. Let `length` be the length of `S`.
3. Let `lastIndex` be the result of `Get(R, "lastIndex")`.
4. Let `i` be the value of `ToInteger(lastIndex)`.
5. ReturnIfAbrupt(`i`).
6. Let `global` be the result of `ToBoolean(Get(R, "global"))`.
7. ReturnIfAbrupt(`global`).
8. Let sticky be the result of ToBoolean(Get(R, "sticky"));
9. ReturnIfAbrupt(sticky).
10. If global is false and sticky is false, then let i = 0.
11. Let matcher be the value of R’s [[RegExpMatcher]] internal slot.
12. Let matchSucceeded be false.
13. 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).
   b. Let r be the result of calling matcher with arguments S and i.
   c. If r is failure, then
      i. If sticky is true, then
         1. Let putStatus be the result of Put(R, "lastIndex", 0, true).
         2. ReturnIfAbrupt(putStatus).
         3. Return null.
      ii. Let i = i + 1.
   d. else
      i. Assert: r is a State.
      ii. Set matchSucceeded to true.
14. Let e be r’s endIndex value.
15. If global is true or sticky is true, then
   a. Let putStatus be the result of Put(R, "lastIndex", e, true).
   b. ReturnIfAbrupt(putStatus).
16. Let n be the length of r’s captures List. (This is the same value as 21.2.2.1’s capturingParens.)
17. Let A be the result of the abstract operation ArrayCreate(n + 1).
18. Let matchIndex be i.
19. Assert: The following CreateDataProperty calls will not result in an abrupt completion.
20. Call CreateDataProperty(A, "index", matchIndex).
22. Let matchedSubstr be the matched substring (i.e. the portion of S between offset i inclusive and offset e exclusive).
23. Call CreateDataProperty(A, "0", matchedSubstr).
24. For each integer i such that 0 ≤ i ≤ n
   a. Let captured be i-th element of r’s captures List.
   b. Call CreateDataProperty(A, ToString(i), captured).
25. Return A.

21.2.5.3 get RegExp.prototype.global

RegExp.prototype.global is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:
1. Let R be the this value.
2. If Type(R) is not Object, then throw a TypeError exception.
3. If R does not have an [[OriginalFlags]] internal slot throw a TypeError exception.
4. Let flags be the value of R’s [[OriginalFlags]] internal slot.
5. If flags is undefined, then throw a TypeError exception.
6. If flags contains the character "g", then return true.
7. Return false.

21.2.5.4 get RegExp.prototype.ignoreCase

RegExp.prototype.ignoreCase is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:
1. Let \( R \) be the this value.
2. If Type(\( R \)) is not Object, then throw a TypeError exception.
3. If \( R \) does not have an [[OriginalFlags]] internal slot throw a TypeError exception.
4. Let flags be the value of \( R \)’s [[OriginalFlags]] internal slot.
5. If flags is undefined, then throw a TypeError exception.
6. If flags contains the character “\*”, then return true.
7. Return false.

21.2.5.5 RegExp.prototype.match (string)

When the match method is called with argument \( string \), the following steps are taken:

1. Let \( rx \) be the this value.
2. If Type(\( rx \)) is not Object, then throw a TypeError exception.
3. If \( rx \) does not have a [[RegExpMatcher]] internal slot, then throw a TypeError exception.
4. If the value of \( rx \)’s [[RegExpMatcher]] internal slot is undefined, then throw a TypeError exception.
5. Let \( S \) be the value of ToString(\( string \)).
6. ReturnIfAbrupt(\( S \)).
7. Let global be the result of ToBoolean(Get(\( rx \), “global”)).
8. ReturnIfAbrupt(global).
9. If global is not true, then
   a. Return the result of calling the abstract operation RegExpExec (see 21.2.5.2) 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 the abstract operation RegExpExec 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 ToInteger(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. Let defineStatus be the result of CreateDataPropertyOrThrow(A, ToString(\( n \)), matchStr).
          7. ReturnIfAbrupt(defineStatus).
          8. Increment \( n \).
             a. If \( n = 0 \), then return null.
             b. Return A.
11. Else:
    h. If n = 0, then return null.
    i. Return A.

21.2.5.6 get RegExp.prototype.multiline

RegExp.prototype.multiline is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let \( R \) be the this value.
2. If Type(\( R \)) is not Object, then throw a TypeError exception.
3. If \( R \) does not have an [[OriginalFlags]] internal slot throw a TypeError exception.
4. Let flags be the value of `R`'s `[[OriginalFlags]]` internal slot.
5. If `flags` is `undefined`, then throw a `TypeError` exception.
6. If `flags` contains the character "m", then return `true`.
7. Return `false`.

### 21.2.5.7 RegExp.prototype.replace (S, replaceValue)

When the `replace` method is called with arguments `S` and `replaceValue` the following steps are taken:

#### TODO: need to finish this and have it make use of GetReplaceSubstitution operation in 21.1.3.14

1. Let `rx` be the `this` value.
2. If `Type(rx)` is not Object, then throw a `TypeError` exception.
3. If `rx` does not have a `[[RegExpMatcher]]` internal slot, then throw a `TypeError` exception.
4. If the value of `rx`'s `[[RegExpMatcher]]` internal slot is `undefined`, then throw a `TypeError` exception.
5. Let `string` be `ToString(S)`.
6. ReturnIfAbrupt(`string`).
7. 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 `RegExp.prototype.match`, including the update of `searchValue.lastIndex`. Let `m` be the number of left capturing parentheses in `searchValue` (using `NcapturingParens` as specified in 21.2.2.1).
8. If `replaceValue` is a function, then
   a. 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 21.2.2.1). 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.
9. Else,
   a. 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, "$01-$012") returns "$1-$11, $1-$122". A $ in `newstring` that does not match any of the forms below is left as is.

### 21.2.5.8 RegExp.prototype.search (S)

When the `search` method is called with argument `S`, the following steps are taken:

1. Let `rx` be the `this` value.
2. If `Type(rx)` is not Object, then throw a `TypeError` exception.
3. If `rx` does not have a `[[RegExpMatcher]]` internal slot, then throw a `TypeError` exception.
4. If the value of `rx`'s `[[RegExpMatcher]]` internal slot is `undefined`, then throw a `TypeError` exception.
5. Let `string` be `ToString(S)`.
6. ReturnIfAbrupt(`string`).
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.

21.2.5.9 get RegExp.prototype.source

RegExp.prototype.source is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let R be the this value.
2. If Type(R) is not Object, then throw a TypeError exception.
3. If R does not have an [[OriginalSource]] internal slot throw a TypeError exception.
4. If R does not have an [[OriginalFlags]] internal slot throw a TypeError exception.
5. Let src be the value of R’s [[OriginalSource]] internal slot.
6. Let flags be the value of R’s [[OriginalFlags]] internal slot.
7. If either src or flags is undefined, then throw a TypeError exception.
8. Return the result of the abstract operation EscapeRegExpPattern with arguments src and flags.

21.2.5.10 RegExp.prototype.split (string, limit)

Returns an Array object into which substrings of the result of converting string to a String have been stored. The substrings are determined by searching from left to right for matches of the this value regular expression; these occurrences are not part of any substring in the returned array, but serve to divide up the String value.

The this value may be an empty regular expression or a regular expression that can match an empty String. In this case, regular expression does not match the empty substring at the beginning or end of the input String, nor does it match the empty String itself.

If the string is (or converts to) the empty String, the result depends on whether the regular expression 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 the 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, /<.*?[^<]*>/.split("<A>B</B>and<CODE>coded</CODE>\")

evaluates to the array

["A", undefined, "B", "bold", ",/", "B", "and", undefined, "CODE", """"]

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. Let rx be the this value.
2. If Type(rx) is not Object, then throw a TypeError exception.
3. If rx does not have a [[RegExpMatcher]] internal slot, then throw a TypeError exception.
4. If the value of rx’s [[RegExpMatcher]] internal slot is undefined, then throw a TypeError exception.
5. Let matcher be the value of rx’s [[RegExpMatcher]] internal slot.
6. Let S be ToString(string).
7. ReturnIfAbrupt(S).
8. Let A be the result of the abstract operation ArrayCreate with argument 0.
9. ReturnIfAbrupt(A).
10. Let lengthA be 0.
11. If `limit` is `undefined`, let `lim = 2^{53} - 1`; else let `lim = ToLength(limit)`.
12. Let `s` be the number of elements in `S`.
13. Let `p = 0`.
14. If `lim = 0`, return `A`.
15. If `s = 0`, then
   a. Let `z` be the result of calling the `matcher` with arguments `S` and `0`.
      b. ReturnIfAbrupt(`z`).
     c. If `z` is not `failure`, return `A`.
   d. Assert: The following call will never result in an abrupt completion.
     e. Call `CreateDataProperty(A, "0", S)`.
      f. Return `A`.
16. Let `q = p`.
17. Repeat, while `q ≠ s`
   a. Let `z` be the result of calling the `matcher` with arguments `S` and `q`.
      b. ReturnIfAbrupt(`z`).
     c. If `z` is `failure`, then let `q = q+1`.
   d. Else `z` is not `failure`.
      i. Let `v` be a State. Let `e` be `v`’s `endIndex` and let `cap` be `v`’s `captures` List.
      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. Assert: The following call will never result in an abrupt completion.
         3. Call `CreateDataProperty(A, ToString(lengthA), T)`.
         4. If `lengthA = lim`, return `A`.
         5. Let `p = e`.
         6. Let `i = 0`.
         7. Repeat, while `i` is not equal to the number of elements in `cap`.
            a. Let `i = i+1`.
            b. Assert: The following call will never result in an abrupt completion.
            c. Call `CreateDataProperty(A, ToString(lengthA), cap[i])`.
            d. Increment `lengthA` by 1.
            e. If `lengthA = lim`, return `A`.
   8. Let `q = p`.
18. Let `T` be a String value equal to the substring of `S` consisting of the elements at positions `p` (inclusive) through `s` (exclusive).
19. Assert: The following call will never result in an abrupt completion.
20. Call `CreateDataProperty(A, ToString(lengthA), T)`.

The `length` property of the `split` method is `2`.

NOTE 1 The `split` method ignores the value of the `global` property of this RegExp object.

21.2.5.11 get RegExp.prototype.sticky

RegExp.prototype.sticky is an accessor property whose set accessor function is `undefined`. Its get accessor function performs the following steps:

1. Let `R` be the `this` value.
2. If `R` is not `Object`, then throw a `TypeError` exception.
3. If `R` does not have an `[[OriginalFlags]]` internal slot throw a `TypeError` exception.
4. Let `flags` be the value of `R`’s `[[OriginalFlags]]` internal slot.
5. If `flags` is `undefined`, then throw a `TypeError` exception.
6. If `flags` contains the character “y”, then return `true`.
7. Return `false`. 
21.2.5.12 RegExp.prototype.test(string)
The following steps are taken:
1. Let \( R \) be the this value.
2. If \( \text{Type}(R) \) is not Object, then throw a TypeError exception.
3. If \( R \) does not have a [[RegExpMatcher]] internal slot, then throw a TypeError exception.
4. If the value of \( R \)'s [[RegExpMatcher]] internal slot is undefined, then throw a TypeError exception.
5. Let \( S \) be the value of \( \text{ToString}(string) \).
6. ReturnIfAbrupt(\( S \)).
7. Let match be the result of the RegExpExec abstract operation with arguments \( R \) and \( S \).
8. ReturnIfAbrupt(match).
9. If match is not null, then return true; else return false.

21.2.5.13 RegExp.prototype.toString()
1. Let \( R \) be the this value.
2. If \( \text{Type}(R) \) is not Object, then throw a TypeError exception.
3. If \( R \) does not have a [[RegExpMatcher]] internal slot, then throw a TypeError exception.
4. If the value of \( R \)'s [[RegExpMatcher]] internal slot is undefined, then throw a TypeError exception.
5. Let pattern be the result of \( \text{ToString}(\text{Get}(R, \text{"source"})) \).
6. ReturnIfAbrupt(pattern).
7. Let result be the String value formed by concatenating "/", pattern, and "/".
8. Let global be the result of \( \text{ToBoolean}(\text{Get}(R, \text{"global"})) \).
9. ReturnIfAbrupt(global).
10. If global is true, then append "g" as the last character of result.
11. Let ignoreCase be the result of \( \text{ToBoolean}(\text{Get}(R, \text{"ignoreCase"})) \).
12. ReturnIfAbrupt(ignoreCase).
13. If ignoreCase is true, then append "i" as the last character of result.
14. Let multiline be the result of \( \text{ToBoolean}(\text{Get}(R, \text{"multiline"})) \).
15. ReturnIfAbrupt(multiline).
16. If multiline is true, then append "m" as the last character of result.
17. Let unicode be the result of \( \text{ToBoolean}(\text{Get}(R, \text{"unicode"})) \).
18. ReturnIfAbrupt(unicode).
19. If unicode is true, then append "u" as the last character of result.
20. Let sticky be the result of \( \text{ToBoolean}(\text{Get}(R, \text{"sticky"})) \).
21. ReturnIfAbrupt(sticky).
22. If sticky is true, then append "y" as the last character of result.
23. Return result.

NOTE The returned String has the form of a RegularExpressionLiteral that evaluates to another RegExp object with the same behaviour as this object.

21.2.5.14 get RegExp.prototype.unicode
RegExp.prototype.unicode is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:
1. Let \( R \) be the this value.
2. If \( \text{Type}(R) \) is not Object, then throw a TypeError exception.
3. If \( R \) does not have an [[OriginalFlags]] internal slot throw a TypeError exception.
4. Let flags be the value of \( R \)'s [[OriginalFlags]] internal slot.
5. If flags is undefined, then throw a TypeError exception.
6. If flags contains the character "u", then return true.
7. Return false.

21.2.5.15 RegExp.prototype @@isRegExp
The initial value of the @@isRegExp property is true.
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

21.2.6 Properties of RegExp Instances

RegExp instances are ordinary objects that inherit properties from the RegExp prototype object. RegExp instances have internal slots [[RegExpMatcher]], [[OriginalSource]], and [[OriginalFlags]]. The value of the [[RegExpMatcher]] internal slot is an implementation dependent representation of the Pattern of the RegExp object.

NOTE Prior to the 6th Edition, RegExp instances were specified as having the own data properties source, global, ignoreCase, and multiline. Those properties are now specified as accessor properties of RegExp.prototype.

RegExp instances also have the following property:

21.2.6.1 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 21.2.5.2). This property shall have the attributes { [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false }.

22 Indexed Collections

22.1 Array Objects

Array objects are exotic objects that give special treatment to a certain class of property names. See 9.4.2 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\leq 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.

22.1.1 The Array Constructor

The Array constructor is the %Array% intrinsic object and the initial value of the Array property of the global object. 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. However, if the this value passed in the call is an Object with an [[ArrayInitialisationState]] internal slot whose value is undefined, it initialises the this value using the argument values. This permits Array to be used both as a factory method and to perform constructor instance initialisation.

The Array constructor is designed to be subclassable. It may be used as the value of an extends clause of a class declaration. Subclass constructors that intended to inherit the specified Array behaviour must include a super call to the Array constructor to initialise subclass instances.

22.1.1.1 Array ([ item1 [, item2 [, ... ] ] ])

This description applies if and only if the Array constructor is called with no arguments or at least two arguments.
When the `Array` function is called the following steps are taken:

1. Let `numberOfArgs` be the number of arguments passed to this constructor call.
3. Let `O` be the `this` value.
4. If `Type(O)` is Object and `O` has an `[[ArrayInitialisationState]]` internal slot and the value of `[[ArrayInitialisationState]]` is `false`, then
   a. Set the value of `O'`s `[[ArrayInitialisationState]]` internal slot to `true`.
   b. Let `array` be `O`.
5. Else,
   a. Let `F` be this function.
   b. Let `proto` be the result of `GetPrototypeOfConstructor(F, "%ArrayPrototype%")`.
   c. ReturnIfAbrupt(proto).
   d. Let `array` be the result of the abstract operation `ArrayCreate` with arguments `numberOfArgs` and `proto`.
6. ReturnIfAbrupt(array).
7. Let `k` be 0.
8. Let `items` be a zero-originated List containing the argument items in order.
9. Repeat, while `k < numberOfArgs`
   a. Let `Pk` be `ToString(k)`.
   b. Let `itemK` be the `k`-th element of `items`.
   c. Let `defineStatus` be the result of `CreateDataPropertyOrThrow(array, Pk itemK)`.
   d. ReturnIfAbrupt(defineStatus).
   e. Increase `k` by 1.
10. Let `putStatus` be the result of `Put(array, "length", numberOfArgs, true)`.
11. ReturnIfAbrupt(putStatus).
12. Return `array`.

### 22.1.1.2 Array (len)

This description applies if and only if the Array constructor is called with exactly one argument.

1. Let `numberOfArgs` be the number of arguments passed to this constructor call.
3. Let `O` be the `this` value.
4. If `Type(O)` is Object and `O` has an `[[ArrayInitialisationState]]` internal slot and the value of `[[ArrayInitialisationState]]` is `false`, then
   a. Set the value of `O'`s `[[ArrayInitialisationState]]` internal slot to `true`.
   b. Let `array` be `O`.
5. Else,
   a. Let `F` be this function.
   b. Let `proto` be the result of `GetPrototypeOfConstructor(F, "%ArrayPrototype%")`.
   c. ReturnIfAbrupt(proto).
   d. Let `array` be the result of `ArrayCreate(0, proto)`.
6. ReturnIfAbrupt(array).
7. If `Type(len)` is not `Number`, then
   a. Let `defineStatus` be the result of `CreateDataPropertyOrThrow(array, "0", len)`.
   b. ReturnIfAbrupt(defineStatus).
   c. Let `intLen` be 1.
8. Else,
   a. Let `intLen` be `ToUint32(len)`.
   b. If `intLen` ≠ `len`, then throw a `RangeError` exception.
9. Let `putStatus` be the result of `Put(array, "length", intLen, true)`.
10. ReturnIfAbrupt(putStatus).
11. Return `array`.

### 22.1.1.3 new Array ( ... argumentsList)

When `Array` is called as part of a `new` expression, it initialises a newly created object.
1. Let \( F \) be the `Array` function object on which the `new` operator was applied.
2. Let `argumentsList` be the `argumentsList` argument of the `[[Construct]]` internal method that was invoked by the `new` operator.
3. Return the result of `OrdinaryConstruct(F, argumentsList)`.

If `Array` is implemented as an ECMAScript function object, its `[[Construct]]` internal method will perform the above steps.

### 22.1.2 Properties of the Array Constructor

The value of the `[[Prototype]]` internal slot of the `Array` constructor is the `Function` prototype object (19.2.3).

Besides the `length` property (whose value is 1), the `Array` constructor has the following properties:

#### 22.1.2.1 `Array.from(arrayLike, mapfn=undefined, thisArg=undefined)`

When the `from` method is called with argument `arrayLike` and optional arguments `mapfn` and `thisArg` the following steps are taken:

1. Let \( C \) be the `this` value.
2. Let `items` be `ToObject(arrayLike)`.
3. ReturnIfAbrupt(`items`).
4. If `mapfn` is `undefined`, then let `mapping` be `false`.
5. else
   a. If `IsCallable(mapfn)` is `false`, throw a `TypeError` exception.
   b. If `thisArg` was supplied, let \( T \) be \( thisArg \); else let \( T \) be `undefined`.
5. c. Let `mapping` be `true`.
6. Let `usingIterator` be the result of `HasProperty(items, @@iterator)`.
7. ReturnIfAbrupt(`usingIterator`).
8. If `usingIterator` is `true`, then
   a. If `IsConstructor(C)` is `true`, then
      i. Let `A` be the result of calling the `[[Construct]]` internal method of `C` with an empty argument list.
   b. Else,
      i. Let `A` be the result of the abstract operation `ArrayCreate` with argument 0.
      c. ReturnIfAbrupt(`A`).
      d. Let `iterator` be the result of performing `GetIterator(items)`.
      e. ReturnIfAbrupt(`iterator`).
     f. Let `k` be 0.
    g. Repeat
      i. Let `Pk` be `ToString(k)`.
      ii. Let `next` be the result of `IteratorStep(iterator)`.
      iii. ReturnIfAbrupt(`next`).
      iv. If `next` is `false`, then
         i. Let `putStatus` be the result of `Put(A, "length", k, true)`.
         2. ReturnIfAbrupt(`putStatus`).
         3. Return `A`.
       v. Let `nextValue` be `IteratorValue(next)`.
       vi. ReturnIfAbrupt(`nextValue`).
     vii. If `mapping` is `true`, then
        1. Let `mappedValue` be the result of calling the `[[Call]]` internal method of `mapfn` with \( T \) as `thisArgument` and a `List` containing `nextValue` as `argumentsList`.
        2. ReturnIfAbrupt(`mappedValue`).
    viii. Else, let `mappedValue` be `nextValue`.
    ix. Let `defineStatus` be the result of `CreateDataPropertyOrThrow(A, Pk, mappedValue)`.  

Commented [AWB7114]: It would be nice to have a more explicit way to create a collection with a pre-specified number of elements.
375

ReturnIfAbrupt(defineStatus).

Increase k by 1.

9. \textbf{Assert:} items is not an Iterator so assume it is an array-like object.

10. Let lenValue be the result of Get(items, "\texttt{length}").

11. Let len be ToLength(lenValue).

12. ReturnIfAbrupt(len).

13. If IsConstructor(C) is \texttt{true}, then
   a. Let A be the result of calling the [[Construct]] internal method of C with an argument list containing the single item \texttt{len}.

14. Else,
   a. Let A be the result of the abstract operation ArrayCreate with argument \texttt{len}.

15. ReturnIfAbrupt(A).

16. Let k be 0.

17. Repeat, while k < len
   a. Let \texttt{Pk} be ToString(k).
   b. Let kPresent be the result of HasProperty(items, \texttt{Pk}).
   c. ReturnIfAbrupt(kPresent).
   d. If kPresent is \texttt{true}, then
      i. Let kValue be the result of Get(items, \texttt{Pk}).
      ii. ReturnIfAbrupt(kValue).
      iii. If mapping is \texttt{true}, then
        1. Let mappedValue be the result of calling the [[Call]] internal method of mapfn with \texttt{T} as thisArgument and a List containing kValue, k, and items as argumentsList.
        2. ReturnIfAbrupt(mappedValue).
      iv. Else, let mappedValue be kValue.
      v. Let defineStatus be the result of CreateDataPropertyOrThrow(A, \texttt{Pk}, mappedValue).
      vi. ReturnIfAbrupt(defineStatus).
     e. Increase k by 1.

18. Let putStatus be the result of Put(A, \texttt{"length"}, len, true).

19. ReturnIfAbrupt(putStatus).

20. Return A.

The length property of the from method is 1.

\textbf{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.

22.1.2.2 Array.isArray ( arg )

The isArray function takes one integer arg, and performs the following:

1. If Type(arg) is not Object, return \texttt{false}.
2. If arg is an exotic Array object, then return \texttt{true}.
3. Return \texttt{false}.

22.1.2.3 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, "\texttt{length}").
2. Let len be ToLength(lenValue).
3. Let C be the this value.
4. If IsConstructor(C) is \texttt{true}, then
   a. Let A be the result of calling the [[Construct]] internal method of C with an argument list containing the single item \texttt{len}.
5. Else,
   a. Let A be the result of the abstract operation ArrayCreate with argument \texttt{len}.
6. ReturnIfAbrupt(A).
7. Let k be 0.
8. Repeat, while \( k < \text{len} \)
   a. Let \( Pk \) be \( \text{ToString}(k) \).
   b. Let \( kValue \) be the result of \( \text{Get(items, Pk)} \).
   c. Let \( \text{defineStatus} \) be the result of \( \text{CreateDataPropertyOrThrow}(A, Pk, kValue, [\text{value}]) \).
   d. ReturnIfAbrupt(\( \text{defineStatus} \)).
   e. Increase \( k \) by 1.

9. Let \( \text{putStatus} \) be the result of \( \text{Put}(A, \"\text{length}\", \text{len}, \text{true}) \).
10. ReturnIfAbrupt(\( \text{putStatus} \)).
11. Return \( A \).

The \text{length} property of the of method is 0.

NOTE 1 The \text{items} argument is assumed to be a well-formed rest argument value.

NOTE 2 The of function is an intentionally generic factory method; it does not require that its this value be the Array constructor. Therefore it can be transferred to or inherited by other constructors that may be called with a single numeric argument.

22.1.2.4 Array.prototype

The value of Array.prototype is \%ArrayPrototype\%, the intrinsic Array prototype object (22.1.3).

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

22.1.2.5 Array[ @@create ] ()

The @@create method of an object \( F \) performs the following steps:

1. Let \( F \) be the this value.
2. Let \( \text{proto} \) be the result of \( \text{GetPrototypeFromConstructor}(F, \%\text{ArrayPrototype}\%) \).
3. ReturnIfAbrupt(\( \text{proto} \)).
4. Let \( \text{obj} \) be the result of calling ArrayCreate with arguments undefined and \( \text{proto} \).
5. Return \( \text{obj} \).

The value of the name property of this function is \"Symbol.create\".

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

NOTE 1 Passing undefined as the first argument to ArrayCreate causes the [[ArrayInitialisationState]] internal slot of the array to be initially assigned the value false. This is a flag used to indicate that the instance has not yet been initialised by the Array constructor. This flag value is never directly exposed to ECMAScript code; hence implementations may choose to encode the flag in any unobservable manner.

NOTE 2 The Array @@create function is intentionally generic; it does not require that its this value be the Array constructor object. It can be transferred to other constructor functions for use as a @@create method. When used with other constructors, this function will create an exotic Array object whose [[Prototype]] value is obtained from the associated constructor.

22.1.3 Properties of the Array Prototype Object

The value of the [[Prototype]] internal slot of the Array prototype object is the intrinsic object \%ObjectPrototype\%.

The Array prototype object is itself an ordinary object. It is not an Array instance and does not have a length property.
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.

22.1.3.1 `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 `undefined`.
4. If `O` is an exotic Array object, then
   a. Let `C` be `Get(O, "constructor")`.
   b. ReturnIfAbrupt(`C`).
   c. If `IsConstructor(C)` is `true`, then
      i. Let `thisRealm` be the running execution context’s Realm.
      ii. If `thisRealm` and the value of `C`'s `[[Realm]]` internal slot are the same value, then
         1. Let `A` be the result of calling the `[[Construct]]` internal method of `C` with argument (0).
5. If `A` is `undefined`, then
   a. Let `A` be the result of the abstract operation `ArrayCreate` with argument 0.
6. ReturnIf Abrupt(`A`).
7. Let `n` be 0.
8. Let `items` be a 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.
9. Repeat, while `items` is not empty
   a. Remove the first element from `items` and let `E` be the value of the element.
   b. Let `spreadable` be the result of `IsConcatSpreadable(E)`.
   c. ReturnIfAbrupt(`spreadable`).
   d. If `spreadable` is `true`, then
      i. Let `k` be 0.
      ii. Let `lenVal` be the result of `Get(E, "length")`.
      iii. Let `len` be `ToLength(lenVal)`.
      iv. ReturnIfAbrupt(`len`).
      v. 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. ReturnIfAbrupt(`subElement`).
            c. Let `status` be the result of `CreateDataPropertyOrThrow (A, ToString(n), subElement)`.
            d. ReturnIfAbrupt(`status`).
            5. Increase `n` by 1.
         6. Increase `k` by 1.
     e. else `E` is added as a single item rather than spread,
        i. Let `status` be the result of `CreateDataPropertyOrThrow (A, ToString(n), E)`.
        ii. ReturnIfAbrupt(`status`).
        iii. Increase `n` by 1.
   10. Let `putStatus` be the result of `Put(A, "length", n, true)`.
11. ReturnIfAbrupt(`putStatus`).
12. Return `A`.

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

NOTE 1 The explicit setting of the `length` property in step 10 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.

22.1.3.1.1 IsConcatSpreadable ( O ) Abstract Operation

The abstract operation IsConcatSpreadable with argument O performs the following steps:

1. If Type(O) is not Object, then return false.
2. Let spreadable be Get(O, @@isConcatSpreadable).
3. ReturnIfAbrupt(spreadable).
4. If spreadable is not undefined, then return ToBoolean(spreadable).
5. If O is an exotic Array object, then return true.
6. Return false.

22.1.3.2 Array.prototype.constructor

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

22.1.3.3 Array.prototype.copyWithin (target, start, end = this.length)

The copyWithin method takes up to three arguments target, start and end. The end argument is optional with the length of the this object as its default value. If target is negative, it is treated as length+target where length is the length of the array. If start is negative, it is treated as length+start. If end is negative, it is treated as length+end. 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 ToLength(lenVal).
5. ReturnIfAbrupt(len).
6. Let relativeTarget be ToInteger(target).
7. ReturnIfAbrupt(relativeTarget).
8. If relativeTarget is negative, let to be max((len + relativeTarget),0); else let to be min(relativeTarget, len).
9. Let relativeStart be ToInteger(start).
10. ReturnIfAbrupt(relativeStart).
11. If relativeStart is negative, let from be max((len + relativeStart),0); else let from be min(relativeStart, len).
12. If end is undefined, let relativeEnd be len; else let relativeEnd be ToInteger(end).
13. ReturnIfAbrupt(relativeEnd).
14. If relativeEnd is negative, let final be max((len + relativeEnd),0); else let final be min(relativeEnd, len).
15. Let count be min(final-from, len-to).
16. If from<to and to<from+count
a. Let direction = -1.
b. Let from = from + count × -1.
c. Let to = to + count × -1.
17. Else
a. Let direction = 1.
18. Repeat, while count > 0
a. Let fromKey be ToString(from).
b. Let toKey be ToString(to).
c. Let fromPresent be the result of HasProperty(O, fromKey).
d. ReturnIfAbrupt(fromPresent).
e. If fromPresent is true, then
   i. Let fromVal be the result of Get(O, fromKey).
   ii. ReturnIfAbrupt(fromVal).
   iii. Let putStatus be the result of Put(O, toKey, fromVal, true),

iv. ReturnIfAbrupt(putStatus).

f. Else fromPresent is false:
   i. Let deleteStatus be the result of DeletePropertyOrThrow(O, toKey).
   ii. ReturnIfAbrupt(deleteStatus).

g. Let from be from + direction.
   b. Let to be to + direction.
   i. Let count be count + 1.

19. Return O.

The length property of the copyWithin method is 2.

NOTE 1. The copyWithin 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 copyWithin function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

22.1.3.4 Array.prototype.entries ()

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".

22.1.3.5 Array.prototype.every ( callbackfn, thisArg = undefined )

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 ToLength(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 false, return false.
   e. 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.

22.1.3.6 Array.prototype.fill (value, start = 0, end = this.length)

The fill method takes up to three arguments value, start and end. The start and end arguments are optional
with default values of 0 and the length of the this object. 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. 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 ToLength(lenVal).
5. ReturnIfAbrupt(len).
6. Let relativeStart be ToInteger(start).
7. ReturnIfAbrupt(relativeStart).
8. If relativeStart is negative, let k be max((len + relativeStart),0); else let k be min(relativeStart, len).
9. If end is undefined, let relativeEnd be len; else let relativeEnd be ToInteger(end).
10. ReturnIfAbrupt(relativeEnd).
11. If relativeEnd is negative, let final be max((len + relativeEnd),0); else let final be min(relativeEnd, len).
12. Repeat, while k < final
       a. Let Pk be ToString(k).
       b. Let putStatus be the result of Put(O, Pk, value, true).
       c. ReturnIfAbrupt(putStatus).
       d. Increase k by 1.
13. Return O.

The length property of the fill method is 1.

NOTE 1 The fill 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 fill function can be applied successfully to
an exotic object that is not an Array is implementation-dependent.

22.1.3.7 Array.prototype.filter (callbackfn, thisArg = undefined)

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 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 ToLength(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 undefined.
9. If O is an exotic Array object, then
   a. Let C be Get(O, “constructor”).
   b. ReturnIfAbrupt(C).
   c. If IsConstructor(C) is true, then
      i. Let thisRealm be the running execution context’s Realm.
      ii. If thisRealm and the value of C’s [[Realm]] internal slot are the same value, then
         1. Let A be the result of calling the [[Construct]] internal method of C with an argument list containing the single item 0.
10. If A is undefined, then
    a. Let A be the result of the abstract operation ArrayCreate with argument 0.
11. ReturnIfAbrupt(A).
12. Let k be 0.
13. Let to be 0.
14. 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).
   e. If ToBoolean(selected) is true, then
      1. Let status be the result of CreateDataPropertyOrThrow (A, ToString(to), kValue).
      2. ReturnIfAbrupt(status).
      3. Increase to by 1.
   f. Increase k by 1.
15. 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.
22.1.3.8 Array.prototype.find (predicate, thisArg = undefined)

`predicate` should be a function that accepts three arguments and returns a value that is coercible to the Boolean value `true` or `false`. `find` calls `predicate` once for each element present in the array, in ascending order, until it finds one where `predicate` returns `true`. If such an element is found, `find` immediately returns that element value. Otherwise, `find` returns `undefined`. `predicate` 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 `predicate`. If it is not provided, `undefined` is used instead.

`predicate` is called with three arguments: the value of the element, the index of the element, and the object being traversed.

`find` does not directly mutate the object on which it is called but the object may be mutated by the calls to `predicate`.

The range of elements processed by `find` is set before the first call to `callbackfn`. Elements that are appended to the array after the call to `find` begins will not be visited by `callbackfn`. If existing elements of the array are changed, their value as passed to `predicate` will be the value at the time that `find` visits them; elements that are deleted after the call to `find` begins and before being visited are not visited.

When the `find` 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 ToLength(`lenValue`).
5. ReturnIfAbrupt(`len`).
6. If `IsCallable(`predicate`)` 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 `predicate` with `T` as thisArgument and a List containing `kValue`, `k`, and `O` as argumentsList.
      iv. If ToBoolean(`testResult`) is `true`, return `kValue`.
   e. Increase `k` by 1.
10. Return `undefined`.

The `length` property of the `find` method is `1`.

NOTE The `find` 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 `find` function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

22.1.3.9 Array.prototype.findIndex (predicate, thisArg = undefined)

`predicate` should be a function that accepts three arguments and returns a value that is coercible to the Boolean value `true` or `false`. `findIndex` calls `predicate` once for each element present in the array, in
ascending order, until it finds one where predicate returns true. If such an element is found, findIndex immediately returns the index of that element value. Otherwise, findIndex returns -1. predicate 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 predicate. If it is not provided, undefined is used instead.

predicate is called with three arguments: the value of the element, the index of the element, and the object being traversed.

findIndex does not directly mutate the object on which it is called but the object may be mutated by the calls to predicate.

The range of elements processed by findIndex is set before the first call to callbackfn. Elements that are appended to the array after the call to findIndex begins will not be visited by callbackfn. If existing elements of the array are changed, their value as passed to predicate will be the value at the time that findIndex visits them; elements that are deleted after the call to findIndex begins and before being visited are not visited.

When the findIndex 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 ToLength(lenValue).
5. ReturnIfAbrupt(len).
6. If IsCallable(predicate) 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 predicate with T as thisArgument and a List containing kValue, k, and O as argumentsList.
      iv. ReturnIfAbrupt(testResult).
      v. If ToBoolean(testResult) is true, return k.
   e. Increase k by 1.

The length property of the findIndex method is 1.

NOTE The findIndex 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 findIndex function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

22.1.3.10 Array.prototype.forEach (callbackfn , thisArg = undefined )

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 ToLength(**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.

#### 22.1.3.11 Array.prototype.indexOf ( searchElement [ , fromIndex ] )

**indexOf** compares **searchElement** to the elements of the array, in ascending order, using the Strict Equality Comparison algorithm (7.2.10), 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 ToLength(**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 ≥ len`, return -1.
10. If `n ≥ 0`, then
    a. Let `k` be `n`.
11. Else `n<0`,
    a. Let `k` be `- 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 Strict Equality Comparison `searchElement === elementK`.
        iv. If `same` is true, return `k`.
    d. Increase `k` by 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.

22.1.3.12 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 `ToLength(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.

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.
22.1.3.13 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".

22.1.3.14 Array.prototype.lastIndexOf ( searchElement [ , fromIndex ] )

lastIndexOf compares searchElement to the elements of the array in descending order using the Strict Equality Comparison algorithm (7.2.10), 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 ToLength(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 ≥ 0, then let k be min(n, len - 1).
10. Else n < 0,
   a. Let k be len - abs(n).
11. Repeat, while k ≥ 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 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.

22.1.3.15 Array.prototype.map ( callbackfn, thisArg = undefined )

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 `ToLength(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 `undefined`.
9. If `O` is an exotic Array object, then
   a. Let `C` be `Get(``O``, "constructor")`.
   b. ReturnIfAbrupt(`C`).
   c. If `IsConstructor(C)` is `true`, then
      i. Let `thisRealm` be the running execution context’s Realm.
      ii. If `thisRealm` and the value of `A`’s [[Realm]] internal slot are the same value, then
         1. Let `A` be the result of calling the [[Construct]] internal method of `C` with an argument list containing the single item `len`.
10. If `A` is `undefined`, then
11. Let `A` be the result of the abstract operation `ArrayCreate` with argument `len`.
12. ReturnIfAbrupt(`A`).
13. Let `k` be `0`.
14. 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 this `argumentList` and a List containing `kValue`, `k`, and `O` as `argumentsList`.
      iv. ReturnIfAbrupt(`mappedValue`).
      v. Let `status` be the result of `CreateDataPropertyOrThrow(``A``, `Pk``, mappedValue``)`.
      vi. ReturnIfAbrupt(`status`).
   e. Increase `k` by `1`.
15. 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.
22.1.3.16 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 ToLength(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.

22.1.3.17 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 ToLength(lenVal).
5. ReturnIfAbrupt(n).
6. Let items be a 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.
22.1.3.18 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 ToLength(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, then
      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.

22.1.3.19 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 ToLength(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 ≥ 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
          i. Let accumulator be the result of Get(O, Pk).
          2. ReturnIfAbrupt(accumulator).
       v. Decrease k by 1.
       c. If kPresent is false, throw a TypeError exception.
11. Repeat, while k ≥ 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).

c. 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.

22.1.3.20 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 ToLength(lenVal).
5. Let lower be 0.
6. 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, then
      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.

22.1.3.21 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 ToLength(`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`, false).
       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`, false).
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.

22.1.3.22 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 `lenVal` be the result of Get(`O`, "length").
4. Let `len` be ToLength(`lenVal`).
5. ReturnIfAbrupt(`len`).
6. Let relativeStart be ToInteger(start).
7. ReturnIfAbrupt(relativeStart).
8. If relativeStart is negative, let k be max((len + relativeStart), 0); else let k be min(relativeStart, len).
9. If end is undefined, let relativeEnd be len; else let relativeEnd be ToInteger(end).
10. ReturnIfAbrupt(relativeEnd).
11. If relativeEnd is negative, let final be max((len + relativeEnd), 0); else let final be min(relativeEnd, len).
12. Let count be max(final – k, 0).
13. Let A be undefined.
14. If O is an exotic Array object, then
   a. Let C be the result of Get(O, "constructor").
   b. ReturnIfAbrupt(C).
   c. If IsConstructor(C) is true, then
      i. Let thisRealm be the running execution context’s Realm.
      ii. If thisRealm and the value of C’s [[Realm]] internal slot are the same value, then
         1. Let A be the result of calling the [[Construct]] internal method of C with argument (count).
15. If A is undefined, then
   a. Let A be the result of the abstract operation ArrayCreate with argument count.
16. ReturnIfAbrupt(A).
17. Let n be 0.
18. 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 CreateDataPropertyOrThrow(A, ToString(n), kValue).
      iv. ReturnIfAbrupt(status).
   e. Increase k by 1.
   f. Increase n by 1.
19. Let putStatus be the result of Put(A, "length", n, true).
20. ReturnIfAbrupt(putStatus).
21. 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 19 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.

22.1.3.23 Array.prototype.some (callbackfn, thisArg = undefined)
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 ToLength(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.

22.1.3.24 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.

Upon entry, the following steps are performed to initialize evaluation of the sort function:

1. Let obj be the result of calling ToObject passing the this value as the argument.
2. Let lenValue be the result of Get(obj, "length").
3. Let len be ToLength(lenValue).
4. ReturnIfAbrupt(len).

The result of the sort function is then determined as follows:
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 `[[GetPrototypeOf]]` 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 (22.1)
- `0 \leq j <\text{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 `HasProperty(proto, ToString(i))` is `false`.
- Any array index property of `obj` whose name is a nonnegative integer less than `\text{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 `\text{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 `[[Get]]` and `[[Set]]` 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 `[[Get]]`, `[[Set]]`, or `DeletePropertyOrThrow` is the string representation of a nonnegative integer less than `\text{len}` and where the arguments for calls to `SortCompare` are results of previous calls to the `[[Get]]` internal method. If `obj` is not sparse then `DeletePropertyOrThrow` must not be called. If any `[[Set]]` 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 `\pi` of the nonnegative integers less than `\text{len}`, such that for every nonnegative integer `j` less than `\text{len}`, if property `old[j]` existed, then `new[\pi(j)]` is exactly the same value as `old[j]`. But if property `old[j]` did not exist, then `new[\pi(j)]` does not exist.
- Then for all nonnegative integers `j` and `k`, each less than `\text{len}`, if `SortCompare(j, k) < 0` (see `SortCompare` below), then `\pi(j) < \pi(k)`.

Here the notation `old[j]` is used to refer to the hypothetical result of calling the `[[Get]]` internal method of `obj` with argument `j` before this function is executed, and the notation `new[j]` to refer to the hypothetical result of calling the `[[Set]]` 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 values) in the set `S`: The notation `a \text{<} b` means `comparefn(a, b) < 0`; `a = b` means `comparefn(a, b) = 0` (of either sign); and `a \text{>} 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 \text{<} b`, `a = b`, and `a \text{>} b` will be true for a given pair of `a` and `b`.
- Calling `comparefn(a, b)` does not modify `obj`.
- `a \text{=} a` (reflexivity)
- If `a \text{<} b`, then `b \text{>} a` (symmetry)
- If `a \text{=} b` and `b \text{=} c`, then `a \text{=} c` (transitivity of `\text{=}`)
- If `a \text{<} b` and `b \text{<} c`, then `a \text{<} c` (transitivity of `<`)
- If `a \text{=} b` and `b \text{=} c`, then `a \text{=} c` (transitivity of `\text{=}`)
- If `a \text{<} b` and `b \text{<} c`, then `a \text{<} c` (transitivity of `<`)
NOTE 1  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.

NOTE 2  The assert 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 assert function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

22.1.3.24.1 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 containing 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: 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.

22.1.3.25 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 integer 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 \( lenVal \) be Get(\( O \), "length")
4. Let \( len \) be ToLength(lenVal).
5. ReturnIfAbrupt(\( len \)).
6. Let relativeStart be ToInteger(start).
7. ReturnIfAbrupt(relativeStart).
8. If `relativeStart` is negative, let `actualStart` be `max((len + relativeStart),0)`; else let `actualStart` be `min(relativeStart, len)`.
9. If `start` is not present, then
   a. Let `actualDeleteCount` be 0.
10. Else if `deleteCount` is not present, then
    a. Let `actualDeleteCount` be `len - actualStart`
11. Else,
    a. Let `dc` be `ToInteger(deleteCount)`.
    b. ReturnIfAbrupt(`dc`).
    c. Let `actualDeleteCount` be `min(max(dc,0), len - actualStart)`.
12. Let `A` be `undefined`.
13. If `O` is an exotic Array object, then
   a. Let `C` be the result of `Get(O, "constructor")`.
   b. ReturnIfAbrupt(`C`).
   c. If `IsConstructor(C)` is `true`, then
      i. Let `thisRealm` be the running execution context’s Realm.
      ii. If `thisRealm` and the value of `C`’s [[Realm]] internal slot are the same value, then
         1. Let `A` be the result of calling the `[[Construct]]` internal method of `C` with argument `actualDeleteCount`.
14. If `A` is `undefined`, then
    a. Let `A` be the result of the abstract operation `ArrayCreate` with argument `actualDeleteCount`.
15. ReturnIfAbrupt(`A`).
16. Let `k` be 0.
17. 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. Let `status` be the result of `CreateDataPropertyOrThrow(A, ToString(k), fromValue)`.
      iv. ReturnIfAbrupt(`status`).
   e. Increment `k` by 1.
18. Let `putStatus` be the result of `Put(A, "length", actualDeleteCount, true)`.
19. ReturnIfAbrupt(`putStatus`).
20. Let `items` be a 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.
21. Let `itemCount` be the number of elements in `items`.
22. 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, from)`.
           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.
23. Else if `itemCount > actualDeleteCount`, then

Commented [AWB7121]: It would be nice to have a more explicit way to create a collection with a pre-specified number of elements.
a. Let \( k \) be \((\text{len} – \text{actualDeleteCount})\).
b. Repeat, while \( k > \text{actualStart} \)
   i. Let \( \text{from} \) be \(\text{ToString}(k + \text{actualDeleteCount} – 1)\).
   ii. Let \( \text{to} \) be \(\text{ToString}(k + \itemCount – 1)\)
   iii. Let \( \text{fromPresent} \) be the result of \(\text{HasProperty}(O, \text{from})\).
   iv. ReturnIfAbrupt(\(\text{fromPresent}\)).
   v. If \( \text{fromPresent} \) is \(\text{true}\), then
      1. Let \( \text{fromValue} \) be the result of \(\text{Get}(O, \text{from})\).
      2. ReturnIfAbrupt(\(\text{fromValue}\)).
      3. Let \( \text{putStatus} \) be the result of \(\text{Put}(O, \text{to}, \text{fromValue}, \text{true})\).
      4. ReturnIfAbrupt(\(\text{putStatus}\)).
   vi. Else \( \text{fromPresent} \) is \(\text{false}\),
      1. Let \( \text{deleteStatus} \) be the result of \(\text{DeletePropertyOrThrow}(O, \text{to})\).
      2. ReturnIfAbrupt(\(\text{deleteStatus}\)).
   vii. Decrease \( k \) by 1.
24. Let \( k \) be \(\text{actualStart}\).
25. Repeat, while \( \text{items} \) is not empty
   a. Remove the first element from \(\text{items} \) and let \( E \) be the value of that element.
   b. Let \( \text{putStatus} \) be the result of \(\text{Put}(O, \text{ToString}(k), E, \text{true})\).
   c. ReturnIfAbrupt(\(\text{putStatus}\)).
   d. Increase \( k \) by 1.
26. Let \( \text{putStatus} \) be the result of \(\text{Put}(O, \text{“length”}, \text{len} – \text{actualDeleteCount} + \itemCount, \text{true})\).
27. ReturnIfAbrupt(\(\text{putStatus}\)).
28. Return \( A \).

The \text{length} property of the \text{splice} method is \text{2}.

NOTE 1  The explicit setting of the \text{length} property of the result Array in step 18 is necessary to ensure that its value is correct in situations where its trailing elements are not present.

NOTE 2  The \text{splice} function is intentionally generic; it does not require that its \text{this} value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the \text{splice} function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

22.1.3.26 Array.prototype.toLocaleString ( )

The elements of the array are converted to Strings using their \text{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 \text{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 \text{ToObject} passing the \text{this} value as the argument.
2. ReturnIfAbrupt(\(\text{array}\)).
3. Let \( \text{arrayLen} \) be the result of \(\text{Get}(\text{array}, \text{“length”})\).
4. Let \( \text{len} \) be \(\text{ToLength}(\text{arrayLen})\).
5. ReturnIfAbrupt(\(\text{len}\)).
6. Let \( \text{separator} \) 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 \( \text{len} \) is zero, return the empty String.
8. Let \( \text{firstElement} \) be the result of \(\text{Get}(\text{array}, \text{“0”})\).
9. ReturnIfAbrupt(\(\text{firstElement}\)).
10. If \( \text{firstElement} \) is \text{undefined} or \text{null}, then
    a. Let \( R \) be the empty String.
11. Else
a. Let \( R \) be the result of Invoke(firstElement, "toLocaleString").  
b. Let \( R \) be ToString(R).  
c. ReturnIfAbrupt(R).

12. Let \( k \) be 1.

13. Repeat, while \( k < \text{len} \)
   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.
      ii. 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.

14. Return \( R \).

NOTE 1  The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

NOTE 2  The toLocaleString function is intentionally generic; it does not require that its this value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the toLocaleString function can be applied successfully to an exotic object that is not an Array is implementation-dependent.

22.1.3.27 Array.prototype.toString ()

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

1. Let array be the result of calling ToObject passing the this value
2. ReturnIfAbrupt(array).
3. Let func be the result of Get(array, "join").
4. ReturnIfAbrupt(func).
5. If IsCallable(func) is false, then let func be the standard built-in method Object.prototype.toString (19.1.3.6).
6. 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.

22.1.3.28 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 ToLength(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).

© Ecma International 2009 – All rights reserved
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 a 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).


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.

22.1.3.29 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”.

22.1.3.30 Array.prototype [ @@iterator ] ( )

The initial value of the @@iterator property is the same function object as the initial value of the Array.prototype.values property.

22.1.3.31 Array.prototype [ @@unscopables ]

The initial value of the @@unscopables data property is an object created by the following steps:

1. Let blackList be the result of calling ArrayCreate(7, %ArrayPrototype%).
2. Call CreateDataProperty(blackList, "0", "find").
3. Call CreateDataProperty(blackList, "1", "findIndex").
4. Call CreateDataProperty(blackList, "2", "fill").
5. Call CreateDataProperty(blackList, "3", "copyWithin").
6. Call CreateDataProperty(blackList, "4", "entries").
7. Call CreateDataProperty(blackList, "5", "keys").
8. Call CreateDataProperty(blackList, "6", "values").
9. Assert: Each of the above calls will return true.

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

NOTE The elements of this array are property names that were not included as standard properties of `Array.prototype` prior to the sixth edition of this specification. These names are ignored for `with` statement binding purposes in order to preserve the behaviour of existing code that might use one of these names as a binding in an outer scope that is shadowed by a `with` statement whose binding object is an `Array` object.

22.1.4 Properties of Array Instances

Array instances are exotic `Array` objects and have the internal methods specified for such objects. `Array` instances inherit properties from the `Array.prototype` object. `Array` instances also have an [[ArrayInitialisationState]] internal slot.

Array instances have a `length` property, and a set of enumerable properties with array index names.

22.1.4.1 `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 9.4.2.1.

22.1.5 Array Iterator Objects

An `Array` iterator is an object, that represents 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.

22.1.5.1 CreateArrayIterator Abstract Operation

Several methods of `Array` objects return `Iterator` objects. The abstract operation `CreateArrayIterator` with arguments `array` and `kind` is used to create such iterator objects. It performs the following steps:

1. Let `O` be the result of calling `ToObject(array)`.
2. ReturnIfAbrupt(`O`).
3. Let `iterator` be the result of `ObjectCreate(%ArrayIteratorPrototype%, ([[IteratedObject]], [[ArrayIteratorNextIndex]], [[ArrayIterationKind]]))`.
4. Set `iterator`'s [[IteratedObject]] internal slot to `O`.
5. Set `iterator`'s [[ArrayIteratorNextIndex]] internal slot to 0.
6. Set `iterator`'s [[ArrayIterationKind]] internal slot to `kind`.
7. Return `iterator`.

22.1.5.2 The `%ArrayIteratorPrototype% Object

All `Array` Iterator Objects inherit properties from the `%ArrayIteratorPrototype% intrinsic object. The `%ArrayIteratorPrototype% object is an ordinary object and its [[Prototype]] internal slot is the `%ObjectPrototype%` intrinsic object. In addition, `%ArrayIteratorPrototype%` has the following properties:

22.1.5.2.1 `%ArrayIteratorPrototype%. 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 slots of an Array Iterator Instance (22.1.5.3), throw a TypeError exception.
4. Let a be the value of the [[IteratedObject]] internal slot of O.
5. If a is undefined, then return CreateIterResultObject(undefined, true).
6. Let index be the value of the [[ArrayIteratorNextIndex]] internal slot of O.
7. Let itemKind be the value of the [[ArrayIterationKind]] internal slot of O.
8. Let lenValue be the result of Get(a, "length").
9. Let len be ToLength(lenValue).
10. ReturnIfAbrupt(len).
11. If index ≥ len, then
   a. Set the value of the [[IteratedObject]] internal slot of O to undefined.
   b. Return CreateIterResultObject(undefined, true).
12. Set the value of the [[ArrayIteratorNextIndex]] internal slot of O to index + 1.
13. If itemKind contains the substring "value", then
   a. Let elementKey be ToString(index).
   b. Let elementValue be the result of Get(a, elementKey).
   c. ReturnIfAbrupt(elementValue).
14. If itemKind contains the substring "key+value", then
   a. Let result be the result of performing ArrayCreate(2).
   b. Assert: result is a new, well-formed Array object so the following operations will never fail.
   c. Call CreateDataProperty(result, "0", index).
   d. Call CreateDataProperty(result, "1", elementValue).
   e. Return CreateIterResultObject(result, false).
15. Else if itemKind contains the substring "key" then, return CreateIterResultObject(index, false).
16. Assert: itemKind contains the substring "value".
17. Return CreateIterResultObject(elementValue, false).

22.1.5.2.2 %ArrayIteratorPrototype%[@@iterator]()

The following steps are taken:
1. Return the this value.

The value of the name property of this function is "[Symbol.iterator]".

22.1.5.2.3 %ArrayIteratorPrototype%[@@toStringTag]

The initial value of the @@toStringTag property is the string value "Array Iterator".

22.1.5.3 Properties of Array Iterator Instances

Array Iterator instances are ordinary objects that inherit properties from the %ArrayIteratorPrototype% intrinsic object. Array Iterator instances are initially created with the internal slots listed in Table 35.

Table 35 — Internal Slots of Array Iterator Instances

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[IteratedObject]]</td>
<td>The object whose array elements are being iterated.</td>
</tr>
<tr>
<td>[[ArrayIteratorNextIndex]]</td>
<td>The integer index of the next array index to be examined by this iteration.</td>
</tr>
<tr>
<td>[[ArrayIterationKind]]</td>
<td>A string value that identifies what is to be returned for each element of the iteration. The possible values are: &quot;key&quot;, &quot;value&quot;, &quot;key+value&quot;, &quot;sparse: key&quot;, &quot;sparse: value&quot;, &quot;sparse: key+value&quot;.</td>
</tr>
</tbody>
</table>
22.2 TypedArray Objects

TypedArray objects present an array-like view of an underlying binary data buffer (24.1). Each element of a TypedArray instance has the same underlying binary scalar data type. There is a distinct TypedArray constructor. Listed in Table 36, for each of the nine supported element types. For each constructor in Table 36 have a corresponding distinct prototype object.

<table>
<thead>
<tr>
<th>Constructor Name</th>
<th>Element Type</th>
<th>Element Size</th>
<th>Conversion Operation</th>
<th>Description</th>
<th>Equivalent C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int8Array</td>
<td>Int8</td>
<td>1</td>
<td>ToInt8</td>
<td>8-bit 2's complement signed integer</td>
<td>signed char</td>
</tr>
<tr>
<td>Uint8Array</td>
<td>Uint8</td>
<td>1</td>
<td>ToUint8</td>
<td>8-bit unsigned integer</td>
<td>unsigned char</td>
</tr>
<tr>
<td>Uint8ClampedArray</td>
<td>Uint8C</td>
<td>1</td>
<td>ToUint8Clamp</td>
<td>8-bit unsigned integer (clamped conversion)</td>
<td>unsigned char</td>
</tr>
<tr>
<td>Int16Array</td>
<td>Int16</td>
<td>2</td>
<td>ToInt16</td>
<td>16-bit 2's complement signed integer</td>
<td>Short</td>
</tr>
<tr>
<td>Uint16Array</td>
<td>Uint16</td>
<td>2</td>
<td>ToUint16</td>
<td>16-bit unsigned integer</td>
<td>unsigned short</td>
</tr>
<tr>
<td>Int32Array</td>
<td>Int32</td>
<td>4</td>
<td>ToInt32</td>
<td>32-bit 2's complement signed integer</td>
<td>Int</td>
</tr>
<tr>
<td>Uint32Array</td>
<td>Uint32</td>
<td>4</td>
<td>ToUint32</td>
<td>32-bit unsigned integer</td>
<td>unsigned int</td>
</tr>
<tr>
<td>Float32Array</td>
<td>Float32</td>
<td>4</td>
<td></td>
<td>32-bit IEEE floating point</td>
<td>Float</td>
</tr>
<tr>
<td>Float64Array</td>
<td>Float64</td>
<td>8</td>
<td></td>
<td>64-bit IEEE floating point</td>
<td>Double</td>
</tr>
</tbody>
</table>

In the definitions below, references to TypedArray 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.

22.2.1 The %TypedArray% Intrinsic Object

The %TypedArray% intrinsic object is a constructor-like function object that all of the TypedArray constructor object inherit from. %TypedArray% and its corresponding prototype object provide common properties that are inherited by all TypedArray constructors and their instances. The %TypedArray% intrinsic does not have a global name or appear as a property of the global object.

If the this value passed in the call is an Object with a [[ViewedArrayBuffer]] internal slot whose value is undefined, it initialises the this value using the argument values. This permits super invocation of the TypedArray constructors by TypedArray subclasses.

The %TypedArray% intrinsic function object is designed to act as the superclass of the various TypedArray constructors. Those constructors use %TypedArray% to initialise their instances by invoking %TypedArray% as if by making a super call. The %TypedArray% intrinsic function is not designed to be directly called in any other way. If %TypedArray% is directly called or called as part of a new expression an exception is thrown.

The actual behaviour of a super call of %TypedArray% depends upon the number and kind of arguments that are passed to it.

22.2.1.1 %TypedArray% ( length )

This description applies if and only when %TypedArray% function is called and the Type of the first argument is not Object.
%TypedArray% called with argument length performs the following steps:

1. Assert: Type(length) is not Object.
2. Let O be the this value.
3. If Type(O) is not Object, then throw a TypeError exception.
4. If O does not have a [[TypedArrayName]] internal slot, then throw a TypeError exception.
5. If the value of O’s [[TypedArrayName]] internal slot is undefined, then throw a TypeError exception.
6. Assert: O has a [[ViewedArrayBuffer]] internal slot.
7. If the value of O’s [[ViewedArrayBuffer]] internal slot is not undefined, then throw a TypeError exception.
   Let constructorName be the string value O’s [[TypedArrayName]] internal slot.
8. Let elementType be the string value of the Element Type value in Table 36 for constructorName.
9. Let numberOfLength be ToNumber(length).
10. Let elementLength be ToLength(numberLength).
11. ReturnIfAbrupt(elementLength).
12. If SameValueZero(numberLength, elementLength) is false, then throw a RangeError exception.
13. Let data be the result of calling Allocate ArrayBuffer( %ArrayBuffer% ).
14. ReturnIfAbrupt(data).
15. Let byteLength be the Size Element value in Table 36 for constructorName.
16. Let byteLength be elementSize x elementLength.
17. Let status be the result of Set ArrayBufferData( data, byteLength ).
18. ReturnIfAbrupt(status).
19. Set O’s [[ViewedArrayBuffer]] to data.
20. Set O’s [[ByteLength]] internal slot to byteLength.
21. Set O’s [[ByteOffset]] internal slot to 0.
22. Set O’s [[ArrayLength]] internal slot to elementLength.
23. Return O.

22.2.1.2 %TypedArray% ( typedArray )

This description applies if and only if the %TypedArray% function is called with at least one argument and the Type of the first argument is Object and that object has a [[TypedArrayName]] internal slot.

%TypedArray% called with argument typedArray performs the following steps:

1. Assert: Type(typedArray) is Object and typedArray has a [[TypedArrayName]] internal slot.
2. Let srcArray be typedArray.
3. Let O be the this value.
4. If Type(O) is not Object or O does not have a [[TypedArrayName]] internal slot, then throw a TypeError exception.
5. If the value of O’s [[TypedArrayName]] internal slot is undefined, then throw a TypeError exception.
6. Assert: O has a [[ViewedArrayBuffer]] internal slot.
7. If the value of O’s [[ViewedArrayBuffer]] internal slot is not undefined, then throw a TypeError exception.
8. If the value of srcArray’s [[ViewedArrayBuffer]] internal slot is undefined, then throw a TypeError exception.
9. Let constructorName be the string value O’s [[TypedArrayName]] internal slot.
10. Let elementType be the string value of the Element Type value in Table 36 for constructorName.
11. Let elementLength be the value of srcArray’s [[ArrayLength]] internal slot.
12. Let srcName be the string value srcArray’s [[TypedArrayName]] internal slot.
13. Let srcType be the string value of the Element Type value in Table 36 for srcName.
14. Let srcElementSize be the Size Element value in Table 36 for srcName.
15. Let srcData be the value of srcArray’s [[ViewedArrayBuffer]] internal slot.
16. Let srcByteOffset be the value of srcArray’s [[ByteOffset]] internal slot.
17. Let elementSize be the Size Element value in Table 36 for constructorName.
18. Let byteLength be elementSize x elementLength.
19.
20.
21. If SameValue(elementType, srcType), then
   a. Let data be the result of calling CloneArrayBuffer(srcData, srcByteOffset).
      b. ReturnIfAbrupt(data).
22. Else
   a. Let bufferConstructor be the result of Get(srcBuffer, "constructor").
      b. ReturnIfAbrupt(bufferConstructor).
   c. If bufferConstructor is undefined, then let bufferConstructor be %ArrayBuffer%.
      d. Let data be the result of calling AllocateArrayBuffer(bufferConstructor).
      e. Let status be the result of SetArrayBufferData(data, byteLength).
      f. ReturnIfAbrupt(status).
   g. Let srcByteIndex be srcByteOffset.
      h. Let targetByteIndex be 0.
      i. Let count be elementLength.
      j. Repeat, count > 0
         i. Let value be the result of GetValueFromBuffer(srcData, srcByteIndex, srcType).
         ii. Let status be the result of SetValueInBuffer(data, targetByteIndex, elementType, value).
         iii. Set srcByteIndex to srcByteIndex + targetByteIndex.
         iv. Set targetByteIndex to targetByteIndex + elementSize.
      v. Decrement count by 1.
24. If the value of O's [[ViewedArrayBuffer]] internal slot is not undefined, then throw a TypeError exception.
25. Assert: O has not been reentrantly initialized.
26. Set O's [[ViewedArrayBuffer]] internal slot to data.
27. Set O's [[ByteLength]] internal slot to byteLength.
28. Set O's [[ByteOffset]] internal slot to 0.
29. Set O's [[ArrayLength]] internal slot to elementLength.
30. Return O.

22.2.1.3 %TypedArray% ([array])

This description applies if and only if the %TypedArray% function is called with at least one argument and the type of first argument is Object and that object does not have either a [[TypedArrayName]] or an [[ArrayBufferData]] internal slot.

%TypedArray% called with argument array performs the following steps:

1. Assert: Type(array) is Object and array does not have either a [[TypedArrayName]] or an [[ArrayBufferData]] internal slot.
2. Let O be the this value.
3. Let srcArray be array.
4. If Type(O) is not Object or if O does not have a [[TypedArrayName]] internal slot, then throw a TypeError exception.
5. If the value of O's [[TypedArrayName]] internal slot is undefined, then throw a TypeError exception.
6. Assert: O has a [[ViewedArrayBuffer]] internal slot.
7. If the value of O's [[ViewedArrayBuffer]] internal slot is not undefined, then throw a TypeError exception.
8. Let constructorName be the string value O's [[TypedArrayName]] internal slot.
9. Let elementType be the string value of the Element Type value in Table 36 for constructorName.
10. Let arrayLength be the result of Get(srcArray, "length").
11. Let elementLength be ToLength(arrayLength).
12. ReturnIfAbrupt(elementLength).
13. Let data be the result of calling AllocateArrayBuffer(%ArrayBuffer%),
14. ReturnIfAbrupt(data).
15. Let elementSize be the Size Element value in Table 36 for constructorName.
16. Let byteLength be elementSize × elementLength.
17. Let status be the result of SetArrayBufferData(data, byteLength)
18. ReturnIfAbrupt(status).
19. Let k be 0.
20. Repeat, while k < elementLength
   a. LetPk be ToString(k).
   b. Let kValue be the result of Get(srcArray, Pk).
c. Let kNumber be ToNumber(kValue).
d. ReturnIfAbrupt(kNumber).
e. Perform SetValueInBuffer(data, k × elementSize, elementType, kNumber).
f. Increase k by 1.

21. Note: Side-effects of preceding steps may have already initialised O.
22. If the value of O’s [[ViewedArrayBuffer]] internal slot is not undefined, then throw a TypeError exception.
23. Set O’s [[ViewedArrayBuffer]] to data.
24. Set O’s [[ByteLength]] internal slot to byteLength.
25. Set O’s [[ByteOffset]] internal slot to 0.
26. Set O’s [[ArrayLength]] internal slot to elementLength.
27. Return O.

22.2.1.4 %TypedArray% ( buffer, byteOffset=0, length=undefined )

This description applies if and only if the %TypedArray% function is called with at least one argument and the Type of the first argument is Object and that object has a `constructorName` property.

%TypedArray% called with arguments buffer, byteOffset, and length performs the following steps:

1. Assert: `buffer` is Object and buffer has an [[ArrayBufferData]] internal slot.
2. Let O be the this value.
3. If the value of buffer’s [[ArrayBufferData]] internal slot is undefined, then throw a TypeError exception.
4. If Type(O) is not Object or if O does not have a [[TypedArrayName]] internal slot, then throw a TypeError exception.
5. If the value of O’s [[TypedArrayName]] internal slot is undefined, then throw a TypeError exception.
6. Assert: O has a [[ViewedArrayBuffer]] internal slot.
7. If the value of O’s [[ViewedArrayBuffer]] internal slot is not undefined, then throw a TypeError exception.
8. Let constructorName be the string value O’s [[TypedArrayName]] internal slot.
9. Let elementType be the string value of the Element Type value in Table 36 for constructorName.
10. Let elementSize be the Number value of the Element Size value in Table 36 for constructorName.
11. Let offset be ToInteger(byteOffset).
12. ReturnIfAbrupt(offset).
13. If offset < 0, then throw a RangeError exception.
14. If offset modulo elementSize ≠ 0, then throw a RangeError exception.
15. Let bufferByteLength be the value of buffer’s [[ArrayBufferByteLength]] internal slot.
16. If offset + elementSize > bufferByteLength, then throw a RangeError exception.
17. If length is undefined, then
   a. If bufferByteLength modulo elementSize ≠ 0, then throw a RangeError exception.
   b. Let newByteLength be bufferByteLength – offset.
   c. If newByteLength < 0, then throw a RangeError exception.
18. Else
   a. Let newLength be ToLength(length).
   b. ReturnIfAbrupt(newLength).
   c. Let newByteLength be newLength × elementSize.
   d. If offset + newByteLength > bufferByteLength, then throw a RangeError exception.
19. If the value of O’s [[ViewedArrayBuffer]] internal slot is not undefined, then throw a TypeError exception.
20. Set O’s [[ViewedArrayBuffer]] to buffer.
21. Set O’s [[ByteLength]] internal slot to newByteLength.
22. Set O’s [[ByteOffset]] internal slot to offset.
23. Set O’s [[ArrayLength]] internal slot to newByteLength / elementSize.
24. Return O.

22.2.1.5 %TypedArray% ( all other argument combinations )

If the %TypedArray% function is called with arguments that do not match any of the preceding argument descriptions a TypeError exception is thrown.

Commented [AWB13126]: Issue: we need to decide whether we are going to fully apply WebIDL’s complex set of overload resolution rules.

Commented [AWB13127]: Issue: the order of exceptions that may be thrown during argument validation is probably not currently the same as would be produced by WebIDL overloading plus Khronos validation rules (if that order is even well defined??). It isn’t clear that this difference is very important.


22.2.2 Properties of the %TypedArray% Intrinsic Object

The %TypedArray% intrinsic object is a built-in function object. The value of the [[Prototype]] internal slot of %TypedArray% is the Function prototype object (19.2.3).

Besides a length property whose value is 3 and a name property whose value is "TypedArray", %TypedArray% has the following properties:

22.2.2.1 %TypedArray% from ( source , mapfn=undefined, thisArg=undefined )

When the from method is called with argument source, and optional arguments mapfn and thisArg, the following steps are taken:

1. Let C be the this value.
2. If IsConstructor(C) is false, then throw a TypeError exception.
3. Let items be ToObject(source).
4. ReturnIfAbrupt(items).
5. If mapfn is undefined, then let mapping be false.
6. else
   a. If IsCallable(mapfn) is false, throw a TypeError exception.
   b. If thisArg was supplied, let T be thisArg; else let T be the undefined.
   c. Let mapping be true.
7. Let usingIterator be the result of HasProperty(items, @@iterator).
8. ReturnIfAbrupt(usingIterator).
9. If usingIterator is true, then
   a. Let iterator be the result of performing GetIterator(items).
   b. ReturnIfAbrupt(iterator).
   c. Let values be a new empty List.
   d. Let next be true.
   e. Repeat, while next is not false
      i. Let next be the result of IteratorStep(iterator).
      ii. ReturnIfAbrupt(next).
      iii. If next is not false, then
           1. Let nextValue be IteratorValue(next).
           2. ReturnIfAbrupt(nextValue).
           3. Append nextValue to the end of the List values.
   f. Let len be the number of elements in values.
   g. Let newObj be the result of calling the [[Construct]] internal method of C with argument (len).
   h. ReturnIfAbrupt(newObj).
   i. Let k be 0.
   j. Repeat, while k < len
      i. Let Pk be ToString(k).
      ii. Let kValue be the first element of values and remove that element from list.
      iii. If mapping is true, then
           1. Let mappedValue be the result of calling the [[Call]] internal method of mapfn with T as thisArgument and a List containing kValue as argumentsList.
           2. ReturnIfAbrupt(mappedValue).
      iv. Else, let mappedValue be kValue.
      v. Let putStatus be the result of Put(newObj, Pk, mappedValue, true).
      vi. ReturnIfAbrupt(putStatus).
   k. Assert: values is now an empty List.
   l. Return newObj.
10. Assert: items is not an iterator so assume it is an array-like object.
11. Let lenValue be the result of Get(items, "length").
12. Let len be ToLength(lenValue).
13. ReturnIfAbrupt(len).
14. Let newObj be the result of calling the [[Construct]] internal method of C with argument (len).
15. ReturnIfAbrupt(newObj).

Commented [AWB7128]: It would be nice to have a more explicit way to create a collection with a pre-specified number of elements.

Commented [AWB7129]: It would be nice to have a more explicit way to create a collection with a pre-specified number of elements.
16. Let \( k \) be 0.
17. Repeat, while \( k < \text{len} \)
   a. Let \( \text{Pk} \) be ToString(\( k \)).
   b. Let \( \text{kValue} \) be the result of Get(\( \text{items}, \text{Pk} \)).
   c. ReturnIfAbrupt(\( \text{kValue} \)).
   d. If mapping is true, then
      i. Let \( \text{mappedValue} \) be the result of calling the \([\text{Call}]\) internal method of \( \text{mapfn} \) with \( T \) as thisArgument and a List containing \( \text{kValue}, k \), and \( \text{items} \) as argumentsList.
      ii. ReturnIfAbrupt(\( \text{mappedValue} \)).
   e. Else, let \( \text{mappedValue} \) be \( \text{kValue} \).
   f. Let \( \text{putStatus} \) be the result of Put(\( \text{newObj}, \text{Pk}, \text{mappedValue}, \text{true} \)).
   g. ReturnIfAbrupt(\( \text{putStatus} \)).
   h. Increase \( k \) by 1.
18. Return \( \text{newObj} \).

The length property of the from method is 1.

NOTE The from function is an intentionally generic factory method; it does not require that its this value be a TypedArray constructor. Therefore it can be transferred to or inherited by any other constructors that may be called with a single numeric argument. This function uses \([\text{Put}]\) to store elements into a newly created object and assume that the constructor sets the length property of the new object to the argument value passed to it.

22.2.2.2 `%TypedArray%.of ( ...items )`

When the of method is called with any number of arguments, the following steps are taken:
1. Let \( \text{lenValue} \) be the result of Get(\( \text{items}, \text{"length"} \)).
2. Let \( \text{len} \) be ToLength(\( \text{lenValue} \)).
3. Let \( C \) be the this value.
4. If IsConstructor(\( C \)) is true, then
   a. Let \( \text{newObj} \) be the result of calling the \([\text{Construct}]\) internal method of \( C \) with argument \( \text{len} \).
   b. ReturnIfAbrupt(\( \text{newObj} \)).
5. Else,
   a. Throw a TypeError exception.
6. Let \( k \) be 0.
7. Repeat, while \( k < \text{len} \)
   a. Let \( \text{Pk} \) be ToString(\( k \)).
   b. Let \( \text{kValue} \) be the result of Get(\( \text{items}, \text{Pk} \)).
   c. Let \( \text{status} \) be the result of Put(\( \text{newObj}, \text{Pk}, \text{kValue}[[\text{value}]], \text{true} \)).
   d. ReturnIfAbrupt(\( \text{status} \)).
   e. Increase \( k \) by 1.
8. Return \( \text{newObj} \).

The length property of the of method is 0.

NOTE 1 The items argument is assumed to be a well-formed rest argument value.

NOTE 2 The of function is an intentionally generic factory method; it does not require that its this value be a TypedArray constructor. Therefore it can be transferred to or inherited by other constructors that may be called with a single numeric argument. However, it does assume that the constructor creates and initialises a length property that is initialised to its argument value.

22.2.2.3 `%TypedArray%.prototype`

The initial value of `%TypedArray%.prototype` is the `%TypedArrayPrototype%` intrinsic object (22.2.3).

This property has the attributes ([Writable]: false, [[Enumerable]]: false, [[Configurable]]: false).
22.2.2.4 %TypedArray% [[@@create]]()

The [[@create]] method of %TypedArray% performs the following steps:

1. Let F be the this value.
2. If Type(F) is not Object, then throw a TypeError exception.
3. Let proto be the result of GetPrototypeOf(F, "%TypedArrayPrototype").
4. ReturnAbrupt(proto).
5. Let obj be the result of calling ObjectCreate (proto).
6. Add a [[ViewedArrayBuffer]] internal slot to obj and set its initial value to undefined.
7. Add a [[TypedArrayName]] internal slot to obj and set its initial value to undefined.
8. Add a [[ByteLength]] internal slot to obj and set its initial value to 0.
9. Add a [[ByteOffset]] internal slot to obj and set its initial value to 0.
10. Add an [[ArrayLength]] internal slot to obj and set its initial value to 0.
11. Return obj.

The value of the name property of this function is "[[Symbol.create]]."

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

22.2.3 Properties of the %TypedArrayPrototype% Object

The value of the [[Prototype]] internal slot of the %TypedArrayPrototype% object is the standard built-in Object prototype object (19.1.3). The %TypedArrayPrototype% object is an ordinary object. It does not have a [[ViewedArrayBuffer]] or or any other of the internal slots that are specific to %TypedArray% instance objects.

22.2.3.1 get %TypedArray%.prototype.buffer

%TypedArray%.prototype.buffer is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let O be the this value.
2. If Type(O) is not Object, throw a TypeError exception.
3. If O does not have a [[ViewedArrayBuffer]] internal slot throw a TypeError exception.
4. Let buffer be the value of O’s [[ViewedArrayBuffer]] internal slot.
5. If buffer is undefined, then throw a TypeError exception.
6. Return buffer.

22.2.3.2 get %TypedArray%.prototype.byteLength

%TypedArray%.prototype.byteLength is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let O be the this value.
2. If Type(O) is not Object, throw a TypeError exception.
3. If O does not have a [[ViewedArrayBuffer]] internal slot throw a TypeError exception.
4. Let byteBuffer be the value of O’s [[ViewedArrayBuffer]] internal slot.
5. If buffer is undefined, then throw a TypeError exception.
6. Let size be the value of O’s [[ByteLength]] internal slot.
7. Return size.

22.2.3.3 get %TypedArray%.prototype.byteOffset

%TypedArray%.prototype.byteOffset is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let O be the this value.
2. If Type(O) is not Object, throw a TypeError exception.
3. If O does not have a [[ViewedArrayBuffer]] internal slot throw a TypeError exception.
4. Let `buffer` be the value of `O`'s `[[ViewedArrayBuffer]]` internal slot.
5. If `buffer` is `undefined`, then throw a `TypeError` exception.
6. Let `offset` be the value of `O`'s `[[ByteOffset]]` internal slot.
7. Return `offset`.

22.2.3.4 `%TypedArray%.prototype.constructor`

The initial value of `%TypedArray%.prototype.constructor` is the `%TypedArray%` intrinsic object.

22.2.3.5 `%TypedArray%.prototype.copyWithin(target, start, end = this.length)`

%TypedArray%.prototype.copyWithin is a distinct function that implements the same algorithm as `Array.prototype.copyWithin` as defined in 22.1.3.3 except that the this object's `[[ArrayLength]]` internal slot is accessed in place of performing a `[[Get]]` of "length". The implementation of the algorithm may be optimized with the knowledge that the this value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm.

This function is not generic. If the this value is not a object with a `[[TypedArrayName]]` internal slot, a `TypeError` exception is immediately thrown when this function is called.

The `length` property of the copyWithin method is 2.

22.2.3.6 `%TypedArray%.prototype.entries()`

The initial value of the `%TypedArray%.prototype.entries` data property is the same built-in function object as the `Array.prototype.entries` method defined in 22.1.3.4.

22.2.3.7 `%TypedArray%.prototype.every(callbackfn, thisArg = undefined)`

%TypedArray%.prototype.every is a distinct function that implements the same algorithm as `Array.prototype.every` as defined in 22.1.3.5 except that the this object's `[[ArrayLength]]` internal slot is accessed in place of performing a `[[Get]]` of "length". The implementation of the algorithm may be optimized with the knowledge that the this value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm.

This function is not generic. If the this value is not a object with a `[[TypedArrayName]]` internal slot, a `TypeError` exception is immediately thrown when this function is called.

The `length` property of the every method is 1.

22.2.3.8 `%TypedArray%.prototype.fill(value, start = 0, end = this.length)`

%TypedArray%.prototype.fill is a distinct function that implements the same algorithm as `Array.prototype.fill` as defined in 22.1.3.6 except that the this object's `[[ArrayLength]]` internal slot is accessed in place of performing a `[[Get]]` of "length". The implementation of the algorithm may be optimized with the knowledge that the this value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm.

This function is not generic. If the this value is not a object with a `[[TypedArrayName]]` internal slot, a `TypeError` exception is immediately thrown when this function is called.

The `length` property of the fill method is 1.
22.2.3.9 `%TypedArray%.prototype.filter (callbackfn, thisArg = undefined)`

The interpretation and use of the arguments of `%TypedArray%.prototype.filter` are the same as for `Array.prototype.filter` as defined in 22.1.3.7.

When the `filter` method is called with one or two arguments, the following steps are taken:

1. Let `O` be the `this` value.
2. If `Type(O)` is not `Object`, throw a `TypeError` exception.
3. If `O` does not have a `[[TypedArrayName]]` internal slot, then throw a `TypeError` exception.
4. Let `len` be the value of `O`'s `[[ArrayLength]]` internal slot.
5. If `IsCallable(callbackfn)` is false, throw a `TypeError` exception.
6. If `thisArg` was supplied, let `T` be `thisArg`; else let `T` be `undefined`.
7. Let `C` be the result of `Get(O, "constructor")`.
8. ReturnIfAbrupt(`C`).
9. If `IsConstructor(C)` is false, then
   a. Throw a `TypeError` exception.
10. Let `kept` be a new empty List.
11. Let `k` be 0.
12. Let `captured` be 0.
13. Repeat, while `k < len`
    a. Let `Pk` be `ToString(k)`.
    b. Let `kValue` be the result of `Get(O, Pk)`.
    c. ReturnIfAbrupt(`kValue`).
    d. 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`.
    e. ReturnIfAbrupt(`selected`).
    f. If `ToBoolean(selected)` is true, then
       i. Append `kValue` to the end of `kept`.
       ii. Increase `captured` by 1.
    g. Increase `k` by 1.
14. Let `A` be the result of calling the `[[Construct]]` internal method of `C` with argument `(captured)`.
15. ReturnIfAbrupt(`A`).
16. Let `n` be 0.
17. For each element `e` of `kept`
    a. Let `status` be the result of `Put(A, ToString(n), e, true)`.
    b. ReturnIfAbrupt(`status`).
    c. Increment `n` by 1.
18. Return `A`.

This function is not generic. If the `this` value is not a object with a `[[TypedArrayName]]` internal slot, a `TypeError` exception is immediately thrown when this function is called.

The `length` property of the `filter` method is 1.

22.2.3.10 `%TypedArray%.prototype.find (predicate, thisArg = undefined)`

 `%TypedArray%.prototype.find` is a distinct function that implements the same algorithm as `Array.prototype.find` as defined in 22.1.3.8 except that the `this` object's `[[ArrayLength]]` internal slot is accessed in place of performing a `[[Get]]` of "`length`". The implementation of the algorithm may be optimized with the knowledge that the `this` value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm.

This function is not generic. If the `this` value is not a object with a `[[TypedArrayName]]` internal slot, a `TypeError` exception is immediately thrown when this function is called.

The `length` property of the `find` method is 1.
22.2.3.11 %TypedArray%.prototype.findIndex ( predicate, thisArg = undefined )

%TypedArray%.prototype.findIndex is a distinct function that implements the same algorithm as Array.prototype.findIndex as defined in 22.1.3.9 except that the this object’s [[ArrayLength]] internal slot is accessed in place of performing a [[Get]] of “length”. The implementation of the algorithm may be optimized with the knowledge that the this value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm.

This function is not generic. If the this value is not a object with a [[TypedArrayName]] internal slot, a TypeError exception is immediately thrown when this function is called.

The length property of the findIndex method is 1.

22.2.3.12 %TypedArray%.prototype.forEach ( callbackfn, thisArg = undefined )

%TypedArray%.prototype.forEach is a distinct function that implements the same algorithm as Array.prototype.forEach as defined in 22.1.3.10 except that the this object’s [[ArrayLength]] internal slot is accessed in place of performing a [[Get]] of “length”. The implementation of the algorithm may be optimized with the knowledge that the this value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm.

This function is not generic. If the this value is not a object with a [[TypedArrayName]] internal slot, a TypeError exception is immediately thrown when this function is called.

The length property of the forEach method is 1.

22.2.3.13 %TypedArray%.prototype.indexOf ( searchElement, fromIndex = 0 )

%TypedArray%.prototype.indexOf is a distinct function that implements the same algorithm as Array.prototype.indexOf as defined in 22.1.3.11 except that the this object’s [[ArrayLength]] internal slot is accessed in place of performing a [[Get]] of “length”. The implementation of the algorithm may be optimized with the knowledge that the this value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm.

This function is not generic. If the this value is not a object with a [[TypedArrayName]] internal slot, a TypeError exception is immediately thrown when this function is called.

The length property of the indexOf method is 1.

22.2.3.14 %TypedArray%.prototype.join ( separator )

%TypedArray%.prototype.join is a distinct function that implements the same algorithm as Array.prototype.join as defined in 22.1.3.12 except that the this object’s [[ArrayLength]] internal slot is accessed in place of performing a [[Get]] of “length”. The implementation of the algorithm may be optimized with the knowledge that the this value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm.

This function is not generic. If the this value is not a object with a [[TypedArrayName]] internal slot, a TypeError exception is immediately thrown when this function is called.
22.2.3.15 %TypedArray%.prototype.keys ( )

The initial value of the %TypedArray%.prototype.keys data property is the same built-in function object as the Array.prototype.keys method defined in 22.1.3.13.

22.2.3.16 %TypedArray%.prototype.lastIndexOf ( searchElement, fromIndex = this.length-1 )

%TypedArray%.prototype.lastIndexOf is a distinct function that implements the same algorithm as Array.prototype.lastIndexOf as defined in 22.1.3.14 except that the this object’s [[ArrayLength]] internal slot is accessed in place of performing a [[Get]] of “length”. The implementation of the algorithm may be optimized with the knowledge that the this value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm.

This function is not generic. If the this value is not a object with a [[TypedArrayName]] internal slot, a TypeError exception is immediately thrown when this function is called.

The length property of the lastIndexOf method is 1.

22.2.3.17 get %TypedArray%.prototype.length

%TypedArray%.prototype.length is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let O be the this value.
2. If Type(O) is not Object, throw a TypeError exception.
3. If O does not have a [[TypedArrayName]] internal slot, then throw a TypeError exception.
4. Assert: O has [[ViewedArrayBuffer]] and [[ArrayLength]] internal slots.
5. Let buffer be the value of O’s [[ViewedArrayBuffer]] internal slot.
6. If buffer is undefined, then throw a TypeError exception.
7. Let length be the value of O’s [[ArrayLength]] internal slot.
8. Return length.

This function is not generic. If the this value is not a object with a [[TypedArrayName]] internal slot, a TypeError exception is immediately thrown when this function is called.

22.2.3.18 %TypedArray%.prototype.map ( callbackfn, thisArg = undefined )

The interpretation and use of the arguments of %TypedArray%.prototype.map are the same as for Array.prototype.map as defined in 22.1.3.15.

When the map method is called with one or two arguments, the following steps are taken:

1. Let O be the this value.
2. If Type(O) is not Object, throw a TypeError exception.
3. If O does not have a [[TypedArrayName]] internal slot, then throw a TypeError exception.
4. Let len be the value of O’s [[ArrayLength]] internal slot.
5. If IsCallable(callbackfn) is false, throw a TypeError exception.
6. If thisArg was supplied, let T be thisArg; else let T be undefined.
7. Let C be the result of Get(O, "constructor").
8. ReturnIfAbrupt(C).
9. If IsConstructor(C) is true, then
   a. Let A be the result of calling the [[Construct]] internal method of C with argument List ([len]).
   b. ReturnIfAbrupt(A).
10. Else,
    a. Throw a TypeError exception.
11. Let k be 0.
12. Repeat, while k < len
    a. Let Pk be ToString(k).

Commented [AWB13135]: buffer needs to be an accessor both to comply with WebIDL requirements and to support the Kronos neutering strawman requirements.

Commented [AWB17136]: It would be nice to have a more explicit way to create a collection with a pre-specified number of elements.
b. Let kValue be the result of Get( O, Pk).
c. ReturnIfAbrupt( kValue ).
d. 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.
e. ReturnIfAbrupt( mappedValue ).
f. Let status be the result of Put( A, Pk, mappedValue, true ).
g. ReturnIfAbrupt( status ).
h. Increase k by 1.

13. Return A.

This function is not generic. If the this value is not a object with a [[TypedArrayName]] internal slot, a 
TypeError exception is immediately thrown when this function is called.

The length property of the map method is 1.

22.2.3.19 %TypedArray%.prototype.reduce ( callbackfn [, initialValue] )

%TypedArray%.prototype.reduce is a distinct function that implements the same algorithm as 
Array.prototype.reduce as defined in 22.1.3.18 except that the this object's [[ArrayLength]] internal slot 
is accessed in place of performing a [[Get]] of "length". The implementation of the algorithm may be 
optimized with the knowledge that the this value is an object that has a fixed length and whose integer 
indexed properties are not sparse. However, such optimization must not introduce any observable changes in 
the specified behaviour of the algorithm.

This function is not generic. If the this value is not a object with a [[TypedArrayName]] internal slot, a 
TypeError exception is immediately thrown when this function is called.

The length property of the reduce method is 1.

22.2.3.20 %TypedArray%.prototype.reduceRight ( callbackfn [, initialValue] )

%TypedArray%.prototype.reduceRight is a distinct function that implements the same algorithm as 
Array.prototype.reduceRight as defined in 22.1.3.19 except that the this object's [[ArrayLength]] internal slot 
is accessed in place of performing a [[Get]] of "length". The implementation of the algorithm may be 
optimized with the knowledge that the this value is an object that has a fixed length and whose integer 
indexed properties are not sparse. However, such optimization must not introduce any observable changes in 
the specified behaviour of the algorithm.

This function is not generic. If the this value is not a object with a [[TypedArrayName]] internal slot, a 
TypeError exception is immediately thrown when this function is called.

The length property of the reduceRight method is 1.

22.2.3.21 %TypedArray%.prototype.reverse ( )

%TypedArray%.prototype.reverse is a distinct function that implements the same algorithm as 
Array.prototype.reverse as defined in 22.1.3.20 except that the this object's [[ArrayLength]] internal slot 
is accessed in place of performing a [[Get]] of "length". The implementation of the algorithm may be 
optimized with the knowledge that the this value is an object that has a fixed length and whose integer 
indexed properties are not sparse. However, such optimization must not introduce any observable changes in 
the specified behaviour of the algorithm.

This function is not generic. If the this value is not a object with a [[TypedArrayName]] internal slot, a 
TypeError exception is immediately thrown when this function is called.
22.2.3.22 %TypedArray%.prototype.set(array, offset = 0)

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

1. Assert: array does not have a [[TypedArrayName]] internal slot. If it does, the definition in 22.2.3.23 applies.
2. Let target be the this value.
3. If Type(target) is not Object, throw a TypeError exception.
4. If target does not have a [[TypedArrayName]] internal slot, then throw a TypeError exception.
5. Assert: target has [[ViewedArrayBuffer]] internal slot.
6. Let targetBuffer be the value of target’s [[ViewedArrayBuffer]] internal slot.
7. If targetBuffer is undefined, then throw a TypeError exception.
8. Let targetLength be the value of target’s [[ArrayLength]] internal slot.
9. Let targetOffset be ToInteger (offset).
10. ReturnIfAbrupt(targetOffset).
11. If targetOffset < 0, then throw a RangeError exception.
12. Let targetName be the string value target’s [[TypedArrayName]] internal slot.
13. Let targetElementSize be the Number value of the Element Size value specified in Table 36 for targetName.
14. Let targetType be the string value of the Element Type value in Table 36 for targetName.
15. Let targetByteOffset be the value of target’s [[ByteOffset]] internal slot.
16. Let src be the result of ToObject(array).
17. ReturnIfAbrupt(src).
18. Let srcLen be the result of Get(src, “length”).
19. Let numberLength be ToNumber(srcLen).
20. Let srcLength be ToInteger(numberLength).
21. ReturnIfAbrupt(srcLength).
22. If numberLength ≠ srcLength or srcLength < 0, then throw a TypeError exception.
23. If srcLength + targetOffset > targetLength, then throw a RangeError exception.
24. Let targetByteIndex be targetOffset × targetElementSize + targetByteOffset.
25. Let k be 0.
26. Let limit be targetByteIndex + targetElementSize × min(srcLength, targetLength – targetOffset).
27. Repeat, while targetByteIndex < limit
   a. Let Pk be ToString(k).
   b. Let kValue be the result of Get(src, Pk).
   c. Let kNumber be ToInteger(kValue).
   d. ReturnIfAbrupt(kNumber).
   e. Perform SetValueInBuffer(targetBuffer, targetByteIndex, targetType, kNumber).
   f. Set k to k + 1.
   g. Set targetByteIndex to targetByteIndex + targetElementSize.

22.2.3.23 %TypedArray%.prototype.set(typedArray, offset = 0)

Set multiple values in this TypedArray, reading the values from the typedArray argument object. The optional offset value indicates the first element index in this TypedArray where values are written. If omitted, it is assumed to be 0.

1. Assert: typedArray has a [[TypedArrayName]] internal slot. If it does not, the definition in 22.2.3.22 applies.
2. Let target be the this value.
3. If Type(target) is not Object, throw a TypeError exception.
4. If target does not have a [[TypedArrayName]] internal slot, then throw a TypeError exception.
5. Assert: target has [[ViewedArrayBuffer]] internal slot.
6. Let targetBuffer be the value of target’s [[ViewedArrayBuffer]] internal slot.
7. If targetBuffer is undefined, then throw a TypeError exception.
8. Let srcBuffer be the value of typedArray’s [[ViewedArrayBuffer]] internal slot.
9. If srcBuffer is undefined, then throw a TypeError exception.
10. Let targetLength be the value of target’s [[ArrayLength]] internal slot.
11. Let targetOffset be ToInteger (offset).
12. ReturnIfAbrupt(targetOffset).
13. If targetOffset < 0, then throw a **RangeError** exception.
14. Let **targetName** be the string value of **target**’s [[TypedArrayName]] internal slot.
15. Let **targetType** be the string value of the Element Type value in Table 36 for **targetName**.
16. Let **targetElementSize** be the Number value of the Element Size value specified in Table 36 for **targetName**.
17. Let **targetByteOffset** be the value of **target**’s [[ByteOffset]] internal slot.
18. Let **srcName** be the string value of **typedArray**’s [[TypedArrayName]] internal slot.
19. Let **srcType** be the string value of the Element Type value in Table 36 for **srcName**.
20. Let **srcElementSize** be the Number value of the Element Size value specified in Table 36 for **srcName**.
21. Let **srcLength** be the value of **typedArray**’s [[ArrayLength]] internal slot.
22. Let **srcByteOffset** be the value of **typedArray**’s [[ByteOffset]] internal slot.
23. If **srcLength** + targetOffset < **targetLength**, then throw a **RangeError** exception.
24. If SameValue(srcBuffer, targetBuffer) is true, then
   a. Let **srcBuffer** be the result of calling CloneArrayBuffer(srcBuffer, srcByteOffset).
   b. Let **srcByteIndex** be 0.
25. Else, let **srcByteIndex** be srcByteOffset.
26. Let **targetByteIndex** be targetOffset × targetElementSize + targetByteOffset.
27. Let **limit** be targetByteIndex + targetElementSize × min(srcLength, **targetLength** – targetOffset).
28. Repeat, while targetByteIndex < **limit**
   a. Let **value** be the result of GetValueFromBuffer(srcBuffer, srcByteIndex, srcType).
   b. Let **status** be the result of SetValueInBuffer(targetBuffer, targetByteIndex, targetType, **value**).
   c. Set srcByteIndex to srcByteIndex + srcElementSize.
   d. Set targetByteIndex to targetByteIndex + targetElementSize.
29. Return **undefined**.

**22.2.3.24 %TypedArray%.prototype.slice ( start, end )**

The interpretation and use of the arguments of %TypedArray%.prototype.slice are the same as for Array.prototype.slice as defined in 22.1.3.22. The following steps are taken:

1. Let **O** be the **this** value.
2. If **Type**(O) is not Object, throw a **TypeError** exception.
3. If **O** does not have a [[TypedArrayName]] internal slot, then throw a **TypeError** exception.
4. Let **len** be the value of **O**’s [[ArrayLength]] internal slot.
5. Let **relativeStart** be ToInteger(start).
6. ReturnIfAbrupt(relativeStart).
7. If **relativeStart** is negative, let **k** be max((len + relativeStart),0); else let **k** be min(relativeStart, **len**).
8. If **end** is **undefined**, let **relativeEnd** be **len**; else let **relativeEnd** be ToInteger(end).
9. ReturnIfAbrupt(relativeEnd).
10. If **relativeEnd** is negative, let **final** be max((len + relativeEnd),0); else let **final** be min(relativeEnd, **len**).
11. Let **count** be max(final – **k**, 0).
12. Let **C** be the result of Get(O, "constructor").
13. ReturnIfAbrupt(C).
14. If IsConstructor(C) is true, then
   a. Let **A** be the result of calling the [[Construct]] internal method of **C** with argument (count).
   b. ReturnIfAbrupt(A).
15. Else, a. Throw a **TypeError** exception.
16. Let **n** be 0.
17. Repeat, while **k** < **final**
   a. Let **Pk** be ToString(**k**).
   b. Let **kValue** be the result of Get(O, **Pk**).
   c. ReturnIfAbrupt(**kValue**).
   d. Let **status** be the result of Put(A, ToString(n), **kValue**, true).
   e. ReturnIfAbrupt(status).
   f. Increase **k** by 1.
g. Increase \( n \) by 1.

18. Return \( A \).

This function is not generic. If the \texttt{this} value is not a object with a [[TypedArrayName]] internal slot, a \texttt{TypeError} exception is immediately thrown when this function is called.

The \texttt{length} property of the \texttt{slice} method is 2.

22.2.3.25 \%TypedArray\%\_prototype\_some ( callbackfn, thisArg = undefined )

\%TypedArray\%\_prototype\_some is a distinct function that implements the same algorithm as \texttt{Array.prototype.some} as defined in 22.1.3.23 except that the \texttt{this} object's [[ArrayLength]] internal slot is accessed in place of performing a [[Get]] of "\texttt{length}". The implementation of the algorithm may be optimized with the knowledge that the \texttt{this} value is an object that has a fixed length whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm.

This function is not generic. If the \texttt{this} value is not a object with a [[TypedArrayName]] internal slot, a \texttt{TypeError} exception is immediately thrown when this function is called.

The \texttt{length} property of the \texttt{some} method is 1.

22.2.3.26 \%TypedArray\%\_prototype\_sort ( comparefn )

\%TypedArray\%\_prototype\_sort is a distinct function that, except as described below, implements the same requirements as those of \texttt{Array.prototype.sort} as defined in 22.1.3.24. The implementation of the \%TypedArray\%\_prototype\_sort specification may be optimized with the knowledge that the \texttt{this} value is an object that has a fixed length whose integer indexed properties are not sparse. The only internal methods of the \texttt{this} object that the algorithm may call are [[Get]] and [[Set]].

This function is not generic. If the \texttt{this} value is not a object with a [[TypedArrayName]] internal slot, a \texttt{TypeError} exception is immediately thrown when it is called.

Upon entry, the following steps are performed to initialize evaluation of the \texttt{sort} function. These steps are used instead of the entry steps in 22.1.3.24:

1. Let \( obj \) be the \texttt{this} value as the argument.
2. If \( obj \) does not have a [[TypedArrayName]] internal slot, then throw a \texttt{TypeError} exception.
3. Let \( len \) be the value of \( obj \)'s [[ArrayLength]] internal slot.

The following version of SortCompare is used by %TypedArray%.prototype.sort. It performs a numeric comparison rather than the string comparison used in 22.1.3.24.

The Typed Array SortCompare abstract operation is called with two arguments \( j \) and \( k \), the following steps are taken:

1. Let \( j\text{String} \) be To\text{String}(\( j \)).
2. Let \( k\text{String} \) be To\text{String}(\( k \)).
3. Let \( x \) be the result of Get(\( obj \), \( j\text{String} \)).
4. ReturnIfAbrupt(\( x \)).
5. Let \( y \) be the result of Get(\( obj \), \( k\text{String} \)).
6. ReturnIfAbrupt(\( y \)).
7. Assert: Both Type(\( x \)) and Type(\( y \)) is Number.
8. If \( x \) and \( y \) are both NaN, return +0.
9. If \( x \) is NaN, return 1.
10. If \( y \) is NaN, return -1.
11. If the argument \( comparefn \) is not undefined, then
   a. If IsCallable(\( comparefn \)) is \text{false}, throw a \texttt{TypeError} exception.
b. Return the result of calling the [[Call]] internal method of `comparefn` passing `undefined` as thisArgument and with a List containing the values of `x` and `y` as the argumentsList.

12. If `x < y`, return `-1`.
13. If `x > y`, return `1`.
14. Return `+0`.

NOTE 1 Because NaN always compares greater than any other value, NaN property values always sort to the end of the result.

22.2.3.27 `%TypedArray%.prototype.subarray(begin = 0, end = this.length )`

Returns a new `TypedArray` object whose element types is the same as this `TypedArray` and whose ArrayBuffer is the same as the ArrayBuffer of 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. Let `O` be the this value.
2. If `Type(O)` is not Object, throw a `TypeError` exception.
3. If `O` does not have a [[TypedArrayName]] internal slot, then throw a `TypeError` exception.
4. Assert: `O` has [[ViewedArrayBuffer]] internal slot.
5. Let `buffer` be the value of `O`'s [[ViewedArrayBuffer]] internal slot.
6. If `buffer` is `undefined`, then throw a `TypeError` exception.
7. Let `srcLength` be the value of `O`'s [[ArrayLength]] internal slot.
8. Let `beginInt` be `ToInteger(begin)`
9. ReturnIfAbrupt(`beginInt`).
10. If `beginInt < 0`, then let `beginInt` be `srcLength + beginInt`.
11. Let `beginIndex` be `min(srcLength, max(0, beginInt))`.
12. If `end` is `undefined`, then let `end` be `srcLength`.
13. Let `endInt` be `ToInteger(end)`.
14. ReturnIfAbrupt(`endInt`).
15. If `endInt < 0`, then let `endInt` be `srcLength + endInt`.
16. Let `endIndex` be `max(0, min(srcLength, endInt))`.
17. If `endIndex < beginIndex`, then let `endIndex` be `beginIndex`.
18. Let `newLength` be `endIndex - beginIndex`.
19. Let `constructorName` be the string value `O`'s [[TypedArrayName]] internal slot.
20. Let `elementType` be the string value of the Element Type value in Table 36 for `constructorName`.
21. Let `elementSize` be the Number value of the Element Size value specified in Table 36 for `constructorName`.
22. Let `srcByteOffset` be the value of `O`'s [[ByteOffset]] internal slot.
23. Let `beginByteOffset` be `srcByteOffset + beginIndex * elementSize`.
24. Let `constructor` be the result of `Get(O, "constructor")`.
25. ReturnIfAbrupt(`constructor`).
26. If `IsConstructor(constructor)` is `false`, then throw a `TypeError` exception.
27. Let `argumentsList` be a List consisting of `buffer`, `beginByteOffset`, and `newLength`.
28. Return the result of calling the [[Construct]] internal method of `constructor` with `argumentsList` as the argument.

22.2.3.28 `%TypedArray%.prototype.toLocaleString ( )`

The initial value of the `%TypedArray%.prototype.toLocaleString` data property is the same built-in function object as the `Array.prototype.toLocaleString` method defined in 22.1.3.26.

22.2.3.29 `%TypedArray%.prototype.toString ( )`

The initial value of the `%TypedArray%.prototype.toString` data property is the same built-in function object as the `Array.prototype.toString` method defined in 22.1.3.27.
22.2.3.30 %TypedArray%.prototype.values ( )

The initial value of the %TypedArray%.prototype.values data property is the same built-in function object as the Array.prototype.values method defined in 22.1.3.29.

22.2.3.31 %TypedArray%.prototype[@@iterator] ( )

The initial value of the @@iterator property is the same function object as the initial value of the %TypedArray%.prototype.values property.

22.2.3.32 get %TypedArray%.prototype[ @@toStringTag ]

%TypedArray%.prototype[@@toStringTag] is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let O be the this value.
2. If Type(O) is not Object, throw a TypeError exception.
3. If O does not have a [[TypedArrayName]] internal slot throw a TypeError exception.
4. Let name be the value of O’s [[TypedArrayName]] internal slot.
5. Assert: name is a String value.
6. Return name.

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

The initial value of the name property of this function is "[Symbol.toStringTag]."

22.2.4 The TypedArray Constructors

Each of these TypedArray constructor objects has the structure described below, differing only in the name used as the constructor name instead of TypedArray, in Table 36.

When a TypedArray constructor is called as a function rather than as a constructor, it initialises a new TypedArray object. The this value passed in the call must be an Object with a [[TypedArrayName]] internal slot and a [[ViewedArrayBuffer]] internal slot whose value is undefined. The constructor function initialises the this value using the argument values.

The TypedArray constructors are designed to be subclassable. They may be used as the value of an extends clause of a class declaration. Subclass constructors that intended to inherit the specified TypedArray behaviour must include a super call to the TypedArray constructor to initialise subclass instances.

22.2.4.1 TypedArray( ... argumentsList)

A TypedArray constructor with a list of arguments argumentsList performs the following steps:

1. Let O be the this value.
2. If Type(O) is not Object, then throw a TypeError exception.
3. If O does not have a [[TypedArrayName]] internal slot, then throw a TypeError exception.
4. If the value of O’s [[TypedArrayName]] internal slot is not undefined, then throw a TypeError exception.
5. Set O’s [[TypedArrayName]] internal slot to the String value from the constructor name column in the row of Table 36 corresponding to this constructor.
6. Let F be the TypedArray function object that was called.
7. Let realmF be F’s [[Realm]] internal slot.
8. Let super be realmF’s %TypedArray% intrinsic object.
9. Let argumentsList be the argumentsList argument of the [[Call]] internal method that invoked F.
10. Return the result of calling the [[Call]] internal method of super with O and argumentsList as arguments.

22.2.4.2 new TypedArray( ... argumentsList)

A TypedArray constructor called as part of a new expression performs the following steps:
1. Let \( F \) be the `TypedArray` function object on which the `new` operator was applied.
2. Let `argumentsList` be the `argumentsList` argument of the `[[Construct]]` internal method that was invoked by the `new` operator.
3. Return the result of `OrdinaryConstruct(F, argumentsList)`.

22.2.5 Properties of the `TypedArray` Constructors

The value of the `[[Prototype]]` internal slot of each `TypedArray` constructor is the `%TypedArray% intrinsic object (22.2.1).

Each `TypedArray` constructor has a `name` property whose value is the String value of the constructor name specified for it in Table 36.

Besides a `length` property (whose value is 3), each `TypedArray` constructor has the following properties:

22.2.5.1 `TypedArray.BYTES_PER_ELEMENT`

The value of `TypedArray.BYTES_PER_ELEMENT` is the Number value of the Element Size value specified in Table 36 for `TypedArray`.

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

22.2.5.2 `TypedArray.prototype`

The initial value of `TypedArray.prototype` is the corresponding `TypedArray` prototype object (22.2.6).

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

22.2.6 Properties of `TypedArray` Prototype Objects

The value of the `[[Prototype]]` internal slot of a `TypedArray` prototype object is the standard built-in `%TypedArrayPrototype% object (22.2.3). A `TypedArray` prototype object is an ordinary object. It does not have a `[[ViewedArrayBuffer]]` or any other of the internal slots that are specific to `TypedArray` instance objects.

22.2.6.1 `TypedArray.prototype.BYTES_PER_ELEMENT`

The value of `TypedArray.prototype.BYTES_PER_ELEMENT` is the Number value of the Element Size value specified in Table 36 for `TypedArray`.

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

22.2.6.2 `TypedArray.prototype.constructor`

The initial value of a `TypedArray.prototype.constructor` is the corresponding standard built-in `TypedArray` constructor.

22.2.7 Properties of `TypedArray` Instances

`TypedArray` instances are Integer Indexed exotic objects. Each `TypedArray` instances inherits properties from the corresponding `TypedArray` prototype object. Each `TypedArray` instances have the following internal slots: `[[TypedArrayName]], [[ViewedArrayBuffer]], [[ByteLength]], [[ByteOffset]], and [[ArrayLength]]`. 
23 Keyed Collection

23.1 Map Objects

Map objects are collections of key/value pairs where both the keys and values may be arbitrary ECMAScript language values. A distinct key value may only occur in one key/value pair within the Map’s collection. Distinct key values as discriminated using the a comparison algorithm that is selected when the Map is created.

A Map object can iterate its elements in insertion order. Map object must be implemented using either 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.

23.1.1 The Map Constructor

The Map constructor is the %Map% intrinsic object and the initial value of the Map property of the global object. When Map is called as a function rather than as a constructor, it initialises its this value with the internal state necessary to support the Map.prototype internal methods.

The Map constructor is designed to be subclassable. It may be used as the value in an extends clause of a class definition. Subclass constructors that intend to inherit the specified Map behaviour must include a super call to Map.

23.1.1.1 Map (iterable = undefined , comparator = undefined )

When the Map function is called with optional arguments iterable and comparator the following steps are taken:

1. Let map be the this value.
2. If Type(map) is not Object then, throw a TypeError exception.
3. If map does not have a [[MapData]] internal slot, then throw a TypeError exception.
4. If map’s [[MapData]] internal slot is not undefined, then throw a TypeError exception.
5. If iterable is not present, let iterable be undefined.
6. If iterable is either undefined or null, then let iter be undefined.
7. Else,
   a. Let iter be the result of GetIterator(iterable).
   b. ReturnIfAbrupt(iter).
   c. Let adder be the result of Get(map, "set").
   d. ReturnIfAbrupt(adder).
   e. If IsCallable(adder) is false, throw a TypeError Exception.
8. If comparator is not undefined, then
   a. If comparator is not "is", then throw a RangeError Exception.
10. Assert: map has not been reentrantly initialized.
11. Set map’s [[MapData]] internal slot to a new empty List.
12. Set map’s [[MapPrototype]] internal slot to comparator.
13. If iter is undefined, then return map.
14. Repeat
   a. Let next be the result of IteratorStep(iter).
   b. ReturnIfAbrupt(next).
   c. If next is false, then return NormalCompletion(map).
   d. Let nextItem be IteratorValue(next).
   e. ReturnIfAbrupt(nextItem).
   f. If Type(nextItem) is not Object, then throw a TypeError exception.
   g. Let k be the result of Get(nextItem, "0").
   h. ReturnIfAbrupt(k).
   i. Let v be the result of Get(nextItem, "1").
   j. ReturnIfAbrupt(v).
k. Let status be the result of calling the [[Call]] internal method of adder with `map` as `thisArgument` and a List whose elements are `k` and `v` as `argumentsList`.

l. ReturnIfAbrupt(status).

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 a two element array-like object 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.

### 23.1.1.2 new Map ( ... argumentsList )

When `Map` is called as part of a `new` expression it is a constructor: it initialises a newly created object.

`Map` called as part of a new expression with argument list `argumentsList` performs the following steps:

1. Let `F` be the Map function object on which the `new` operator was applied.
2. Let `argumentsList` be the `argumentsList` argument of the `[[Construct]]` internal method that was invoked by the `new` operator.
3. Return the result of `OrdinaryConstruct(F, argumentsList)`.

If `Map` is implemented as an ECMAScript function object, its `[[Construct]]` internal method will perform the above steps.

#### 23.1.2 Properties of the `Map` Constructor

The value of the `[[Prototype]]` internal slot of the `Map` constructor is the Function prototype object (19.2.3).

Besides the `length` property (whose value is 0), the `Map` constructor has the following properties:

##### 23.1.2.1 Map.prototype

The initial value of `Map.prototype` is the `Map` prototype object (23.1.3).

This property has the attributes ( [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false ).

##### 23.1.2.2 Map @@create ()

The @@create method of a Map function object `F` performs the following steps:

1. Let `F` be the this value.
2. Let `obj` be the result of calling `OrdinaryCreateFromConstructor(F, "%MapPrototype%", ( [[MapData]], [[MapComparator]]) )`.
3. Return `obj`.

The value of the `name` property of this function is "`[Symbol.create]`".

This property has the attributes ( [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true ).

#### 23.1.3 Properties of the Map Prototype Object

The value of the `[[Prototype]]` internal slot of the Map prototype object is the standard built-in `Object` prototype object (19.1.3). The Map prototype object is an ordinary object. It does not have a `[[MapData]]` or a `[[MapComparator]]` internal slot.
23.1.3.1 Map.prototype.clear ()

The following steps are taken:

1. Let M be the this value.
2. If Type(M) is not Object, then throw a TypeError exception.
3. If M does not have a [[MapData]] internal slot throw a TypeError exception.
4. If M’s [[MapData]] internal slot is undefined, then throw a TypeError exception.
5. Let entries be the List that is the value of M’s [[MapData]] internal slot.
6. Repeat for each Record {{[[key]], [[value]]}} p that is an element of entries,
   a. Set p.{{[[key]]}} to empty.
   b. Set p.{{[[value]]}} to empty.
7. Return undefined.

NOTE The existing [[MapData]] List is preserved because there may be existing MapIterator objects that are suspended midway through iterating over that List.

23.1.3.2 Map.prototype.constructor

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

23.1.3.3 Map.prototype.delete ( key )

The following steps are taken:

1. Let M be the this value.
2. If Type(M) is not Object, then throw a TypeError exception.
3. If M does not have a [[MapData]] internal slot throw a TypeError exception.
4. If M’s [[MapData]] internal slot is undefined, then throw a TypeError exception.
5. If M’s [[MapComparator]] internal slot is undefined, then let same be the abstract operation SameValueZero.
6. Else, let same be the abstract operation SameValue.
7. Let entries be the List that is the value of M’s [[MapData]] internal slot.
8. Repeat for each Record {{[[key]], [[value]]}} p that is an element of entries,
   a. If same(p.{{[[key]]}}, key), then
      i. Set p.{{[[key]]}} to empty.
      ii. Set p.{{[[value]]}} to empty.
     iii. Return true.
9. Return false.

NOTE The value empty is used as a specification device to indicate that an entry has been deleted. Actual implementations may take other actions such as physically removing the entry from internal data structures.

23.1.3.4 Map.prototype.entries ()

The following steps are taken:

1. Let M be the this value.
2. If Type(M) is not Object, then throw a TypeError exception.
3. Return the result of calling the CreateMapIterator abstract operation with arguments M and "key+value".

23.1.3.5 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.
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.

When the forEach method is called with one or two arguments, the following steps are taken:

1. Let M be the this value.
2. If Type(M) is not Object, then throw a TypeError exception.
3. If M does not have a [[MapData]] internal slot throw a TypeError exception.
4. If M's [[MapData]] internal slot is undefined, then throw a TypeError exception.
5. If IsCallable(callbackfn) is false, throw a TypeError exception.
6. If thisArg was supplied, let T be thisArg; else let T be undefined.
7. Let entries be the List that is the value of M's [[MapData]] internal slot.
8. 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).
9. Return undefined.

The length property of the forEach method is 1.

23.1.3.6 Map.prototype.get (key)
The following steps are taken:

1. Let M be the this value.
2. If Type(M) is not Object, then throw a TypeError exception.
3. If M does not have a [[MapData]] internal slot throw a TypeError exception.
4. If M's [[MapData]] internal slot is undefined, then throw a TypeError exception.
5. Let entries be the List that is the value of M's [[MapData]] internal slot.
6. If M's [[MapComparator]] internal slot is undefined, then let same be the abstract operation
   SameValueZero.
7. Else, let same be the abstract operation SameValue.
8. Repeat for each Record [[key]], [[value]] p that is an element of entries,
   a. If same(p.[[key]], key), then return p.[[value]].
9. Return undefined.

23.1.3.7 Map.prototype.has (key)
The following steps are taken:

1. Let M be the this value.
2. If Type(M) is not Object, then throw a TypeError exception.
3. If M does not have a [[MapData]] internal slot throw a TypeError exception.
4. If M's [[MapData]] internal slot is undefined, then throw a TypeError exception.
5. Let entries be the List that is the value of M's [[MapData]] internal slot.
6. If M's [[MapComparator]] internal slot is undefined, then let same be the abstract operation
   SameValueZero.
7. Else, let same be the abstract operation SameValue.
8. Repeat for each Record [[key]], [[value]] p that is an element of entries,
   a. If same(p.[[key]], key), then return true.
9. Return false.
23.1.3.8 Map.prototype.keys ()

The following steps are taken:

1. Let M be the this value.
2. Return the result of calling the CreateMapIterator abstract operation with arguments M and "key".

23.1.3.9 Map.prototype.set ( key , value )

The following steps are taken:

1. Let M be the this value.
2. If Type(M) is not Object, then throw a TypeError exception.
3. If M does not have a [[MapData]] internal slot throw a TypeError exception.
4. If M's [[MapData]] internal slot is undefined, then throw a TypeError exception.
5. Let entries be the List that is the value of M's [[MapData]] internal slot.
6. If M's [[MapComparator]] internal slot is undefined, then let same be the abstract operation SameValueZero.
7. Else, let same be the abstract operation SameValue.
8. Repeat for each Record {
    [[key]], [[value]]
} p that is an element of entries,
   a. If same(p.[[key]], key), then
      i. Set p.[[value]] to value.
     ii. Return M.
9. Let p be the Record {
    [[key]], [[value]]: value
}. Append p as the last element of entries.
10. Return M.

23.1.3.10 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 this value.
2. If Type(M) is not Object, then throw a TypeError exception.
3. If M does not have a [[MapData]] internal slot throw a TypeError exception.
4. If M's [[MapData]] internal slot is undefined, then throw a TypeError exception.
5. Let entries be the List that is the value of M's [[MapData]] internal slot.
6. Let count be 0.
7. 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.
8. Return count.

23.1.3.11 Map.prototype.values ( )

The following steps are taken:

1. Let M be the this value.
2. Return the result of calling the CreateMapIterator abstract operation with arguments M and "value".

23.1.3.12 Map.prototype [ @@iterator ] ( )

The initial value of the @@iterator property is the same function object as the initial value of the entries property.

23.1.3.13 Map.prototype [ @@toStringTag ]

The initial value of the @@toStringTag property is the string value "Map".
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

23.1.4 Properties of Map Instances

Map instances are ordinary objects that inherit properties from the Map prototype. Map instances also have a [[MapData]] internal slot and a [[MapComparator]] internal slot.

23.1.5 Map Iterator Objects

A Map Iterator is an object that represents 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.

23.1.5.1 CreateMapIterator Abstract Operation

Several methods of Map objects return iterator objects. The abstract operation CreateMapIterator with arguments `map` and `kind` is used to create such iterator objects. It performs the following steps:

1. If `Type(map)` is not Object, throw a TypeError exception.
2. If `map` does not have a [[MapData]] internal slot throw a TypeError exception.
3. Let `entries` be the List that is the value of `map`'s [[MapData]] internal slot.
4. Let `iterator` be the result of ObjectCreate(%MapIteratorPrototype%, ([[Map]], [[MapNextIndex]], [[MapIterationKind]])).
5. Set `iterator`'s [[Map]] internal slot to `map`.
6. Set `iterator`'s [[MapNextIndex]] internal slot to 0.
7. Set `iterator`'s [[MapIterationKind]] internal slot to `kind`.
8. Return `iterator`.

23.1.5.2 The %MapIteratorPrototype% Object

All Map Iterator Objects inherit properties from the %MapIteratorPrototype% intrinsic object. The %MapIteratorPrototype% intrinsic object is an ordinary object and its [[Prototype]] internal slot is the %ObjectPrototype% intrinsic object. In addition, %MapIteratorPrototype% has the following properties:

23.1.5.2.1 %MapIteratorPrototype%.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 slots of a Map Iterator Instance (23.1.5.3), throw a TypeError exception.
4. Let `m` be the value of the [[Map]] internal slot of `O`.
5. Let `index` be the value of the [[MapNextIndex]] internal slot of `O`.
6. Let `itemKind` be the value of the [[MapIterationKind]] internal slot of `O`.
7. If `m` is `undefined`, then return CreateIterResultObject(undefined, true).
8. Assert: `m` has a [[MapData]] internal slot and `m` has been initialised so the value of [[MapData]] is not `undefined`.
9. Let `entries` be the List that is the value of the [[MapData]] internal slot of `m`.
10. Repeat while `index` is less than the total number of elements of `entries`. The number of elements must be reetermined each time this method is evaluated.
    a. Let `e` be the Record {[[key]], [[value]]} that is the value of `entries[index]`.
    b. Set `index` to `index+1`.
    c. Set the [[MapNextIndex]] internal slot 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]]`. 

© Ecma International 2013
iii. Else,
   1. Assert: `itemKind` is "key+value".
   2. Let `result` be the result of performing `ArrayCreate(2)`.
   3. Assert: `result` is a new, well-formed Array object so the following operations will never fail.
   4. Call `CreateDataProperty(result, "0", e.[[key]])`.
   5. Call `CreateDataProperty(result, "1", e.[[value]])`.

   iv. Return `CreateIteratorResultObject(result, false)`.

12. Set the `[[Map]]` internal slot of `O` to `undefined`.
13. Return `CreateIteratorResultObject(undefined, true)`.

23.1.5.2.2 `%MapIteratorPrototype% [ @@iterator ]` ( )

The following steps are taken:

1. Return the `this` value.

The value of the `name` property of this function is "[[Symbol.iterator]]".

23.1.5.2.3 `%MapIteratorPrototype% [ @@toStringTag ]`

The initial value of the `@@toStringTag` property is the string value "`Map Iterator`".

23.1.5.3 Properties of Map Iterator Instances

Map iterator instances are ordinary objects that inherit properties from the `%MapIteratorPrototype%` intrinsic object. Map iterator instances are initially created with the internal slots described in Table 37.

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[Map]]</code></td>
<td>The Map object that is being iterated.</td>
</tr>
<tr>
<td><code>[[MapNextIndex]]</code></td>
<td>The integer index of the next Map data element to be examined by this iteration.</td>
</tr>
<tr>
<td><code>[[MapIterationKind]]</code></td>
<td>A string value that identifies what is to be returned for each element of the iteration. The possible values are: &quot;key&quot;, &quot;value&quot;, &quot;key+value&quot;.</td>
</tr>
</tbody>
</table>

23.2 Set Objects

Set objects are collections of ECMAScript language values. A distinct value may only occur once as an element of a Set’s collection. Distinct values are discriminated using a comparison algorithm that is selected when the Set is created.

A Set object can iterate its elements in insertion order. Set objects must be implemented using either 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.
23.2.1 The Set Constructor

The Set constructor is the %Set% intrinsic object and the initial value of the `Set` property of the global object. When `Set` is called as a function rather than as a constructor, it initialises its `this` value with the internal state necessary to support the `Set.prototype` internal methods.

The `Set` constructor is designed to be subclassable. It may be used as the value in an `extends` clause of a class definition. Subclass constructors that intend to inherit the specified `Set` behaviour must include a `super` call to `Set`.

23.2.1.1 `Set(iterable = undefined, comparator = undefined)`

When the `Set` function is called with optional arguments `iterable` and `comparator` the following steps are taken:

1. Let `set` be the `this` value.
2. If `Type(set)` is not Object then, throw a `TypeError` exception.
3. If `set` does not have a `[[SetData]]` internal slot, then throw a `TypeError` exception.
4. If `set`'s `[[SetData]]` internal slot is not undefined, then throw a `TypeError` exception.
5. If `iterable` is either `undefined` or `null`, then let `it`e be undefined.
6. Else,
   a. Let `it`e be the result of GetIterator(`iterable`).
   b. ReturnIfAbrupt(`it`e).
   c. Let `adder` be the result of Get(`set`, "add").
   d. ReturnIfAbrupt(`adder`).
   e. If `IsCallable(adder)` is false, throw a `TypeError` Exception.
7. If `comparator` is not present, let `comparator` be undefined.
8. If `comparator` is not `undefined`, then
   a. If `comparator` is not "is", then throw a `RangeError` Exception.
9. Set `set`'s `[[SetComparator]]` internal slot to `comparator`.
10. If `iter` is `undefined`, then return `set`.
11. Repeat
    a. Let `next` be the result of IteratorStep(`iter`).
    b. ReturnIfAbrupt(`next`).
    c. If `next` is false, then return `set`.
    d. Let `nextValue` be IteratorValue(`next`).
    e. ReturnIfAbrupt(`nextValue`).
    f. Let `status` be the result of calling the `[[Call]]` internal method of `adder` with `set` as `thisArgument` and a List whose sole element is `nextValue` as `argumentsList`.
    g. ReturnIfAbrupt(`status`).

23.2.1.2 `new Set(... argumentsList)`

When `Set` is called as part of a `new` expression it is a constructor: it initialises a newly created object.

`Set` called as part of a `new` expression with argument list `argumentsList` performs the following steps:

1. Let `F` be the Set function object on which the `new` operator was applied.
2. Let `argumentsList` be the `argumentsList` argument of the `[[Construct]]` internal method that was invoked by the `new` operator.
3. Return the result of OrdinaryConstruct(`F`, `argumentsList`).

If `Set` is implemented as an ECMAScript function object, its `[[Construct]]` internal method will perform the above steps.

Commented [AWB12140]: Note that using a method call for inserting pairs during initialisation allows subclasses to be more expressive.
23.2.2 Properties of the Set Constructor

The value of the [[Prototype]] internal slot of the Set constructor is the Function prototype object (19.2.3). Besides the length property (whose value is 0), the Set constructor has the following properties:

23.2.2.1 Set.prototype

The initial value of Set.prototype is the intrinsic %SetPrototype% object (23.2.3).

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

23.2.2.2 Set @@create ()

The @@create method of a Set function object F performs the following steps:

1. Let F be the this value.
2. Let obj be the result of calling OrdinaryCreateFromConstructor(F, "%SetPrototype%", [[SetData]], [[SetComparator]]).
3. Return obj.

The value of the name property of this function is "Symbol.create". This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

23.2.3 Properties of the Set Prototype Object

The value of the [[Prototype]] internal slot of the Set prototype object is the standard built-in Object prototype object (19.1.3). The Set prototype object is an ordinary object. It does not have a [[SetData]] or a [[SetComparator]] internal slot.

23.2.3.1 Set.prototype.add (value)

The following steps are taken:

1. Let S be the this value.
2. If Type(S) is not Object, then throw a TypeError exception.
3. If S does not have a [[SetData]] internal slot throw a TypeError exception.
4. If S's [[SetData]] internal slot is undefined, then throw a TypeError exception.
5. Let entries be the List that is the value of S's [[SetData]] internal slot.
6. If S's [[SetComparator]] internal slot is undefined, then let same be the abstract operation SameValueZero.
7. Else, let same be the abstract operation SameValue.
8. Repeat for each e that is an element of entries, in original insertion order
   a. If e is not empty and same(e, value) is true, then
      i. Return S.
9. Append value as the last element of entries.
10. Return S.

23.2.3.2 Set.prototype.clear ()

The following steps are taken:

1. Let S be this value.
2. If Type(S) is not Object, then throw a TypeError exception.
3. If S does not have a [[SetData]] internal slot throw a TypeError exception.
4. If S's [[SetData]] internal slot is undefined, then throw a TypeError exception.
5. Let entries be the List that is the value of S's [[SetData]] internal slot.
6. Repeat for each e that is an element of entries,
      a. Replace the element of entries whose value is e with an element whose value is empty.
7. Return undefined.

23.2.3.3 Set.prototype.constructor

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

23.2.3.4 Set.prototype.delete(value)

The following steps are taken:

1. Let \( S \) be the this value.
2. If Type(\( S \)) is not Object, then throw a TypeError exception.
3. If \( S \) does not have a [[SetData]] internal slot throw a TypeError exception.
4. If \( S \)’s [[SetData]] internal slot is undefined, then throw a TypeError exception.
5. Let entries be the List that is the value of \( S \)’s [[SetData]] internal slot.
6. If \( S \)’s [[SetComparator]] internal slot is undefined, then let same be the abstract operation SameValueZero.
7. Else, let same be the abstract operation SameValue.
8. Repeat for each \( e \) that is an element of entries, in original insertion order
   a. If \( e \) is not empty and same(\( e \), value) is true, then
      i. Replace the element of entries whose value is \( e \) with an element whose value is empty.
      ii. Return true.
9. Return false.

NOTE The value empty is used as a specification device to indicate that an entry has been deleted. Actual implementations may take other actions such as physically removing the entry from internal data structures.

23.2.3.5 Set.prototype.entries()

The following steps are taken:

1. Let \( S \) be the this value.
2. Return the result of calling the CreateSetIterator abstract operation with arguments \( S \) and "key+value".

NOTE For iteration purposes, a Set appears similar to a Map where each entry has the same value for its key and value.

23.2.3.6 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 1 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 2 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 3 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 this value.
2. If Type(S) is not Object, then throw a TypeError exception.
3. If S does not have a [[SetData]] internal slot throw a TypeError exception.
4. If S's [[SetData]] internal slot is undefined, then throw a TypeError exception.
5. If IsCallable(callbackfn) is false, throw a TypeError exception.
6. If thisArg was supplied, let T be thisArg; else let T be undefined.
7. Let entries be the List that is the value of S's [[SetData]] internal slot.
8. 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, e, and S as argumentsList.
      ii. ReturnIfAbrupt(funcResult).
9. Return undefined.

The length property of the forEach method is 1.

23.2.3.7 Set.prototype.has ( value )

The following steps are taken:

1. Let S be the this value.
2. If Type(S) is not Object, then throw a TypeError exception.
3. If S does not have a [[SetData]] internal slot throw a TypeError exception.
4. If S's [[SetData]] internal slot is undefined, then throw a TypeError exception.
5. Let entries be the List that is the value of S's [[SetData]] internal slot.
6. If S's [[SetComparator]] internal slot is undefined, then let same be the abstract operation SameValueZero.
7. Else, let same be the abstract operation SameValue.
8. Repeat for each e that is an element of entries,
   a. If e is not empty and same(e, value), then return true.
9. Return false.

23.2.3.8 Set.prototype.keys ()

The initial value of the keys property is the same function object as the initial value of the values property.

NOTE For iteration purposes, a Set appears similar to a Map where each entry has the same value for its key and value.

23.2.3.9 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:

1. Let S be the this value.
2. If Type(S) is not Object, then throw a TypeError exception.
3. If S does not have a [[SetData]] internal slot throw a TypeError exception.
4. If S's [[SetData]] internal slot is undefined, then throw a TypeError exception.
5. Let entries be the List that is the value of S's [[SetData]] internal slot.
6. Let count be 0.
7. For each e that is an element of entries
   a. If e is not empty then
      i. Set count to count+1.
8. Return count.
23.2.3.10 Set.prototype.values ( )

The following steps are taken:
1. Let S be the this value.
2. Return the result of calling the CreateSetIterator abstract operation with argument S and "value".

23.2.3.11 Set.prototype[@@iterator] ( )

The initial value of the @@iterator property is the same function object as the initial value of the values property.

23.2.3.12 Set.prototype[@@toStringTag]

The initial value of the @@toStringTag property is the string value "Set".

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

23.2.4 Properties of Set Instances

Set instances are ordinary objects that inherit properties from the Set prototype. After initialisation by the Set constructor, Set instances also have a [[SetData]] internal slot and a [[SetComparator]] internal slot.

23.2.5 Set Iterator Objects

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.

23.2.5.1 CreateSetIterator Abstract Operation

Several methods of Set objects return iterator objects. The abstract operation CreateSetIterator with arguments set and kind is used to create such iterator objects. It performs the following steps:
1. If Type(set) is not Object, throw a TypeError exception.
2. If set does not have a [[SetData]] internal slot throw a TypeError exception.
3. If set's [[SetData]] internal slot is undefined, then throw a TypeError exception.
4. Let $entries$ be the List that is the value of set's [[SetData]] internal slot.
5. Let iterator be the result of ObjectCreate(%SetIteratorPrototype%, ([[IteratedSet]], [[SetNextIndex]], [[SetIterationKind]])).
6. Set iterator's [[IteratedSet]] internal slot to set.
7. Set iterator's [[SetNextIndex]] internal slot to 0.
8. Set iterator's [[SetIterationKind]] internal slot to kind.
9. Return iterator.

23.2.5.2 The %SetIteratorPrototype% Object

All Set Iterator Objects inherit properties from the %SetIteratorPrototype% intrinsic object. The %SetIteratorPrototype% intrinsic object is an ordinary object and its [[Prototype]] internal slot is the %ObjectPrototype% intrinsic object. In addition, %SetIteratorPrototype% has the following properties:

23.2.5.2.1 %SetIteratorPrototype%.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 slots of a Set Iterator Instance (23.2.5.3), throw a **TypeError** exception.

4. Let s be the value of the [[IteratedSet]] internal slot of O.

5. Let index be the value of the [[SetNextIndex]] internal slot of O.

6. Let itemKind be the value of the [[SetIterationKind]] internal slot of O.

7. Assert: s has a [[SetData]] internal slot and s has been initialised so the value of [[SetData]] is not **undefined**.

8. If s is **undefined**, then return CreateIterResultObject(**undefined**, true).

9. Let entries be the value of the [[SetData]] internal slot and s has been initialised so the value of [[SetData]] is not **undefined**.

10. Repeat while index is less than the total number of elements of entries. The number of elements must be re-determined each time this method is evaluated.
   a. Let e be entries[index].
   b. Set index to index+1;
   c. Set the [[SetNextIndex]] internal slot of O to index.
   d. If e is not empty, then
      i. If itemKind is "key+value" then,
      1. Let result be the result of performing ArrayCreate(2).
      2. Assert: result is a new, well-formed Array object so the following operations will never fail.
      3. Call CreateDataProperty(result, "0", e).
      4. Call CreateDataProperty(result, "1", e).
      5. Return CreateIterResultObject(result, false).
      ii. Return CreateIterResultObject(e, false).

11. Set the [[IteratedSet]] internal slot of O to **undefined**.


23.2.5.2.2 `%SetPrototype%[@@iterator]` ( )

The following steps are taken:

1. Return the **this** value.

   The value of the name property of this function is "Symbol.iterator".

23.2.5.2.3 `%SetPrototype%[@@toStringTag]`

The initial value of the @@toStringTag property is the string value "Set Iterator".

23.2.5.3 Properties of Set Iterator Instances

Set Iterator instances are ordinary objects that inherit properties from the `%SetPrototype%` intrinsic object. Set Iterator instances are initially created with the internal slots specified Table 38.

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[IteratedSet]]</td>
<td>The Set object that is being iterated.</td>
</tr>
<tr>
<td>[[SetNextIndex]]</td>
<td>The integer index of the next Set data element to be examined by this iteration.</td>
</tr>
<tr>
<td>[[SetIterationKind]]</td>
<td>A string value that identifies what is to be returned for each element of the iteration. The possible values are: &quot;key&quot;, &quot;value&quot;, &quot;key+value&quot;, &quot;key&quot; and &quot;value&quot; have the same meaning.</td>
</tr>
</tbody>
</table>
23.3 WeakMap Objects

WeakMap objects are collections of key/value pairs where the keys are 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 observe a key of a WeakMap that does not require the observer to present the observed key.

WeakMap objects must be implemented using either 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 structure 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 and WeakSets are intended to provide mechanisms for dynamically associating state with an object in a manner that does not “leak” memory resources if, in the absence of the WeakMap or WeakSet, the object otherwise became inaccessible and subject to resource reclamation by the implementation’s garbage collection mechanisms. Achieving this characteristic requires coordination between the WeakMap or WeakSet implementation and the garbage collector. The following references describe mechanism that may be useful to implementations of WeakMap and WeakSets:


23.3.1 The WeakMap Constructor

The WeakMap constructor is the %WeakMap% intrinsic object and the initial value of the WeakMap property of the global object. When WeakMap is called as a function rather than as a constructor, it initialises its this value with the internal state necessary to support the WeakMap.prototype methods.

The WeakMap constructor is designed to be subclassable. It may be used as the value in an extends clause of a class definition. Subclass constructors that intend to inherit the specified WeakMap behaviour must include a super call to WeakMap.

23.3.1.1 WeakMap (iterable = undefined)

When the WeakMap function is called with optional argument iterable the following steps are taken:

1. Let map be the this value.
2. If Type(map) is not Object then, throw a TypeError exception.
3. If iterable does not have a [[WeakMapData]] internal slot, then throw a TypeError exception.
4. If iterable’s [[WeakMapData]] internal slot is not undefined, then throw a TypeError exception.
5. If iterable is not present, let iterable be undefined.
6. If iterable is either undefined or null, then let iter be undefined.
7. Else,
a. Let iter be the result of GetIterator(iterable).
b. ReturnIfAbrupt(iter).
c. Let adder be the result of Get(map, "set").
d. ReturnIfAbrupt(adder).
e. If IsCallable(adder) is false, throw a TypeError Exception.

8. Set map’s [[WeakMapData]] internal slot to a new empty List.
9. If iter is undefined, then return map.
10. Repeat
    a. Let next be the result of IteratorStep(iter).
    b. ReturnIfAbrupt(next).
    c. If next is false, then return NormalCompletion(map).
    d. Let nextValue be IteratorValue(next).
    e. ReturnIfAbrupt(nextValue).
    f. If Type(nextValue) is not Object, then throw a TypeError exception
    g. Let k be the result of Get(nextValue, "0").
    h. Let v be the result of Get(nextValue, "1").
    i. ReturnIfAbrupt(k).
    j. ReturnIfAbrupt(v).
    k. Let status be the result of calling the [[Call]] internal method of adder with map as thisArgument and a List whose elements are k and v as argumentsList.
    l. ReturnIfAbrupt(status).

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 a two element array-like object 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.

23.3.1.2 new WeakMap ( ... argumentsList )

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

WeakMap called as part of a new expression with argument list argumentsList performs the following steps:

1. Let F be the WeakMap function object on which the new operator was applied.
2. Let argumentsList be the argumentsList argument of the [[Construct]] internal method that was invoked by the new operator.
3. Return the result of OrdinaryConstruct(F, argumentsList).

If WeakMap is implemented as an ECMAScript function object, its [[Construct]] internal method will perform the above steps.

23.3.2 Properties of the WeakMap Constructor

The value of the [[Prototype]] internal slot of the WeakMap constructor is the Function prototype object (19.2.3).

Besides the length property (whose value is 0), the WeakMap constructor has the following properties:

23.3.2.1 WeakMap.prototype

The initial value of WeakMap.prototype is the WeakMap prototype object (23.3.3).

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

23.3.2.2 WeakMap[@@create] ()

The @@create method of a WeakMap object F performs the following steps:

1. Let F be the this value.

Commented [AWB12143]: Note that using a method call for inserting pairs during initialisation provides allows subclasses to be more expressive.
2. Let \( \text{obj} \) be the result of calling `OrdinaryCreateFromConstructor(F, "WeakMapPrototype", (WeakMapData))`. 
3. Return \( \text{obj} \).

The value of the `name` property of this function is 
"[Symbol.create]".

This property has the attributes `[[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true`.

### 23.3 Properties of the WeakMap Prototype Object

The value of the `[[Prototype]]` internal slot of the WeakMap prototype object is the standard built-in Object prototype object (19.1.3). The WeakMap prototype object is an ordinary object. It does not have a `[[WeakMapData]]` internal slot.

#### 23.3.3.1 WeakMap.prototype.clear()

The following steps are taken:

1. Let \( M \) be the `this` value.
2. If `Type(M)` is not Object, then throw a `TypeError` exception.
3. If \( M \) does not have a `[[WeakMapData]]` internal slot throw a `TypeError` exception.
4. If \( M \)'s `[[WeakMapData]]` internal slot is `undefined`, then throw a `TypeError` exception.
5. Set the value of \( M \)'s `[[WeakMapData]]` internal slot to a new empty List.
6. Return `undefined`.

#### 23.3.3.2 WeakMap.prototype.constructor

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

#### 23.3.3.3 WeakMap.prototype.delete(key)

The following steps are taken:

1. Let \( M \) be the `this` value.
2. If `Type(M)` is not Object, then throw a `TypeError` exception.
3. If \( M \) does not have a `[[WeakMapData]]` internal slot throw a `TypeError` exception.
4. Let `entries` be the List that is the value of \( M \)'s `[[WeakMapData]]` internal slot.
5. If `entries` is `undefined`, then throw a `TypeError` exception.
6. If `Type(key)` is not Object, then return false.
7. Repeat for each Record `{[[key]], [[value]]} p` that is an element of `entries`,
   a. If `p.[[key]]` and `key` are the same object, then
      i. Set `p.[[key]]` to `empty`.
      ii. Set `p.[[value]]` to `empty`.
     iii. Return `true`.
8. Return `false`.

**NOTE** The value `empty` is used as a specification device to indicate that an entry has been deleted. Actual implementations may take other actions such as physically removing the entry from internal data structures.

#### 23.3.3.4 WeakMap.prototype.get(key)

The following steps are taken:

1. Let \( M \) be the `this` value.
2. If `Type(M)` is not Object, then throw a `TypeError` exception.
3. If \( M \) does not have a `[[WeakMapData]]` internal slot throw a `TypeError` exception.
4. Let entries be the List that is the value of M's [[WeakMapData]] internal slot.
5. If entries is undefined, then throw a TypeError exception.
6. If Type(key) is not Object, then return undefined.
7. Repeat for each Record ([[[key]], [[value]]]) p that is an element of entries,
   a. If p.[[key]] and key are the same object, then return p.[[value]].
8. Return undefined.

23.3.3.5 WeakMap.prototype.has (key)

The following steps are taken:
1. Let M be the this value.
2. If Type(M) is not Object, then throw a TypeError exception.
3. If M does not have a [[WeakMapData]] internal slot throw a TypeError exception.
4. Let entries be the List that is the value of M's [[WeakMapData]] internal slot.
5. If entries is undefined, then throw a TypeError exception.
6. If Type(key) is not Object, then return false.
7. Repeat for each Record ([[[key]], [[value]]]) p that is an element of entries,
   a. If p.[[key]] and key are the same object, then return true.
8. Return false.

23.3.3.6 WeakMap.prototype.set (key, value)

The following steps are taken:
1. Let M be the this value.
2. If Type(M) is not Object, then throw a TypeError exception.
3. If M does not have a [[WeakMapData]] internal slot throw a TypeError exception.
4. Let entries be the List that is the value of M's [[WeakMapData]] internal slot.
5. If entries is undefined, then throw a TypeError exception.
6. If Type(key) is not Object, then throw a TypeError exception.
7. Repeat for each Record ([[[key]], [[value]]]) p that is an element of entries,
   a. If p.[[key]] and key are the same object, then
      i. Set p.[[value]] to value.
      ii. Return M.
8. Let p be the Record [[[key], key], [[value], value]].
9. Append p as the last element of entries.
10. Return M.

23.3.3.7 WeakMap.prototype[@@toStringTag]

The initial value of the @@toStringTag property is the string value "WeakMap".

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

23.3.4 Properties of WeakMap Instances

WeakMap instances are ordinary objects that inherit properties from the WeakMap prototype. WeakMap instances also have a [[WeakMapData]] internal slot.

23.4 WeakSet Objects

WeakSet objects are collections of objects. A distinct object may only occur once as an element of a WeakSet’s collection. A WeakSet may be queried to see if it contains a specific object, but no mechanisms is provided for enumerating the objects it holds. If an object that is contain by a WeakSet is only reachable by following a chain of references that start within that WeakSet, then that object is inaccessible and is automatically removed from the WeakSet. WeakSet implementations must detect and remove such objects and any associated resources.
An implementation may impose an arbitrarily determined latency between the time an object contained in a WeakSet becomes inaccessible and the time when the object is removed from the WeakSet. 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 determine if a WeakSet contains a particular object that does not require the observer to present the observed object.

WeakSet objects must be implemented using either hash tables or other mechanisms that, on average, provide access times that are sublinear on the number of elements in the collection. The data structure used in this WeakSet objects specification is only intended to describe the required observable semantics of WeakSet objects. It is not intended to be a viable implementation model.

NOTE See the NOTE in 23.3.

### 23.4.1 The WeakSet Constructor

The WeakSet constructor is the %WeakSet% intrinsic object and the initial value of the WeakSet property of the global object. When WeakSet is called as a function rather than as a constructor, it initialises its this value with the internal state necessary to support the WeakSet.prototype internal methods.

The WeakSet constructor is designed to be subclassable. It may be used as the value in an extends clause of a class definition. Subclass constructors that intend to inherit the specified WeakSet behaviour must include a super call to WeakSet.

#### 23.4.1.1 WeakSet ( iterable = undefined )

When the WeakSet function is called with optional argument iterable the following steps are taken:

1. Let set be the this value.
2. If Type(set) is not Object then, throw a TypeError exception.
3. If set does not have a [[WeakSetData]] internal slot, then throw a TypeError exception.
4. If set’s [[WeakSetData]] internal slot is not undefined, then throw a TypeError exception.
5. If iterable is not present, let iterable be undefined.
6. If iterable is either undefined or null, then let iter be undefined.
7. Else,
   a. Let iter be the result of GetIterator(iterable).
   b. ReturnIfAbrupt(iter).
   c. Let adder be the result of Get(set, "add").
   d. ReturnIfAbrupt(adder).
   e. If IsCallable(adder) is false, throw a TypeError Exception.
8. Set set’s [[WeakSetData]] internal slot to a new empty List.
9. If iter is undefined, then return set.
10. Repeat
    a. Let next be the result of IteratorStep(iter).
    b. ReturnIfAbrupt(next).
    c. If next is false, then return NormalCompletion(set).
    d. Let nextValue be IteratorValue(next).
    e. ReturnIfAbrupt(nextValue).
    f. Let status be the result of calling the [[Call]] internal method of adder with set as thisArgument and a List whose sole element is nextValue as argumentsList.
    g. ReturnIfAbrupt(status).

#### 23.4.1.2 new WeakSet ( ... argumentsList )

When WeakSet is called as part of a new expression it is a constructor: it initialises a newly created object.
WeakSet called as part of a new expression with argument list argumentsList performs the following steps:

1. Let F be the WeakSet function object on which the new operator was applied.
2. Let argumentsList be the argumentsList argument of the [[Construct]] internal method that was invoked by the new operator.
3. Return the result of OrdinaryConstruct(F, argumentsList).

If WeakSet is implemented as an ECMAScript function object, its [[Construct]] internal method will perform the above steps.

23.4.2 Properties of the WeakSet Constructor

The value of the [[Prototype]] internal slot of the WeakSet constructor is the Function prototype object (19.2.3).

Besides the length property (whose value is 0), the WeakSet constructor has the following properties:

23.4.2.1 WeakSet.prototype

The initial value of WeakSet.prototype is the intrinsic %WeakSetPrototype% object (23.4.3).

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

23.4.2.2 WeakSet[@@create] ()

The @@create method of a WeakSet function object F performs the following steps:

1. Let F be the this value.
2. Let obj be the result of calling OrdinaryCreateFromConstructor(F, "%WeakSetPrototype%", ([[[WeakSetData]]])).
3. Return obj.

The value of the name property of this function is "[Symbol.create]".

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

23.4.3 Properties of the WeakSet Prototype Object

The value of the [[Prototype]] internal slot of the WeakSet prototype object is the standard built-in Object prototype object (19.1.3). The WeakSet prototype object is an ordinary object. It does not have a [[WeakSetData]] internal slot.

23.4.3.1 WeakSet.prototype.add (value)

The following steps are taken:

1. Let S be the this value.
2. If Type(S) is not Object, then throw a TypeError exception.
3. If S does not have a [[WeakSetData]] internal slot throw a TypeError exception.
4. If S’s [[WeakSetData]] internal slot is undefined, then throw a TypeError exception.
5. If Type(value) is not Object, then throw a TypeError exception.
6. Let entries be the List that is the value of S’s [[WeakSetData]] internal slot.
7. 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. Return S.
8. Append value as the last element of entries.
9. Return S.
23.4.3.2 WeakSet.prototype.clear()

The following steps are taken:

1. Let S be this value.
2. If Type(S) is not Object, then throw a TypeError exception.
3. If S does not have a [[WeakSetData]] internal slot throw a TypeError exception.
4. If S's [[WeakSetData]] internal slot is undefined, then throw a TypeError exception.
5. Set the value of S's [[WeakSetData]] internal slot to a new empty List.
6. Return undefined.

23.4.3.3 WeakSet.prototype.constructor

The initial value of WeakSet.prototype.constructor is the %WeakSet% intrinsic object.

23.4.3.4 WeakSet.prototype.delete(value)

The following steps are taken:

1. Let S be the this value.
2. If Type(S) is not Object, then throw a TypeError exception.
3. If S does not have a [[WeakSetData]] internal slot throw a TypeError exception.
4. If S's [[WeakSetData]] internal slot is undefined, then throw a TypeError exception.
5. If Type(value) is not Object, then return false.
6. Let entries be the List that is the value of S's [[WeakSetData]] internal slot.
7. 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.
   8. Return false.

NOTE The value empty is used as a specification device to indicate that an entry has been deleted. Actual implementations may take other actions such as physically removing the entry from internal data structures.

23.4.3.5 WeakSet.prototype.has(value)

The following steps are taken:

1. Let S be the this value.
2. If Type(S) is not Object, then throw a TypeError exception.
3. If S does not have a [[WeakSetData]] internal slot throw a TypeError exception.
4. If S's [[WeakSetData]] internal slot is undefined, then throw a TypeError exception.
5. Let entries be the List that is the value of S's [[WeakSetData]] internal slot.
6. If Type(value) is not Object, then return false.
7. Repeat for each e that is an element of entries,
   a. If e is not empty and SameValue(e, value), then return true.
   8. Return false.

23.4.3.6 WeakSet.prototype[@@toStringTag]

The initial value of the @@toStringTag property is the string value "WeakSet".

This property has the attributes ([Writable]: false, [[Enumerable]]: false, [[Configurable]]: true).
23.4.4 Properties of WeakSet Instances

WeakSet instances are ordinary objects that inherit properties from the WeakSet prototype. After initialisation by the WeakSet constructor, WeakSet instances also have a [[WeakSetData]] internal slot.

24 Structured Data

24.1 ArrayBuffer Objects

24.1.1 Abstract Operations For ArrayBuffer Objects

24.1.1.1 AllocateArrayBuffer(constructor)

The abstract operation AllocateArrayBuffer with argument constructor is used to create an uninitialised ArrayBuffer object. It performs the following steps:

1. Let obj be the result of calling OrdinaryCreateFromConstructor(constructor, %ArrayBufferPrototype%, ( [[ArrayBufferData]], [[ArrayBufferByteLength]] )).
2. ReturnIfAbrupt(obj).
3. Set the [[ArrayBufferByteLength]] internal slot of obj to 0.
4. Return obj.

24.1.1.2 SetArrayBufferData(arrayBuffer, bytes)

The abstract operation SetArrayBufferData with arguments arrayBuffer and bytes is used to initialise the storage block encapsulated by an ArrayBuffer object. It performs the following steps:

1. ReturnIfAbrupt(arrayBuffer).
2. Assert: Type(arrayBuffer) is Object and it has an [[ArrayBufferData]] internal slot.
3. Assert: bytes is a positive integer.
4. Let block be the result of CreateByteDataBlock(bytes).
5. ReturnIfAbrupt(block).
6. Set arrayBuffer’s [[ArrayBufferData]] to block.
7. Set arrayBuffer’s [[ArrayBufferByteLength]] internal slot to bytes.
8. Return arrayBuffer.

24.1.1.3 CloneArrayBuffer(srcBuffer, srcByteOffset)

The abstract operation CloneArrayBuffer 24.1.1.3 takes two parameters, an ArrayBuffer srcBuffer, an integer srcByteOffset. It creates a new ArrayBuffer whose data is a copy of srcBuffer’s data starting at srcByteOffset. This operation performs the following steps:

1. Assert: Type(srcBuffer) is Object and it has an [[ArrayBufferData]] internal slot.
2. Let srcBlock be the value of srcBuffer’s [[ArrayBufferData]] internal slot.
3. If srcBlock is undefined or null, then throw a TypeError exception.
4. Let srcLength be the value of srcBuffer’s [[ArrayBufferByteLength]] internal slot.
5. Let bufferConstructor be the result of Get(srcBuffer, "constructor").
6. ReturnIfAbrupt(bufferConstructor).
7. Assert: srcByteOffset ≤ srcLength.
8. Let cloneLength be srcLength–srcByteOffset.
9. If bufferConstructor is undefined, then let bufferConstructor be %ArrayBuffer%.
10. Let targetBuffer be the result of calling AllocateArrayBuffer(bufferConstructor).
11. Let status be the result of SetArrayBufferData(data, cloneLength).
12. ReturnIfAbrupt(status).
13. Let targetBlock be the value of targetBuffer’s [[ArrayBufferData]] internal slot.
14. Perform CopyDataBlock(targetBlock, 0, srcBlock, srcByteOffset, cloneLength).
15. Return targetBlock.
24.1.1.4 GetValueFromBuffer (arrayBuffer, byteIndex, type, isLittleEndian)

The abstract operation GetValueFromBuffer takes four parameters, an ArrayBuffer `arrayBuffer`, an integer `byteIndex`, a String `type`, and optionally a Boolean `isLittleEndian`. If `isLittleEndian` is not present, its default value is `undefined`. This operation performs the follow steps:

1. **Assert**: There are sufficient bytes in `arrayBuffer` starting at `byteIndex` to represent a value of `valueType`.
2. **Assert**: `byteIndex` is a positive integer.
3. Let `block` be `arrayBuffer`'s `[[ArrayBufferData]]` internal slot.
4. If `block` is `undefined` or `null`, then throw a `TypeError` exception.
5. Let `elementSize` be the Number value of the Element Size value specified in Table 36 for `valueType`.
6. Let `rawValue` be a List of `elementSize` containing, in order, the `elementSize` bytes starting at `byteIndex` of `block`.
7. If `isLittleEndian` is `undefined`, set `isLittleEndian` to either `true` or `false`. The choice is implementation dependent and should be the alternative that is most efficient for the implementation. An implementation must use the same value each time this step is executed and the same value must be used for the corresponding step in the SetValueInBuffer abstract operation.
8. If `isLittleEndian` is `false`, reverse the order of the elements of `rawValue`.
9. If `type` is “Float32”, then
   a. Let `value` be the byte elements of `rawValue` concatenated and interpreted as a little-endian bit string encoding of an IEEE 754-2008 binary32 value.
   b. If `value` is an IEEE 754-2008 binary32 NaN value, return the NaN Number value.
   c. Return the Number value that corresponds to `value`.
10. If `type` is “Float64”, then
    a. Let `value` be the byte elements of `rawValue` concatenated and interpreted as a little-endian bit string encoding of an IEEE 754-2008 binary64 value.
    b. If `value` is an IEEE 754-2008 binary64 NaN value, return the NaN Number value.
    c. Return the Number value that is encoded by corresponds to `value`.
11. If the first character of `type` is "U", then
    a. Let `intValue` be the byte elements of `rawValue` concatenated and interpreted as a bit string encoding of an unsigned little-endian binary number.
12. Else
    a. Let `intValue` be the byte elements of `rawValue` concatenated and interpreted as a bit string encoding of a binary little-endian 2's complement number of bit length `elementSize` × 8.
13. Return the Number value that corresponds to `intValue`.

24.1.1.5 SetValueInBuffer (arrayBuffer, byteIndex, type, value, isLittleEndian)

The abstract operation SetValueInBuffer takes five parameters, an ArrayBuffer `arrayBuffer`, an integer `byteIndex`, a String `type`, a Number `value`, and optionally a Boolean `isLittleEndian`. If `isLittleEndian` is not present, its default value is `undefined`. This operation performs the follow steps:

1. **Assert**: There are sufficient bytes in `arrayBuffer` starting at `byteIndex` to represent a value of `valueType`.
2. **Assert**: `byteIndex` is a positive integer.
3. Let `block` be `arrayBuffer`'s `[[ArrayBufferData]]` internal slot.
4. If `block` is `undefined` or `null`, then throw a `TypeError` exception.
5. Let `elementSize` be the Number value of the Element Size value specified in Table 36 for the row containing the value of `type` as its Element Type entry.
6. If `isLittleEndian` is `undefined`, set `isLittleEndian` to either `true` or `false`. The choice is implementation dependent and should be the alternative that is most efficient for the implementation. An implementation must use the same value each time this step is executed and the same value must be used for the corresponding step in the GetValueFromBuffer abstract operation.
7. If `type` is “Float32”, then
   a. Set `rawValue` to a List containing the 4 bytes that are the result of converting `value` to IEEE-868-2005 binary32 format using “Round to nearest, ties to even” rounding mode. If `isLittleEndian` is `false`, the bytes are arranged in big endian order. Otherwise, the bytes are arranged in little endian order.
order. If value is NaN, rawValue may be set to any implementation chosen non-signaling NaN encoding.

8. Else, if type is "Float64", then
   a. Set rawValue to a List containing the 8 bytes that are the IEEE-868-2005 binary64 format encoding of value. If isLittleEndian is false, the bytes are arranged in big endian order. Otherwise, the bytes are arranged in little endian order. If value is NaN, rawValue may be set to any implementation chosen non-signaling NaN encoding.

9. Else,
   a. Let n be the Size Element value in Table 36 for the row containing the value of type as its Element Type entry.
   b. Let convOp be the abstract operation named in the Conversion Operation column in Table 36 for the row containing the value of type as its Element Type entry.
   c. Let intValue be the result of calling convOp with value as its argument.
      i. If intValue ≥ 0, then
         a. Let rawBytes be a List containing the n-byte binary encoding of intValue. If isLittleEndian is false, the bytes are ordered in big endian order. Otherwise, the bytes are ordered in little endian order.
         b. Else, let rawBytes be a List containing the n-byte binary 2’s complement encoding of intValue. If isLittleEndian is false, the bytes are ordered in big endian order. Otherwise, the bytes are ordered in little endian order.

10. Store the individual bytes of rawBytes in order starting at position byteIndex of block.
11. Return NormalCompletion (undefined).

24.1.2 The ArrayBuffer Constructor

The ArrayBuffer constructor is the %ArrayBuffer% intrinsic object and the initial value of the ArrayBuffer property of the global object. When ArrayBuffer is called as a function rather than as a constructor, its this value must be an Object with an [[ArrayBufferData]] internal slot whose value is undefined. The ArrayBuffer constructor initialises the this value using the argument values.

The ArrayBuffer constructor is designed to be subclassable. It may be used as the value of an extends clause of a class declaration. Subclass constructors that intended to inherit the specified ArrayBuffer behaviour must include a super call to the ArrayBuffer constructor to initialise subclass instances.

24.1.2.1 ArrayBuffer(length)

ArrayBuffer called as function with argument length performs the following steps:

1. Let O be the this value.
2. If Type(O) is not Object or if O does not have an [[ArrayBufferData]] internal slot or if the value of O’s [[ArrayBufferData]] internal slot is not undefined, then
   a. Throw a TypeError exception.
3. Let numberOfLength be ToNumber(length).
4. Let byteLength be ToLength(numberLength).
5. ReturnIfAbrupt(byteLength).
6. If SameValueZero(numberLength, byteLength) is false, then throw a RangeError exception.
7. Return the result of SetArrayBufferData(O, byteLength).

24.1.2.2 new ArrayBuffer(...argumentsList)

ArrayBuffer called as part of a new expression performs the following steps:

1. Let F be the ArrayBuffer function object on which the new operator was applied.
2. Let argumentsList be the argumentsList argument of the [[Construct]] internal method that was invoked by the new operator.
3. Return the result of OrdinaryConstruct(F, argumentsList).

If ArrayBuffer is implemented as an ECMAScript function object, its [[Construct]] internal method will perform the above steps.
24.1.3 Properties of the ArrayBuffer Constructor

The value of the [[Prototype]] internal slot of the ArrayBuffer constructor is the Function prototype object (19.2.3).

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

24.1.3.1 ArrayBuffer.isView (arg)

The isView function takes one argument arg, and performs the following steps are taken:

1. If Type(arg) is not Object, return false.
2. If arg has a [[ViewedArrayBuffer]] internal slot, then return true.
3. Return false.

24.1.3.2 ArrayBuffer.prototype

The initial value of ArrayBuffer.prototype is the ArrayBuffer prototype object (24.1.4).

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

24.1.3.3 ArrayBuffer @@create

The @@create method of an ArrayBuffer function object F performs the following steps:

1. Let F be the this value.
2. Return the result of calling AllocateArrayBuffer(F).

The value of the name property of this function is “[Symbol.create]”.

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

24.1.4 Properties of the ArrayBuffer Prototype Object

The value of the [[Prototype]] internal slot of the ArrayBuffer prototype object is the standard built-in Object prototype object (19.1.3). The ArrayBuffer prototype object is an ordinary object. It does not have a [[ArrayBufferData]] or [[ArrayBufferByteLength]] internal slot.

24.1.4.1 ArrayBuffer.prototype.byteLength

ArrayBuffer.prototype.byteLength is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let O be the this value.
2. If Type(O) is not Object, throw a TypeError exception.
3. If O does not have an [[ArrayBufferData]] internal slot throw a TypeError exception.
4. If the value of O’s [[ArrayBufferData]] internal slot is undefined or null, then throw a TypeError exception.
5. Let length be the value of O’s [[ArrayBufferByteLength]] internal slot.
6. Return length.

24.1.4.2 ArrayBuffer.prototype.constructor

The initial value of ArrayBuffer.prototype.constructor is the standard built-in ArrayBuffer constructor.
24.1.4.3 `ArrayBuffer.prototype.slice(start, end)`

The following steps are taken:

1. Let O be the this value.
2. If Type(O) is not Object, throw a `TypeError` exception.
3. If O does not have an `[[ArrayBufferData]]` internal slot throw a `TypeError` exception.
4. If the value of O’s `[[ArrayBufferData]]` internal slot is `undefined` or `null`, then throw a `TypeError` exception.
5. Let len be the value of O’s `[[ArrayBufferByteLength]]` internal slot.
6. Let relativeStart be `ToInteger(start)`.
7. ReturnIfAbrupt(relativeStart).
8. If relativeStart is negative, let first be `max(len + relativeStart, 0)`; else let first be `min(relativeStart, len)`.
9. If `relativeEnd` is undefined, let end be `len`; else let end be `ToInteger(end)`.
10. ReturnIfAbrupt(relativeEnd).
11. If relativeEnd is negative, let final be `max(len + relativeEnd, 0)`; else let final be `min(relativeEnd, len)`.
12. Let newLen be `max(final - first, 0)`.
13. Let ctor be the result of Get(O, "constructor").
14. ReturnIfAbrupt(ctor).
15. If IsConstructor(ctor) is false, then throw a `TypeError` exception.
16. Let new be the result of calling the `[[Construct]]` internal method of ctor with a new List containing the single element newLen.
17. ReturnIfAbrupt(new).
18. If new does not have an `[[ArrayBufferData]]` internal slot throw a `TypeError` exception.
19. If the value of new’s `[[ArrayBufferData]]` internal slot is `undefined`, then throw a `TypeError` exception.
20. If the value of new’s `[[ArrayBufferByteLength]]` < newLen, then throw a `TypeError` exception.
21. Let fromBuf be the value of O’s `[[ArrayBufferData]]` internal slot.
22. Let toBuf be the value of new’s `[[ArrayBufferData]]` internal slot.
23. Perform `CopyDataBlockBytes(toBuf, 0, fromBuf, first, newLen)`.

24.1.4.4 `ArrayBuffer.prototype[@@toStringTag]`

The initial value of the `[@@toStringTag]` property is the string value "ArrayBuffer".

24.1.5 Properties of the ArrayBuffer Instances

ArrayBuffer instances inherit properties from the ArrayBuffer prototype object. ArrayBuffer instances each have an `[[ArrayBufferData]]` internal slot and an `[[ArrayBufferByteLength]]` internal slot.

ArrayBuffer instances whose `[[ArrayBufferData]]` is `null` are considered to be neutered and all operators to access or modify data contained in the ArrayBuffer instance will fail.

24.2 DataView Objects

24.2.1 Abstract Operations For DataView Objects

24.2.1.1 `GetViewValue(view, requestIndex, isLittleEndian, type)`

The abstract operation `GetViewValue` with arguments `view`, `requestIndex`, `isLittleEndian`, and `type` is used by functions on DataView instances to retrieve values from the view's buffer. It performs the following steps:

1. If `Type(view)` is not Object, throw a `TypeError` exception.
2. If `view` does not have a `[[DataView]]` internal slot, then throw a `TypeError` exception.
3. Let buffer be the value of `view`'s `[[ViewedArrayBuffer]]` internal slot.
4. If `buffer` is `undefined`, then throw a `TypeError` exception.
5. Let `numberIndex` be `ToNumber(requestIndex)`.
6. Let `getIndex` be `ToInteger(numberIndex)`.
7. ReturnIfAbrupt(getIndex).
8. If `numberIndex` ≤ `getIndex` or `getIndex` < 0, then throw a RangeError exception.
9. Let `isLittleEndian` be ToBoolean(`isLittleEndian`).
10. ReturnIfAbrupt(`isLittleEndian`).
11. Let `viewOffset` be the value of `view`'s [[ByteOffset]] internal slot.
12. Let `viewSize` be the value of `view`'s [[ByteLength]] internal slot.
13. Let `elementSize` be the Number value of the Element Size value specified in Table 36 for `type`.
14. If `getIndex + elementSize > viewSize`, then throw a RangeError exception.
15. Let `bufferIndex` be `getIndex + viewOffset`.
16. Return the result of GetValueFromBuffer(`buffer`, `bufferIndex`, `isLittleEndian`).

24.2.1.2 SetViewValue(view, requestIndex, isLittleEndian, type, value)

The abstract operation `SetViewValue` with arguments `view`, `requestIndex`, `isLittleEndian`, `type`, and `value` is used by functions on `DataView` instances to store values into the view's buffer. It performs the following steps:

1. If Type(`view`) is not Object, throw a TypeError exception.
2. If `view` does not have a `[[DataView]]` internal slot, then throw a TypeError exception.
3. Let `buffer` be the value of `view`'s `[[ViewedArrayBuffer]]` internal slot.
4. If `buffer` is undefined, then throw a TypeError exception.
5. Let `numberIndex` be ToInteger(`requestIndex`).
6. Let `getIndex` be ToInteger(`numberIndex`).
7. ReturnIfAbrupt(`getIndex`).
8. If `numberIndex` ≠ `getIndex` or `getIndex` < 0, then throw a RangeError exception.
9. Let `isLittleEndian` be ToBoolean(`isLittleEndian`).
10. ReturnIfAbrupt(`isLittleEndian`).
11. Let `viewOffset` be the value of `view`'s `[[ByteOffset]]` internal slot.
12. Let `viewSize` be the value of `view`'s `[[ByteLength]]` internal slot.
13. Let `elementSize` be the Number value of the Element Size value specified in Table 36 for `type`.
14. If `getIndex + elementSize > viewSize`, then throw a RangeError exception.
15. Let `bufferIndex` be `getIndex + viewOffset`.
16. Return the result of SetValueInBuffer(`buffer`, `bufferIndex`, `isLittleEndian`).

NOTE The algorithms for `GetViewValue` and `SetViewValue` are identical except for their final steps.

24.2.2 The DataView Constructor

The DataView constructor is the %DataView% intrinsic object and the initial value of the DataView property of the global object. When DataView is called as a function rather than as a constructor, it initialises its this value with the internal state necessary to support the DataView.prototype internal methods.

The DataView constructor is designed to be subclassable. It may be used as the value of an extends clause of a class declaration. Subclass constructors that intend to inherit the specified DataView behaviour must include a super call to the DataView constructor to initialise subclass instances.

24.2.2.1 DataView(buffer, byteOffset=0, byteLength=undefined)

DataView called with arguments `buffer`, `byteOffset`, and `length` performs the following steps:

1. Let `O` be the this value.
2. If Type(`O`) is not Object or if `O` does not have a `[[DataView]]` internal slot, throw a TypeError exception.
3. Assert: `O` has a `[[ViewedArrayBuffer]]` internal slot.
4. If the value of `O`'s `[[ViewedArrayBuffer]]` internal slot is not undefined, then
   a. Throw a TypeError exception.
5. If Type(`buffer`) is not Object, then throw a TypeError exception.
6. If `buffer` does not have an `[[ArrayBufferData]]` internal slot, then throw a TypeError exception.
If the value of buffer’s [[ArrayBufferData]] internal slot is `undefined`, then throw a `TypeError` exception.

Let `numberOffset` be `ToNumber(byteOffset)`.

Let `of fset` be `ToInteger(numberOffset)`.

ReturnIfAbrupt(offset).

If `numberOffset ≠ offset` or `offset < 0`, then throw a `RangeError` exception.

Let `bufferByteLength` be the value of buffer’s [[ArrayBufferByteLength]] internal slot.

If `offset > bufferByteLength`, then throw a `RangeError` exception.

If `byteLength` is `undefined`, then

a. Let `viewByteLength` be `bufferByteLength – offset`.

Else,

a. Let `numberLength` be `ToNumber(byteLength)`.

b. Let `viewLength` be `ToInteger(numberLength)`.

c. ReturnIfAbrupt(viewLength).

d. If `numberLength ≠ viewLength` or `viewLength < 0`, then throw a `RangeError` exception.

e. Let `viewByteLength` be `viewLength`.

f. If `offset + viewByteLength > bufferByteLength`, then throw a `RangeError` exception.

If the value of `O`’s [[ViewedArrayBuffer]] internal slot is not `undefined`, then throw a `TypeError` exception.

Set `O`’s [[ViewedArrayBuffer]] to `buffer`.

Set `O`’s [[ByteLength]] internal slot to `viewByteLength`.

Set `O`’s [[ByteOffset]] internal slot to `offset`.

Return `O`.

`new DataView( ... argumentsList)`

When `DataView` is called as part of a new expression it performs the following steps:

1. Let `F` be the function object on which the `new` operator was applied.

2. Let `argumentsList` be the `argumentsList` argument of the `[[Construct]]` internal method that was invoked by the `new` operator.

3. Return the result of `OrdinaryConstruct(F, argumentsList)`.

If `DataView` is implemented as an ECMAscript function object, its `[[Construct]]` internal method will perform the above steps.

24.2.3 Properties of the DataView Constructor

The value of the `[[Prototype]]` internal slot of the `DataView` constructor is the `Function` prototype object (19.2.3).

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

24.2.3.1 DataView.prototype

The initial value of `DataView.prototype` is the DataView prototype object (24.2.4).

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

24.2.3.2 DataView [ @@create ] ()

The `@@create` method of a DataView function object `F` performs the following steps:

1. Let `F` be the `this` value.

2. Let `obj` be the result of calling `OrdinaryCreateFromConstructor(F, "DataViewPrototype", ( [ [ DataView ], [ [ ViewedArrayBuffer ] ], [ [ ByteLength ] ], [ [ ByteOffset ] ] ] )`).

3. Set the value of `obj`’s `[[DataView]]` internal slot to `true`.

4. Return `obj`.

The value of the `name` property of this function is "`[Symbol.create]` ".

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.
NOTE The value of the [[DataView]] internal slot is not used within this specification. The simple presence of that internal slot is used within the specification to identify objects created using this @@create method.

24.2.4 Properties of the DataView Prototype Object

The value of the [[Prototype]] internal slot of the DataView prototype object is the standard built-in Object prototype object (19.1.3). The DataView prototype object is an ordinary object. It does not have a [[DataView]], [[ViewedArrayBuffer]], [[ByteLength]], or [[ByteOffset]] internal slot.

24.2.4.1 get DataView.prototype.buffer

DataView.prototype.buffer is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let O be the this value.
2. If Type(O) is not Object, throw a TypeError exception.
3. If O does not have a [[ViewedArrayBuffer]] internal slot throw a TypeError exception.
4. Let buffer be the value of O’s [[ViewedArrayBuffer]] internal slot.
5. If buffer is undefined, then throw a TypeError exception.
6. Return buffer.

24.2.4.2 get DataView.prototype.byteLength

DataView.prototype.byteLength is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let O be the this value.
2. If Type(O) is not Object, throw a TypeError exception.
3. If O does not have a [[ViewedArrayBuffer]] internal slot throw a TypeError exception.
4. Let buffer be the value of O’s [[ViewedArrayBuffer]] internal slot.
5. If buffer is undefined, then throw a TypeError exception.
6. Let size be the value of O’s [[ByteLength]] internal slot.
7. Return size.

24.2.4.3 get DataView.prototype.byteOffset

DataView.prototype.byteOffset is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let O be the this value.
2. If Type(O) is not Object, throw a TypeError exception.
3. If O does not have a [[ViewedArrayBuffer]] internal slot throw a TypeError exception.
4. Let buffer be the value of O’s [[ViewedArrayBuffer]] internal slot.
5. If buffer is undefined, then throw a TypeError exception.
6. Let offset be the value of O’s [[ByteOffset]] internal slot.
7. Return offset.

24.2.4.4 DataView.prototype.constructor

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

24.2.4.5 DataView.prototype.getFloat32(byteOffset, littleEndian=false)

When the getFloat32 method is called with argument byteOffset and optional argument littleEndian the following steps are taken:
1. Let \( v \) be the this value.
2. If \( \text{littleEndian} \) is not present, then let \( \text{littleEndian} \) be \( \text{false} \).
3. Return the result of GetViewValue(\( v \), byteOffset, \( \text{littleEndian} \), "\text{Float32}").

24.2.4.6 DataView.prototype.getFloat64(byteOffset, littleEndian=false)

When the \text{getFloat64} method is called with argument \( \text{byteOffset} \) and optional argument \( \text{littleEndian} \) the following steps are taken:
1. Let \( v \) be the this value.
2. If \( \text{littleEndian} \) is not present, then let \( \text{littleEndian} \) be \( \text{false} \).
3. Return the result of GetViewValue(\( v \), byteOffset, \( \text{littleEndian} \), "\text{Float64}").

24.2.4.7 DataView.prototype.getInt8(byteOffset)

When the \text{getInt8} method is called with argument \( \text{byteOffset} \) the following steps are taken:
1. Let \( v \) be the this value.
2. Return the result of GetViewValue(\( v \), byteOffset, undefined, "\text{Int8}").

24.2.4.8 DataView.prototype.getInt16(byteOffset, littleEndian=false)

When the \text{getInt16} method is called with argument \( \text{byteOffset} \) and optional argument \( \text{littleEndian} \) the following steps are taken:
1. Let \( v \) be the this value.
2. If \( \text{littleEndian} \) is not present, then let \( \text{littleEndian} \) be \( \text{false} \).
3. Return the result of GetViewValue(\( v \), byteOffset, \( \text{littleEndian} \), "\text{Int16}").

24.2.4.9 DataView.prototype.getInt32(byteOffset, littleEndian=false)

When the \text{getInt32} method is called with argument \( \text{byteOffset} \) and optional argument \( \text{littleEndian} \) the following steps are taken:
1. Let \( v \) be the this value.
2. If \( \text{littleEndian} \) is not present, then let \( \text{littleEndian} \) be \( \text{undefined} \).
3. Return the result of GetViewValue(\( v \), byteOffset, \( \text{littleEndian} \), "\text{Int32}").

24.2.4.10 DataView.prototype.getUint8(byteOffset)

When the \text{getUint8} method is called with argument \( \text{byteOffset} \) the following steps are taken:
1. Let \( v \) be the this value.
2. Return the result of GetViewValue(\( v \), byteOffset, undefined, "\text{Uint8}").

24.2.4.11 DataView.prototype.getUint16(byteOffset, littleEndian=false)

When the \text{getUint16} method is called with argument \( \text{byteOffset} \) and optional argument \( \text{littleEndian} \) the following steps are taken:
1. Let \( v \) be the this value.
2. If \( \text{littleEndian} \) is not present, then let \( \text{littleEndian} \) be \( \text{false} \).
3. Return the result of GetViewValue(\( v \), byteOffset, \( \text{littleEndian} \), "\text{Uint16}").

24.2.4.12 DataView.prototype.getUint32(byteOffset, littleEndian=false)

When the \text{getUint32} method is called with argument \( \text{byteOffset} \) and optional argument \( \text{littleEndian} \) the following steps are taken:
Let $v$ be the this value.

1. If $\text{littleEndian}$ is not present, then let $\text{littleEndian} = \text{false}$.
2. Return the result of $\text{GetViewValue}(v, \text{byteOffset}, \text{littleEndian}, \text{"Uint32"})$.

### 24.2.4.13 DataView.prototype.setFloat32(byteOffset, value, littleEndian=false)

When the `setFloat32` method is called with arguments `byteOffset` and `value` and optional argument `littleEndian` the following steps are taken:

1. Let $v$ be the this value.
2. If $\text{littleEndian}$ is not present, then let $\text{littleEndian} = \text{false}$.
3. Return the result of $\text{SetViewValue}(v, \text{byteOffset}, \text{littleEndian}, \text{"Float32"}, value)$.

### 24.2.4.14 DataView.prototype.setFloat64(byteOffset, value, littleEndian=false)

When the `setFloat64` method is called with arguments `byteOffset` and `value` and optional argument `littleEndian` the following steps are taken:

1. Let $v$ be the this value.
2. If $\text{littleEndian}$ is not present, then let $\text{littleEndian} = \text{false}$.
3. Return the result of $\text{SetViewValue}(v, \text{byteOffset}, \text{littleEndian}, \text{"Float64"}, value)$.

### 24.2.4.15 DataView.prototype.setInt8(byteOffset, value)

When the `setInt8` method is called with arguments `byteOffset` and `value` the following steps are taken:

1. Let $v$ be the this value.
2. Return the result of $\text{SetViewValue}(v, \text{byteOffset}, \text{undefined}, \text{"Int8"}, value)$.

### 24.2.4.16 DataView.prototype.setInt16(byteOffset, value, littleEndian=false)

When the `setInt16` method is called with arguments `byteOffset` and `value` and optional argument `littleEndian` the following steps are taken:

1. Let $v$ be the this value.
2. If $\text{littleEndian}$ is not present, then let $\text{littleEndian} = \text{false}$.
3. Return the result of $\text{SetViewValue}(v, \text{byteOffset}, \text{littleEndian}, \text{"Int16"}, value)$.

### 24.2.4.17 DataView.prototype.setInt32(byteOffset, value, littleEndian=false)

When the `setInt32` method is called with arguments `byteOffset` and `value` and optional argument `littleEndian` the following steps are taken:

1. Let $v$ be the this value.
2. If $\text{littleEndian}$ is not present, then let $\text{littleEndian} = \text{false}$.
3. Return the result of $\text{SetViewValue}(v, \text{byteOffset}, \text{littleEndian}, \text{"Int32"}, value)$.

### 24.2.4.18 DataView.prototype.setUint8(byteOffset, value)

When the `setUint8` method is called with arguments `byteOffset` and `value` the following steps are taken:

1. Let $v$ be the this value.
2. Return the result of $\text{SetViewValue}(v, \text{byteOffset}, \text{undefined}, \text{"Uint8"}, value)$.
24.2.4.19 DataView.prototype.setUint16(byteOffset, value, littleEndian=false)

When the setUint16 method is called with arguments byteOffset and value and optional argument littleEndian the following steps are taken:

1. Let v be the this value.
2. If littleEndian is not present, then let littleEndian be false.
3. Return the result of SetViewValue(v, byteOffset, littleEndian, "Uint16", value).

24.2.4.20 DataView.prototype.setUint32(byteOffset, value, littleEndian=false)

When the setUint32 method is called with arguments byteOffset and value and optional argument littleEndian the following steps are taken:

1. Let v be the this value.
2. If littleEndian is not present, then let littleEndian be false.
3. Return the result of SetViewValue(v, byteOffset, littleEndian, "Uint32", value).

24.2.4.21 DataView.prototype[@@toStringTag]

The initial value of the @@toStringTag property is the string value "DataView".

24.2.5 Properties of DataView Instances

DataView instances are ordinary objects that inherit properties from the DataView prototype object. DataView instances each have a [[DataView]], [[ViewedArrayBuffer]], [[ByteLength]], and [[ByteOffset]] internal slots.

24.3 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 defined in ECMA-404. The JSON interchange format used in this specification is exactly that described by ECMA-404.

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.

The value of the [[Prototype]] internal slot of the JSON object is the standard built-in Object prototype object (19.1.3). The value of the [[Extensible]] internal slot 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.

24.3.1 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 subset of the syntax for ECMAScript literals, Array Initialisers and Object Initialisers. After parsing, JSON objects are realized as ECMAScript objects. JSON arrays are realized as ECMAScript Array instances. JSON strings, numbers, booleans, and null are realized as ECMAScript Strings, Numbers, Booleans, and null.

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 undefined then the property is deleted from the result.
1. Let \( \text{JText} \) be \( \text{ToString} (\text{text}) \).
2. ReturnIfAbrupt(\( \text{JText} \)).
3. Parse \( \text{JText} \) interpreted as UTF-16 encoded Unicode points as a JSON text as specified in ECMA-404. Throw a SyntaxError exception if \( \text{JText} \) is not a valid JSON text as defined in that specification.
4. Let \( \text{scriptText} \) be the result of concatenating " \( (\) \( \text{JText} \) \( ) \) ; ".
5. Let \( \text{unfiltered} \) be the result of parsing and evaluating \( \text{scriptText} \) as if it was the source text of an ECMAScript Script, but using the alternative definition of DoubleStringCharacter provided below. The extended PropertyDefinition/evaluation semantics defined in B.3.1 must not be used during the evaluation.
6. Let \( \text{unfiltered} \) be completion.[[value]].
7. Assert: \( \text{unfiltered} \) will be either a primitive value or an object that is defined by either an ArrayLiteral or an ObjectLiteral.
8. If IsCallable(\( \text{reviver} \)) is true, then
   a. Let \( \text{root} \) be the result of the abstract operation ObjectCreate with the intrinsic object %ObjectPrototype% as its argument.
   b. Let \( \text{status} \) be the result of CreateDataProperty(\( \text{root} \), the empty String, \( \text{unfiltered} \)).
   c. Assert: \( \text{status} \) is true.
   d. Return the result of calling the abstract operation Walk, passing \( \text{root} \) and the empty String. The abstract operation Walk is described below.
9. Else
   a. Return \( \text{unfiltered} \).

JSON allows Unicode code points U+2028 and U+2029 to directly appear in String literals without using an escape sequence. This is enabled by using the following alternative definition of DoubleStringCharacter when parsing \( \text{scriptText} \) in step 5:

\[
\text{DoubleStringCharacter} :: \text{SourceCharacter \ but \ not \ one \ of \ " or \ \ or \ U+0000 \ through \ U+001F EscapeSequence}
\]

• The CV of \( \text{DoubleStringCharacter} :: \text{SourceCharacter \ but \ not \ one \ of \ " or \ \ or \ U+0000 \ through \ U+001F EscapeSequence} \) is the UTF-16 Encoding (10.1.1) of the code point value of \( \text{SourceCharacter} \).

NOTE The syntax of a valid JSON text is a subset of the ECMAScript PrimaryExpression syntax. Hence a valid JSON text is also a valid PrimaryExpression. Step 3 above verifies that \( \text{JText} \) conforms to that subset. When \( \text{scriptText} \) is parsed and evaluated as a Script the result will be either a String, Number, Boolean, or Null primitive value or an Object defined as if by an ArrayLiteral or ObjectLiteral.

24.3.1.1 Runtime Semantics: Walk Abstract Operation

The abstract operation Walk is a recursive abstract operation that takes two parameters: a \( \text{holder} \) object and the String \( \text{name} \) of a property in that object. Walk uses the value of \( \text{reviver} \) that was originally passed to the above parse function.

1. Let \( \text{val} \) be the result of Get(\( \text{holder} \), \( \text{name} \)).
2. ReturnIfAbrupt(\( \text{val} \)).
3. If \( \text{val} \) is an object, then
   a. If \( \text{val} \) is an exotic Array object then
      i. Set \( J \) to 0.
      ii. Let \( \text{len} \) be the result of Get(\( \text{val} \), "length").
      iii. Assert: \( \text{len} \) is not an abrupt completion and its value is a positive integer.
      iv. Repeat while \( J < \text{len} \),
         1. Let newElement be the result of calling the abstract operation Walk, passing \( \text{val} \) and ToString(\( J \)).
         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 PropertyDescriptor{[[Value]]: newElement, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}.
   b. NOTE This algorithm intentionally does not throw an exception if status is false.
4. ReturnIfAbrupt(status).
5. Add 1 to I.

b. Else
i. Let keys be a 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.
   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 PropertyDescriptor{[[Value]]: newElement, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}.
      b. NOTE This algorithm intentionally does not throw an exception if status is false.
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.

24.3.2 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 is an exotic Array object, then
      i. Let PropertyList be an empty 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 slot, 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 [[StringData]] or [[NumberData]] internal slot, then let item be ToString(v).
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 status be the result of the abstract operation ObjectCreate with the intrinsic object %ObjectPrototype% as its argument.
10. Let status be the result of CreateDataProperty(wrapper, the empty String, value).
11. Assert: status is true.
12. Return the result of calling the abstract operation Str(the empty String, wrapper).

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:

   
   ```javascript
   a = [ ];
   a[0] = a;
   my_text = JSON.stringify(a); // This must throw a TypeError.
   ```

NOTE 2 Symbolic primitive values are rendered as follows:

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

NOTE 3 String values are wrapped in double quotes. The characters " and \ are escaped with \ prefixes. Control characters are replaced with escape sequences \uHHHH, or with the shorter forms, \b (backspace), \f (formfeed), \n (newline), \r (carriage return), \t (tab).

NOTE 4 Finite numbers are stringified as if by calling ToString(number). NaN and Infinity regardless of sign are represented as the String null.

NOTE 5 Values that do not have a JSON representation (such as undefined and functions) do not produce a String. Instead they produce the undefined value. In arrays these values are represented as the String null. In objects an unrepresentable value causes the property to be excluded from stringification.

NOTE 6 An object is rendered as an opening left brace followed by zero or more properties, separated with commas, closed with a right brace. A property is a quoted String representing the key or property name, a colon, and then the stringified property value. An array is rendered as an opening left bracket followed by zero or more values, separated with commas, closed with a right bracket.
24.3.2.1 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 `getJSON` be the result of `Get(`value, "toJSON")`.
   b. If `IsCallable(`getJSON`)` is `true`
      i. Let `value` be the result of calling the `[[Call]]` internal method of `getJSON` passing `value` as `thisArgument` and a `List` containing `key` as `argumentsList`.
      ii. ReturnIfAbrupt(`value`).
   c. If `Type(`value`)` is `Object`, then
      i. Let `toJSON` be the result of `Get(`value, "toJSON")`.
      ii. If `IsCallable(`toJSON`)` is `true`
         i. Let `value` be the result of calling the `[[Call]]` internal method of `toJSON` passing `value` as `this` `Argument` and a `List` containing `key` as `argumentsList`.
            ii. ReturnIfAbrupt(`value`).
      iii. If `ReplacerFunction` is not `undefined`, then
            a. Let `value` be the result of calling the `[[Call]]` internal method of `ReplacerFunction` passing `holder` as `this` `value` and with an `argument list` consisting of `key` and `value`.
            b. ReturnIfAbrupt(`value`).
   d. If `ReplacerFunction` is `undefined`, then throw a `TypeError` exception.
4. If `ReplacerFunction` is not `undefined`, then
   a. Let `value` be the result of calling the `[[Call]]` internal method of `ReplacerFunction` passing `holder` as `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 a `[[NumberData]]` internal slot then,
      i. Let `value` be `ToNumber(`value`)`.
   b. Else if `value` has a `[[StringData]]` internal slot then,
      i. Let `value` be `ToString(`value`)`.
   c. Else if `value` has a `[[BooleanData]]` internal slot then,
      i. Let `value` be the value of the `[[BooleanData]]` internal slot of `value`.
      ii. If `value` is `undefined`, then throw a `TypeError` exception.
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` is an exotic `Array object` 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`.

24.3.2.2 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`.
   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.

24.3.2.3 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.
   Else
a. Let K be a List of Strings consisting of the keys of all the own properties of value whose [[Enumerable]] attribute is true and whose property key is a String value. 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 Str(P, 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).
           ii. Let member be the concatenation of member and strP.
           iii. 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.

24.3.2.4 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. Properties whose keys are not array indexes 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 lenVal be the result of Get(value, "length").
8. Let len be ToLength(lenVal).
9. ReturnIfAbrupt(len).
10. Let index be 0.
11. Repeat while index < len
   a. Let strP be the result of calling the abstract operation Str(ToString(index), 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.
12. If partial is empty, then
   a. Let final be "["]
13. 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 "]".
14. Remove the last element of stack.
15. Let indent be stepback.

24.3.3 JSON @@toStringTag

The initial value of the @@toStringTag property is the string value "JSON".

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.
25 Control Abstraction Objects

25.1 Common Iteration Interfaces

An interface is a set of object property keys whose associated values match a specific specification. Any object that provides all the properties of an interface in conformance to the interface's specification conforms to that interface. An interface isn't represented by a single object and there may be many distinctly implemented objects that conform to any interface. An individual object may conform to multiple interfaces.

25.1.1 The \texttt{Iterable} Interface

The \texttt{Iterable} interface includes the following property:

\begin{tabular}{|l|l|}
\hline
Property & Value \\
\hline
@@iterator & A zero arguments function that returns an object. \\
\hline
Requirements & The function returns an object that conforms to the \texttt{iterator} interface. \\
\hline
\end{tabular}

25.1.2 The \texttt{Iterator} Interface

The \texttt{Iterator} interface includes the following properties:

\begin{tabular}{|l|l|}
\hline
Property & Value \\
\hline
next & A function that returns an object. \\
\hline
Requirements & The function returns an object that conforms to the \texttt{IteratorResult} interface. If a previous call to the \texttt{next} method of an \texttt{Iterator} has returned an \texttt{IteratorResult} object whose \texttt{done} property is \texttt{true}, then all subsequent calls to the \texttt{next} method of that object must also return an \texttt{IteratorResult} object whose \texttt{done} property is \texttt{true}. \\
\hline
\end{tabular}

\textbf{NOTE} Arguments may be passed to the \texttt{next} function but their interpretation and validity is dependent upon the target \texttt{Iterator}. Generic use of \texttt{Iterators} should not pass any arguments.

25.1.3 The \texttt{IteratorResult} Interface

The \texttt{IteratorResult} interface includes the following properties:

\begin{tabular}{|l|l|}
\hline
Property & Value \\
\hline
done & Either true or false. \\
\hline
Requirements & This is the result status of the an \texttt{Iterator} \texttt{next} method call. If the end of the iterator was reached \texttt{done} is \texttt{true}. If the end was not reached \texttt{done} is \texttt{false} and a value is available. \\
\hline
value & Any ECMAScript language value. \\
\hline
Requirements & If \texttt{done} is \texttt{false}, this is the current iteration element value. If \texttt{done} is \texttt{true}, this is the return value of the \texttt{Iterator}, if it supplied one. If the \texttt{Iterator} does not have a return value, \texttt{value} is \texttt{undefined}. In that case, the \texttt{value} property may be absent form the conforming object if it does not inherit an explicit \texttt{value} property. \\
\hline
\end{tabular}
25.2 GeneratorFunction Objects

Generator Function objects are constructor functions that are usually created by evaluating GeneratorDeclaration, GeneratorExpression, and GeneratorMethod syntactic productions. They may also be created by calling the GeneratorFunction constructor.

Figure 2 (Informative) -- Generator Objects Relationships

25.2.1 The GeneratorFunction Constructor

The GeneratorFunction constructor is the %GeneratorFunction% intrinsic object and the value of the name GeneratorFunction exported from the built-in module "std:iteration". When GeneratorFunction is called as a function rather than as a constructor, it creates and initialises a new GeneratorFunction object. Thus the function call GeneratorFunction (...) is equivalent to the object creation expression new GeneratorFunction(...) with the same arguments. However, if the this value passed in the call is an Object with a [[Code]] internal slot whose value is undefined, it initialises the this value using the argument values. This permits GeneratorFunction to be used both as factory method and to perform constructor instance initialisation.

Commented [AWB15147]: Before final publication we should try to get a vector graphics version of this diagram.

Commented [AWB21148]: TODO: some of the comment nodes in the diagram need to be updated.


GeneratorFunction may be subclassed and subclass constructors may perform a super invocation of the GeneratorFunction constructor to initialise subclass instances. However, all syntactic forms for defining generator function objects create direct instances of GeneratorFunction. There is no syntactic means to create instances of GeneratorFunction subclasses.

25.2.1.1 GeneratorFunction (p1, p2, …, pn, body)

The last argument specifies the body (executable code) of a generator function; any preceding arguments specify formal parameters.

When the GeneratorFunction 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. 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 clause 10.1.1, using FormalParameters as the goal symbol. Throw a SyntaxError exception if the parse fails.
9. Let funcBody be the result of parsing bodyText, interpreted as UTF-16 encoded Unicode text as described in clause 10.1.1, using FunctionBody as the goal symbol. Throw a SyntaxError exception if the parse fails or if any static semantics errors are detected.
10. If IsSimpleParameterList of parameters is false and any element of the BoundNames of parameters also occurs in the VarDeclaredNames of funcBody, then throw a SyntaxError exception.
11. If any element of the BoundNames of parameters also occurs in the LexicallyDeclaredNames of funcBody, then throw a SyntaxError exception.
12. If bodyText is strict mode code (see 10.2.1) then let strict be true, else let strict be false.
13. Let scope be the Global Environment.
14. Let F be the this value.
15. If Type(F) is not Object or if F does not have a [[Code]] internal slot or if the value of [[Code]] is not undefined, then
   a. Let F be the result of calling FunctionAllocate with arguments %Generator%, strict, and ”generator”.
16. Else, set F’s [[Strict]] internal slot to strict.
17. If the value of F’s [[FunctionKind]] internal slot is not “generator”, then throw a TypeError exception.
18. Using funcBody as the FunctionBody production, let body be the supplemental syntactic grammar production: GeneratorBody ; FunctionBody.
19. Perform the FunctionInitialise abstract operation with arguments F, Normal, parameters, body, and scope.
20. Let prototype be the result of the abstract operation ObjectCreate with the intrinsic object %GeneratorPrototype% as its argument.
21. Perform the abstract operation MakeConstructor with arguments $F$, true, and $prototype$.
22. SetFunctionName($F$, "anonymous").
23. Return $F$.

A $prototype$ property is automatically created for every function created using the GeneratorFunction constructor, to provide for the possibility that the function will be used as a constructor.

25.2.1.2 new GeneratorFunction ( ... argumentsList)

When GeneratorFunction is called as part of a new expression, it creates and initialises a newly created object.

1. Let $F$ be the GeneratorFunction function object on which the new operator was applied.
2. Let argumentsList be the argumentsList argument of the [[Construct]] internal method that was invoked by the new operator.
3. Return the result of OrdinaryConstruct ($F$, argumentsList).

If GeneratorFunction is implemented as an ECMAScript function object, its [[Construct]] internal method will perform the above steps.

25.2.2 Properties of the GeneratorFunction Constructor

The GeneratorFunction constructor is a built-in Function object that inherits from the Function constructor. The value of the [[Prototype]] internal slot of the GeneratorFunction constructor is the intrinsic object %Function%.

The value of the [[Extensible]] internal slot of the GeneratorFunction constructor is true.

The GeneratorFunction constructor has the following properties:

25.2.2.1 GeneratorFunction.length

This is a data property with a value of 1. This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

25.2.2.2 GeneratorFunction.prototype

The initial value of GeneratorFunction.prototype is %Generator%, the standard built-in GeneratorFunction prototype.

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

25.2.2.3 GeneratorFunction[ @@create ] ( )

The @@create method of an object $F$ performs the following steps:

1. Let $F$ be the this value.
2. Let proto be the result of GetPrototypeFromConstructor($F$, %Generator%).
3. ReturnIfAbrupt(proto).
4. Let obj be the result of calling FunctionAllocate with argument proto, false, and "generator".
5. Return obj.

The value of the name property of this function is " [Symbol.create] ".

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

NOTE The GeneratorFunction @@create function passes false as the strict parameter to FunctionAllocate. This causes the allocated ECMAScript function object to have the internal methods of a non-strict function. The
GeneratorFunction constructor may reset the functions [[Strict]] internal slot to true. It is up to the implementation whether this also changes the internal methods.

25.2.3 Properties of the GeneratorFunction Prototype Object

The GeneratorFunction prototype object is an ordinary object. It is not a function object and does not have a [[Code]] internal slot or any other of the internal slots listed in Table 24 or Table 39. In addition to being the value of the prototype property of the %GeneratorFunction% intrinsic and is itself the %Generator% intrinsic.

The value of the [[Prototype]] internal slot of the GeneratorFunction prototype object is the %FunctionPrototype% intrinsic object. The initial value of the [[Extensible]] internal slot of the GeneratorFunction prototype object is true.

25.2.3.1 GeneratorFunction.prototype.constructor

The initial value of GeneratorFunction.prototype.constructor is the intrinsic object %GeneratorFunction%.

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

25.2.3.2 GeneratorFunction.prototype.prototype

The value of GeneratorFunction.prototype.prototype is the %GeneratorPrototype% intrinsic object.

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

25.2.3.3 GeneratorFunction.prototype @@toStringTag

The initial value of the @@toStringTag property is the string value “GeneratorFunction”.

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

25.2.3.4 GeneratorFunction.prototype @@create ( )

The @@create method of an object F performs the following steps:

1. Let F be the this value.
2. Let obj be the result of calling OrdinaryCreateFromConstructor(F, "%GeneratorPrototype%", ([GeneratorState], [GeneratorContext]) ).
3. Return obj.

The value of the name property of this function is “[Symbol.create]”.

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

25.2.4 GeneratorFunction Instances

Every GeneratorFunction instance is an ECMAScript function object and has the internal slots listed in Table 24. The value of the [[FunctionKind]] internal slot for all such instances is “generator”.

The GeneratorFunction instances have the following own properties:
25.2.4.1 length

The value of the length property is an integer that indicates the typical number of arguments expected by the GeneratorFunction. However, the language permits the function to be invoked with some other number of arguments. The behaviour of a GeneratorFunction 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]]: true }.

25.2.4.2 prototype

Whenever a GeneratorFunction instance is created another ordinary object is also created and is the initial value of the generator function’s prototype property. The value of the prototype property is used to initialise the [[Prototype]] internal slot of a newly created Generator object before the generator function object is invoked as a constructor for that newly created object.

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

NOTE Unlike function instances, the object that is the value of the a GeneratorFunction’s prototype property does not have a constructor property whose value is the GeneratorFunction instance.

25.3 Generator Objects

A Generator object is an instance of a generator function and conforms to both the Iterator and Iterable interfaces.

Generator instances directly inherit properties from the object that is the value of the prototype property of the Generator function that created the instance. Generator instances indirectly inherit properties from the Generator Prototype intrinsic, %GeneratorPrototype%.

25.3.1 Properties of Generator Prototype

The Generator prototype object is the %GeneratorPrototype% intrinsic. It is also the initial value of the prototype property of the %Generator% intrinsic (the GeneratorFunction.prototype).

The Generator prototype is an ordinary object. It is not a Generator instance and does not have a [[GeneratorState]] internal slot.

The value of the [[Prototype]] internal slot of the Generator prototype object is the intrinsic object %ObjectPrototype% (19.1.3). The initial value of the [[Extensible]] internal slot of the Function prototype object is true.

All Generator instances indirectly inherit properties of the Generator prototype object.

25.3.1.1 Generator.prototype.constructor

The initial value of Generator.prototype.constructor is the intrinsic object %Generator%.

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

25.3.1.2 Generator.prototype.next ( value )

The next method performs the following steps:

1. Let g be the this value.
2. Return the result of GeneratorResume(g, value).
25.3.1.3 Generator.prototype.throw (exception)

The throw method performs the following steps:

1. Let generator be the this value.
2. If Type(generator) is not Object, then throw a TypeError exception.
3. If generator does not have a [[GeneratorState]] internal slot, then throw a TypeError exception.
4. Let state be the value of generator's [[GeneratorState]] internal slot.
5. Assert: generator also has a [[GeneratorContext]] internal slot.
6. Let E be Completion{[[type]]: throw, [[value]]: exception, [[target]]: empty}.
7. If state is "completed", then return E.
8. If state is neither "suspendedStart" nor "suspendedYield", then throw a TypeError exception.
9. If state is "suspendedStart" then,
   a. Set generator's [[GeneratorState]] internal slot to "completed".
   b. Once a generator enters the "completed" state it never leaves it and its associated execution context is never resumed. Any execution state associated with generator can be discarded at this point.
   c. Return E.
10. Let genContext be value of generator's [[GeneratorContext]] internal slot.
11. Let methodContext be the running execution context.
13. Set generator's [[GeneratorState]] internal slot to "executing".
14. Push genContext onto the execution context stack; genContext is now the running execution context.
15. Resume the suspended evaluation of genContext using E as the result of the operation that suspended it. Let result be the value returned by the resumed companion.
16. Resume the suspended evaluation of genContext using E as the result of the operation that suspended it.
17. Assert: When we return here, genContext has already been removed from the execution context stack and methodContext is the currently running execution context.
18. Return result.

25.3.1.4 Generator.prototype [ @@iterator ] ( )

The following steps are taken:

1. Return the this value.

The value of the name property of this function is "[[Symbol.iterator]]".

25.3.1.5 Generator.prototype [ @@toStringTag ]

The initial value of the @@toStringTag property is the string value "Generator".

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

25.3.2 Properties of Generator Instances

Generator instances are initially created with the internal slots described in Table 39.

Table 39 — Internal Slots of Generator Instances

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[GeneratorState]]</td>
<td>The current execution state of the generator. The possible values are: undefined, &quot;suspendedStart&quot;, &quot;suspendedYield&quot;, &quot;executing&quot;, and &quot;completed&quot;.</td>
</tr>
<tr>
<td>[[GeneratorContext]]</td>
<td>The execution context that is used when executing the code of this function.</td>
</tr>
</tbody>
</table>
25.3.3 Generator Abstract Operations

25.3.3.1 GeneratorStart (generator, generatorBody)

1. Assert: The value of generator’s [[GeneratorState]] internal slot is undefined.
2. Let genContext be the running execution context.
3. Set the Generator component of genContext to generator.
4. Set the code evaluation state of genContext such that when evaluation is resumed for that execution context the following steps will be performed:
   a. Let result be the result of evaluating generatorBody.
   b. Assert: If we return here, the generator either threw an exception or performed either an implicit or explicit return.
   c. Remove genContext from the execution context stack and restore the execution context that is at the top of the execution context stack as the running execution context.
   d. Set generator’s [[GeneratorState]] internal slot to "completed".
   e. Once a generator enters the "completed" state it never leaves it and its associated execution context is never resumed. Any execution state associated with generator can be discarded at this point.
   f. ReturnIfAbrupt(result).
   g. Return CreateIterResultObject(result, true).
5. Set generator’s [[GeneratorContext]] internal slot to genContext.
6. Set generator’s [[GeneratorState]] internal slot to "suspendedStart",
7. Return NormalCompletion(generator).

25.3.3.2 GeneratorResume (generator, value)

The abstract operation GeneratorResume with arguments generator and value performs the following steps:

1. If Type(generator) is not Object, then throw a TypeError exception.
2. If generator does not have a [[GeneratorState]] internal slot, then throw a TypeError exception.
3. Let state be the value of generator’s [[GeneratorState]] internal slot.
4. Assert: generator also has a [[GeneratorContext]] internal slot.
5. If state is "completed", then return CreateIterResultObject(undefined, true).
6. If state is neither "suspendedStart" nor "suspendedYield", then throw a TypeError exception.
7. If state is "suspendedStart" and value is not undefined, then throw a TypeError exception.
8. Let genContext be value of generator’s [[GeneratorContext]] internal slot.
9. Let methodContext be the running execution context.
10. Suspending methodContext.
11. Push generator’s [[GeneratorState]] internal slot to "executing".
12. Push genContext onto the execution context stack; genContext is now the running execution context.
13. Resume the suspended evaluation of genContext using NormalCompletion(value) as the result of the operation that suspended it. Let result be the value returned by the resumed computation.
14. Assert: When we return here, genContext has already been removed from the execution context stack and methodContext is the currently running execution context.
15. Return result.

25.3.3.3 GeneratorYield ( iterNextObj )

The abstract operation GeneratorYield with argument iterNextObj performs the following steps:

1. Assert: iterNextObj is an Object that implemented the IteratorResult interface.
2. Let genContext be the running execution context.
3. Assert: genContext is the execution context of a generator.
4. Let generator be the value of the Generator component of genContext.
5. Set the value of generator’s [[GeneratorState]] internal slot to "suspendedYield".
6. Remove genContext from the execution context stack and restore the execution context that is at the top of the execution context stack as the running execution context.
7. Set the code evaluation state of `genContext` such that when evaluation is resumed with a Completion `resumptionValue` the following steps will be performed:
   a. Return `resumptionValue`.
   b. NOTE: This returns to the evaluation of the `YieldExpression` production that originally called this abstract operation.
8. Resume the suspended evaluation of `genContext` using `NormalCompletion(value)` as the result of the operation that suspended it. Let `result` be the value returned by the resumed computation.
9. Assert: When we return here, `genContext` has already been removed from the execution context stack and the currently running execution context is the context that most recently resumed execution of `generator`.
10. Return `NormalCompletion(iterNextObj)`.
11. NOTE: This returns to the evaluation of the operation that had most previously resumed evaluation of `genContext`.

26 Reflection

26.1 The Reflect Object

The `Reflect` object is a single ordinary object.

The value of the `[[Prototype]]` internal slot of the `Reflect` object is the standard built-in Object prototype object (19.1.3).

The `Reflect` object is not a function object. It does not have a `[[Construct]]` internal method; it is not possible to use the `Reflect` object as a constructor with the `new` operator. The `Reflect` object also does not have a `[[Call]]` internal method; it is not possible to invoke the `Reflect` object as a function.

26.1.1 Reflect.defineProperty(target, propertyKey, attributes)

When the `defineProperty` function is called with arguments `target`, `propertyKey`, and `attributes` the following steps are taken:

1. Let `obj` be `ToObject(target)`.
2. `ReturnIfAbrupt(obj)`.
3. Let `key` be `ToPropertyKey(propertyKey)`.
4. `ReturnIfAbrupt(key)`.
5. Let `desc` be the result of calling `ToPropertyDescriptor` with `attributes` as the argument.
6. `ReturnIfAbrupt(desc)`.
7. Return the result of calling the `[[DefineOwnProperty]]` internal method of `obj` with arguments `key` and `desc`.

26.1.2 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`.

26.1.3 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 `iterator` be the result of calling the `[[Enumerate]]` internal method of `obj`.
4. Return \( \text{iterator} \).

### 26.1.4 Reflect.get (target, propertyKey, receiver=target)

When the \( \text{get} \) function is called with arguments \( \text{target} \), \( \text{propertyKey} \), and \( \text{receiver} \) the following steps are taken:

1. Let \( \text{obj} \) be ToObject(\( \text{target} \)).
2. ReturnIfAbrupt(\( \text{obj} \)).
3. Let \( \text{key} \) be ToPropertyKey(\( \text{propertyKey} \)).
4. ReturnIfAbrupt(\( \text{key} \)).
5. If \( \text{receiver} \) is not present, then
   a. Let \( \text{receiver} \) be \( \text{target} \).
6. Return the result of calling the [[Get]] internal method of \( \text{obj} \) with arguments \( \text{key} \), and \( \text{receiver} \).

### 26.1.5 Reflect.getOwnPropertyDescriptor(target, propertyKey)

When the \( \text{getOwnPropertyDescriptor} \) function is called with arguments \( \text{target} \) and \( \text{propertyKey} \), the following steps are taken:

1. Let \( \text{obj} \) be ToObject(\( \text{target} \)).
2. ReturnIfAbrupt(\( \text{obj} \)).
3. Let \( \text{key} \) be ToPropertyKey(\( \text{propertyKey} \)).
4. ReturnIfAbrupt(\( \text{key} \)).
5. Let \( \text{desc} \) be the result of calling the [[GetOwnProperty]] internal method of \( \text{obj} \) with argument \( \text{key} \).
6. ReturnIfAbrupt(\( \text{desc} \)).
7. Return the result of calling FromPropertyDescriptor(\( \text{desc} \)).

### 26.1.6 Reflect.getPrototypeOf (target)

When the \( \text{getPrototypeOf} \) function is called with argument \( \text{target} \) the following steps are taken:

1. Let \( \text{obj} \) be ToObject(\( \text{target} \)).
2. ReturnIfAbrupt(\( \text{obj} \)).
3. Return the result of calling the [[GetPrototypeOf]] internal method of \( \text{obj} \).

### 26.1.7 Reflect.has (target, propertyKey)

When the \( \text{has} \) function is called with arguments \( \text{target} \) and \( \text{propertyKey} \), the following steps are taken:

1. Let \( \text{obj} \) be ToObject(\( \text{target} \)).
2. ReturnIfAbrupt(\( \text{obj} \)).
3. Let \( \text{key} \) be ToPropertyKey(\( \text{propertyKey} \)).
4. ReturnIfAbrupt(\( \text{key} \)).
5. Return the result of calling the [[HasProperty]] internal method of \( \text{obj} \) with argument \( \text{key} \).

### 26.1.8 Reflect.hasOwn (target, propertyKey)

When the \( \text{hasOwn} \) function is called with arguments \( \text{target} \) and \( \text{propertyKey} \), the following steps are taken:

1. Let \( \text{obj} \) be ToObject(\( \text{target} \)).
2. ReturnIfAbrupt(\( \text{obj} \)).
3. Let \( \text{key} \) be ToPropertyKey(\( \text{propertyKey} \)).
4. ReturnIfAbrupt(\( \text{key} \)).
5. Return the result of calling HasOwnProperty(\( \text{obj} \), \( \text{key} \)).

### 26.1.9 Reflect.isExtensible (target)

When the \( \text{isExtensible} \) function is called with argument \( \text{target} \) the following steps are taken:

1. Let \( \text{obj} \) be ToObject(\( \text{target} \)).
2. ReturnIfAbrupt(obj).
3. Return the result of calling the [[IsExtensible]] internal method of obj.

26.1.10 Reflect.ownKeys (target)

When the ownKeys 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 [[OwnPropertyKeys]] internal method of obj.

26.1.11 Reflect.preventExtensions (target)

When the preventExtensions 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 [[PreventExtensions]] internal method of obj.

26.1.12 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 [[Set]] internal method of obj with arguments key, V, and receiver.

26.1.13 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 [[SetPrototypeOf]] internal method of obj with argument proto.

26.2 Proxy Objects

26.2.1 The Proxy Factory Function

26.2.1.1 Proxy (target, handler)

The Proxy function is used to create an irrevocable Proxy object. When Proxy is called with arguments target and handler the following steps are taken:
1. Let p be ProxyCreate(target, handler).
2. Return p.
26.2.2 Properties of the Proxy Factory Function

26.2.2.1 Proxy.revocable (target, handler)

The `Proxy.revocable` function is used to create a revocable Proxy object. When `Proxy.revocable` is called with arguments `target` and `handler` the following steps are taken:

1. Let `p` be `ProxyCreate(target, handler)`.
2. ReturnIfAbrupt(`p`).
3. Let `revoker` be a new built-in function object as defined in 26.2.2.1.1.
4. Set the `[[RevokableProxy]]` internal slot of `revoker` to `p`.
5. Let `result` be the result of `ObjectCreate()`.
6. CreateDataProperty(`result`, "proxy", `p`).
7. CreateDataProperty(`result`, "revoke", `revoker`).
8. Return `result`.

26.2.2.1.1 Proxy Revocation Functions

A Proxy revocation function is an anonymous function that has the ability to invalidate a specific Proxy object. Each Proxy revocation function has a `[[RevokableProxy]]` internal slot.

When a Proxy revocation function, `F`, is called the following steps are taken:

1. Let `p` be the value of `F`'s `[[RevokableProxy]]` internal slot.
2. If `p` is `null`, then return `undefined`.
3. Set the value of `F`'s `[[RevokableProxy]]` internal slot to `null`.
4. Assert: `p` is a Proxy object.
5. Set the `[[ProxyTarget]]` internal slot of `p` to `null`.
6. Set the `[[ProxyHandler]]` internal slot of `p` to `null`.
7. Return `undefined`. 
Annex A
(informative)

Grammar Summary

TODO: The Grammars in the Annex have not yet been updated for ES6. For now, see the grammars in the main body of the specification.

A.1 Lexical Grammar

SourceCharacter ::
    any Unicode code unit
    See clause 6

InputElementDiv ::
    WhiteSpace
    LineTerminator
    Comment
    Token
    DivPunctuator
    See clause 7

InputElementRegExp ::
    WhiteSpace
    LineTerminator
    Comment
    Token
    RegularExpressionLiteral
    See clause 7

WhiteSpace ::
    <TAB>
    <VT>
    <FF>
    <SP>
    <NBSP>
    <BOM>
    <USP>
    See 7.2

LineTerminator ::
    <LF>
    <CR>
    <LS>
    <PS>
    See 7.3
LineTerminatorSequence ::
    <LF>
    <CR> [lookahead ≠ <LF> ]
    <LS>
    <PS>
    <CR> <LF>

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

Token ::
    IdentifierName
    Punctuator
    NumericLiteral
    StringLiteral

Identifier ::
    IdentifierName but not ReservedWord

IdentifierName ::
    IdentifierStart
    IdentifierName IdentifierPart
IdentifierStart ::
  UnicodeLetter
  $  
  UnicodeEscapeSequence

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

UnicodeLetter ::

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

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

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

ReservedWord ::
  Keyword
  FutureReservedWord
  NullLiteral
  BooleanLiteral

Keyword ::
  one of
  break of 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
  class enum extends super
  const export import

The following tokens are also considered to be FutureReservedWords when parsing strict mode code (see 10.2.1).

  implements let private public
  interface package protected static
yield

Punctuator :: one of

{}()[]See 7.7
- ; , < > <=
>= == != === !==
+ - * % ++ --
<< >> >>> & | ^
! ~ && || ? :
= += -= *= %= <<=
>>= >>>= @= |= ^=

DivPunctuator :: one of

/ /= See 7.7

Literal ::

NullLiteral
BooleanLiteral
NumericLiteral
StringLiteral
RegularExpressionLiteral

NullLiteral ::

null See 7.8.1

BooleanLiteral ::

truetrue
falsefalse

NumericLiteral ::

DecimalLiteral
HexIntegerLiteral

DecimalLiteral ::

DecimalIntegerLiteral . DecimalDigitsopt ExponentPartopt . DecimalDigits ExponentPartopt

DecimalIntegerLiteral ExponentPartopt

DecimalIntegerLiteral ::

0
NonZeroDigit DecimalDigitsopt

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

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

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 is DecimalDigit]
   HexEscapeSequence
   UnicodeEscapeSequence

CharacterEscapeSequence ::
   SingleEscapeCharacter
   NonEscapeCharacter
SingleEscapeCharacter :: one of ' " \ b f n r t v See 7.8.4

NonEscapeCharacter :: SourceCharacter but not one of EscapeCharacter or LineTerminator See 7.8.4

EscapeCharacter :: SingleEscapeCharacter DecimalDigit x u See 7.8.4

HexEscapeSequence :: x HexDigit HexDigit See 7.8.4

UnicodeEscapeSequence :: u HexDigit HexDigit HexDigit HexDigit See 7.8.4

RegularExpressionLiteral :: / RegularExpressionBody / RegularExpressionFlags See 7.8.5

RegularExpressionBody :: RegularExpressionFirstChar RegularExpressionChars See 7.8.5

RegularExpressionChars :: [empty] RegularExpressionChars RegularExpressionChar See 7.8.5

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

RegularExpressionChar :: RegularExpressionNonTerminator but not \ or / or [ ]

RegularExpressionBackslashSequence :: \ RegularExpressionNonTerminator See 7.8.5

RegularExpressionNonTerminator :: SourceCharacter but not LineTerminator See 7.8.5

RegularExpressionClass :: [ RegularExpressionClassChars ] See 7.8.5

RegularExpressionClassChars :: [empty]

RegularExpressionClassChar
RegularExpressionClassChar :: RegularExpressionNonTerminator but not } or \\
RegularExpressionBackslashSequence

RegularExpressionFlags :: [empty]
RegularExpressionFlags IdentifierPart

A.2 Number Conversions

StringNumericLiteral :: StringWhiteSpaceopt
StringWhiteSpaceopt StrNumericLiteral StringWhiteSpaceopt

StringWhiteSpace :: StringWhiteSpaceChar StringWhiteSpaceopt

StringWhiteSpaceChar :: WhiteSpace
LineTerminator

StrNumericLiteral :: StrDecimalLiteral
HexIntegerLiteral

StrDecimalLiteral :: StrUnsignedDecimalLiteral
+ StrUnsignedDecimalLiteral
- StrUnsignedDecimalLiteral

StrUnsignedDecimalLiteral :: Infinity
DecimalDigits . DecimalDigits opt ExponentPart opt
. DecimalDigits ExponentPart opt
DecimalDigits ExponentPart opt

DecimalDigits :: DecDigit
DecDigit DecDigit

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

ExponentPart :: ExponentIndicator SignedInteger

ExponentIndicator :: one of e E

See 9.1.3.1
SignedInteger ::= See 9.1.3.1
    DecimalDigits
    + DecimalDigits
    - DecimalDigits

HexIntegerLiteral ::= See 9.1.3.1
    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

A.3 Expressions

PrimaryExpression : See 11.1
    this
    Identifier
    Literal
    ArrayLiteral
    ObjectLiteral
    ( Expression )

ArrayLiteral : See 11.1.4
    [ Elisionopt ]
    [ ElementList ]
    [ ElementList , Elisionopt ]

ElementList : See 11.1.4
    Elisionopt AssignmentExpression
    ElementList , Elisionopt 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 }
PropertyName:
  IdentifierName
  StringLiteral
  NumericLiteral

PropertySetParameterList:
  Identifier

MemberExpression:
  PrimaryExpression
  FunctionExpression
  MemberExpression [ Expression ]
  MemberExpression . IdentifierName
  new MemberExpression Arguments

NewExpression:
  MemberExpression
  new NewExpression

CallExpression:
  MemberExpression Arguments
  CallExpression Arguments
  CallExpression [ Expression ]
  CallExpression . IdentifierName

Arguments:
  ()
  ( ArgumentList )

ArgumentList:
  AssignmentExpression
  ArgumentList , AssignmentExpression

LeftHandSideExpression:
  NewExpression
  CallExpression

PostfixExpression:
  LeftHandSideExpression
  LeftHandSideExpression [ No LineTerminator here ] ++
  LeftHandSideExpression [ No LineTerminator here ] --
UnaryExpression:
  PostfixExpression
  delete UnaryExpression
  typeof UnaryExpression
  ++ UnaryExpression
  -- UnaryExpression
  + UnaryExpression
  - UnaryExpression
  unary UnaryExpression

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

AdditiveExpression:
  MultiplicativeExpression
  AdditiveExpression + MultiplicativeExpression
  AdditiveExpression - MultiplicativeExpression

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

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

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

BitwiseANDExpression:
  EqualityExpression
  BitwiseANDExpression & EqualityExpression

BitwiseXORExpression:
  BitwiseANDExpression
  BitwiseXORExpression ^ BitwiseANDExpression
BitwiseORExpression:
  BitwiseORExpression
  BitwiseORExpression | BitwiseORExpression

LogicalANDExpression:
  BitwiseORExpression
  LogicalANDExpression && BitwiseORExpression

LogicalORExpression:
  LogicalANDExpression
  LogicalORExpression || LogicalANDExpression

ConditionalExpression:
  LogicalORExpression
  LogicalORExpression ? AssignmentExpression : AssignmentExpression

AssignmentExpression:
  ConditionalExpression
  LeftHandSideExpression = AssignmentExpression
  LeftHandSideExpression AssignmentOperator AssignmentExpression

AssignmentOperator: one of
  *= /= %= += -= <<= >>= >>>= &= ^= |=

Expression:
  AssignmentExpression
  Expression , AssignmentExpression

A.4 Statements

Statement:
  Block
  VariableStatement
  EmptyStatement
  ExpressionStatement
  IfStatement
  IterationStatement
  ContinueStatement
  BreakStatement
  ReturnStatement
  WithStatement
  LabelledStatement
  SwitchStatement
  ThrowStatement
  TryStatement
  DebuggerStatement

See 11.10
See 11.11
See 11.11
See 11.12
See 11.13
See 11.13
See 11.14
See clause 12
Block:
    { StatementListopt }

StatementList:
    Statement
    StatementList Statement

VariableStatement:
    var VariableDeclarationList ;

VariableDeclarationList:
    VariableDeclaration
    VariableDeclarationList , VariableDeclaration

VariableDeclaration:
    Identifier Initialiseropt

Initialiser:
    = AssignmentExpression

EmptyStatement:
    ;

ExpressionStatement:
    [lookahead ≠ {i, function}] Expression ;

IfStatement:
    if ( Expression ) Statement elseStatement
    if ( Expression ) Statement

IterationStatement:
    do Statement while ( Expression ) ;
    while ( Expression ) Statement
    for ( Expressionopt ; Expressionopt ; Expressionopt ) Statement
    for ( var VariableDeclarationList ; Expressionopt ; Expressionopt ) Statement
    for ( var VariableDeclaration in Expression ) Statement

ContinueStatement:
    continue ;
    continue [to LineTerminator here] Identifier ;

BreakStatement:
    break ;
    break [to LineTerminator here] Identifier ;

ReturnStatement:
    return ;
    return [to LineTerminator here] Expression ;
WithStatement:
  \textit{with} \ (\textit{Expression} \ ) \textit{Statement} \\
  \textit{See 12.10}

SwitchStatement:
  \textit{switch} \ (\textit{Expression}) \textit{CaseBlock} \\
  \textit{See 12.11}

CaseBlock:
  \{
    \textit{CaseClauses}_{\text{opt}} \\
    \textit{CaseClauses}_{\text{opt}} \textit{DefaultClause} \textit{CaseClauses}_{\text{opt}}
  \} \\
  \textit{See 12.11}

CaseClauses:
  \textit{CaseClause} \textit{CaseClauses} \textit{CaseClause} \\
  \textit{See 12.11}

CaseClause:
  \textit{case} \textit{Expression} : \textit{StatementList}_{\text{opt}} \\
  \textit{See 12.11}

DefaultClause:
  \textit{default} : \textit{StatementList}_{\text{opt}} \\
  \textit{See 12.11}

LabelledStatement:
  \textit{Identifier} : \textit{Statement} \\
  \textit{See 12.12}

ThrowStatement:
  \textit{throw} \ [\textit{no LineTerminator here}] \textit{Expression} ; \\
  \textit{See 12.13}

TryStatement:
  \textit{try} \textit{Block} \textit{Catch} \textit{try} \textit{Block} \textit{Finally} \textit{try} \textit{Block} \textit{Catch} \textit{Finally} \\
  \textit{See 12.14}

Catch:
  \textit{catch} \ (\textit{Identifier}) \textit{Block} \\
  \textit{See 12.14}

Finally:
  \textit{finally} \textit{Block} \\
  \textit{See 12.14}

DebuggerStatement:
  \textit{debugger} ; \\
  \textit{See 12.15}

\textbf{A.5 Functions and Scripts}

FunctionDeclaration:
  \textit{function} \textit{Identifier} \ (\textit{FormalParameterList}_{\text{opt}}) \ \{ \textit{FunctionBody} \} \\
  \textit{See clause 13}
FunctionExpression:
  function Identifieropt ( FormalParameterListopt ) { FunctionBody }

FormalParameterList:
  Identifier
  FormalParameterList , Identifier

FunctionBody:
  SourceElementsopt

Program:
  SourceElementsopt

SourceElements:
  SourceElement
  SourceElements SourceElement

SourceElement:
  Statement
  FunctionDeclaration

A.6 Universal Resource Identifier Character Classes

uri ::=
  uriCharactersopt

uriCharacters ::=
  uriCharacter uriCharactersopt

uriCharacter ::=
  uriReserved
  uriUnescaped
  uriEscaped

uriReserved ::=
  one of
  ; / ? : @ & = + $ ,

uriUnescaped ::=
  uriAlpha
  DecimalDigit
  uriMark

uriEscaped ::=
  % HexDigit HexDigit

uriAlpha ::=
  one of
  a b c d e f g h i j k l m n o p q r s t u v w x y z
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
A.7 Regular Expressions

Pattern ::
  Disjunction

Disjunction ::
  Alternative
  Alternative | Disjunction

Alternative ::
  [empty]
  Alternative Term

Term ::
  Assertion
  Atom
  Atom Quantifier

Assertion ::
  ^
  $ \
  \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
  \ ControlLetter
  HexEscapeSequence
  UnicodeEscapeSequence
  IdentityEscape

ControlEscape :: one of
  fnrtv

ControlLetter :: one of
  abcd efghijklmnopqrstuvwxyz
  ABCDEFGHIJKLMNOPQRSTUVWXYZ

IdentityEscape ::
  SourceCharacter but not IdentifierPart
  <ZWJ>
  <ZWNJ>

DecimalEscape ::
  DecimalIntegerLiteral [lookahead a DecimalDigit]

CharacterClassEscape :: one of
d D s S w W

CharacterClass ::
  [ [lookahead a \^] ] 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
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 11.8.3 is extended as follows except that this extension is not allowed for strict mode code:

Syntax

\[
\text{NumericLiteral} ::
\quad \text{DecimalLiteral}
\quad \text{BinaryIntegerLiteral}
\quad \text{OctalIntegerLiteral}
\quad \text{HexIntegerLiteral}
\quad \text{LegacyOctalIntegerLiteral}
\]

\[
\text{LegacyOctalIntegerLiteral} ::
\quad 0 \text{OctalDigit}
\quad \text{LegacyOctalIntegerLiteral} \text{OctalDigit}
\]

Static Semantics

- The MV of \text{LegacyOctalIntegerLiteral} :: 0 \text{OctalDigit} is the MV of \text{OctalDigit}.
- The MV of \text{LegacyOctalIntegerLiteral} :: \text{LegacyOctalIntegerLiteral} \text{OctalDigit} is (the MV of \text{LegacyOctalIntegerLiteral} times 8) plus the MV of \text{OctalDigit}.

B.1.2 String Literals

The syntax and semantics of 11.8.4 is extended as follows except that this extension is not allowed for strict mode code:

Syntax

\[
\text{EscapeSequence} ::
\quad \text{CharacterEscapeSequence}
\quad \text{OctalEscapeSequence}
\quad \text{HexEscapeSequence}
\quad \text{UnicodeEscapeSequence}
\]

\[
\text{OctalEscapeSequence} ::
\quad \text{OctalDigit} \text{OctalDigit}
\quad \text{ZeroToThree OctalDigit} \text{OctalDigit}
\quad \text{FourToSeven OctalDigit}
\quad \text{ZeroToThree OctalDigit OctalDigit}
\]

\[
\text{ZeroToThree} :: \text{one of}
\quad 0 1 2 3
\]
Static Semantics

- The CV of `EscapeSequence` :: `OctalEscapeSequence` is the CV of the `OctalEscapeSequence`.
- The CV of `OctalEscapeSequence` :: `OctalDigit` is the character whose code unit value is the MV of the `OctalDigit`.
- The CV of `OctalEscapeSequence` :: `ZeroToThree OctalDigit` is the character whose code unit value is \((8 \text{ times the MV of the ZeroToThree}) \text{ plus the MV of the OctalDigit.}\)
- The CV of `OctalEscapeSequence` :: `FourToSeven OctalDigit` is the character whose code unit value is \((8 \times \text{times the MV of the FourToSeven}) \text{ plus the MV of the OctalDigit.}\)
- The CV of `OctalEscapeSequence` :: `ZeroToThree OctalDigit OctalDigit` is the character whose code unit value is \((84 \text{ (that is, } 8^2 \text{) times the MV of the ZeroToThree}) \text{ plus (8 times the MV of the first OctalDigit) plus the MV of the second OctalDigit.}\)
- 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.1.3 HTML-like Comments

TODO See [http://javascript.spec.whatwg.org/#comment-syntax](http://javascript.spec.whatwg.org/#comment-syntax)

B.1.4 Regular Expressions Patterns

The syntax of 21.2.1 is extended as modified and extended as follows. These changes introduce ambiguities that are broken by the ordering or grammar productions and by contextual information. The following grammar is used, with each alternative considered only if previous production alternatives do not match.

```
Syntax

Term: [±0] ExtendedTerm
    [±0] Assertion
    [±0] Atom
    [±0] Atom Quantifier

ExtendedTerm ::
    Assertion
    AtomNoBrace Quantifier
    Atom
    QuantifiableAssertion Quantifier
```
AtomNoBrace ::
   PatternCharacterNoBrace
   \ AtomEscape
   CharacterClass
   ( Disjunction )
   ( ? : Disjunction )

Atom_0 ::
   PatternCharacter
   \ AtomEscape[\n]
   CharacterClass
   ( Disjunction )
   ( ? : Disjunction )

PatternCharacterNoBrace ::
   SourceCharacter but not one of
   ^ $ \ . * + ? ( ) [ ] { } |

PatternCharacter ::
   SourceCharacter but not one of
   ^ $ \ . * + ? ( ) [ ] { } |

QuantifiableAssertion ::
   ( ? = Disjunction )
   ( ? ! Disjunction )

Assertion_0 ::
   ^ $ \ b \ B
   ( ? = Disjunction )
   ( ? ! Disjunction )
   ~ U QuantifiableAssertion

AtomEscape_0 ::
   ( ?) DecimalEscape
   ( ?) DecimalEscape but only if the integer value of DecimalEscape is <= NCapturingParens
   ( ?) CharacterEscape
   ( ?) CharacterClassEscape
   ( ?) CharacterEscape
   ( ?) CharacterEscape

CharacterEscape_0 ::
   ControlEscape
   ControlLetter
   HexEscapeSequence
   RegExpUnicodeEscapeSequence[\n]
   ( ?) OctalEscapeSequence
   IdentityEscape[\n]

IdentityEscape_0 ::
   ( ?) SyntaxCharacter
   ( ?) SourceCharacter but not c
   ( ?) <ZWJ>
   ( ?) <ZWNJ>
NonemptyClassRanges ::
ClassAtom
NonemptyClassRangesNoDash
[+] ClassAtom - ClassAtom NoDash ClassRanges
[-] ClassAtomInRange - ClassAtomInRange ClassRanges

NonemptyClassRangesNoDash ::
ClassAtom
ClassAtomNoDash NonemptyClassRangesNoDash
[+] ClassAtomNoDash - ClassAtomNoDash ClassRanges
[-] ClassAtomNoDashInRange - ClassAtomNoDashInRange ClassRanges

ClassAtom ::
- ClassAtomNoDash

ClassAtomNoDash ::
SourceCharacter but not one of \ or ] or -
\ ClassEscape

ClassAtomInRange ::
- ClassAtomNoDashInRange

ClassAtomNoDashInRange ::
SourceCharacter but not one of \ or ] or -
\ ClassEscape but only if ClassEscape evaluates to a CharSet with exactly one character
\ IdentityEscape

ClassEscape ::
[+] DecimalEscape
[-] DecimalEscape but only if the integer value of DecimalEscape is <= NCapturingParens
[+] CharacterEscape
[+] CharacterClassEscape
[-] CharacterEscape

Pattern Semantics
The semantics of 21.2.2 is extended as follows:

Within 21.2.2.5 reference to "Atom :: ( Disjunction )" are to be interpreted as meaning "Atom :: ( Disjunction ) or AtomNoBrace :: ( Disjunction )".

Term (21.2.2.5) includes the following additional evaluation rule:

The production Term :: QuantifiableAssertion Quantifier evaluates the same as the production Term :: Atom Quantifier but with QuantifiableAssertion substituted for Atom.

Atom (21.2.2.8) evaluation rules for the Atom productions except for Atom :: PatternCharacter are also used for the AtomNoBrace productions, but with AtomNoBrace substituted for Atom. The following evaluation rule is also added:

The production AtomNoBrace :: PatternCharacterNoBrace evaluates as follows:
1. Let \( ch \) be the character represented by \( \text{PatternCharacterNoBrace} \).
2. Let \( A \) be a one-element CharSet containing the character \( ch \).
3. Call \( \text{CharacterSetMatcher}(A, \text{false}) \) and return its Matcher result.

CharacterEscape (21.2.2.10) includes the following additional evaluation rule:

The production \( \text{CharacterEscape} :: \text{OctalEscapeSequence} \) evaluates by evaluating the CV of the \( \text{OctalEscapeSequence} \) (see B.1.2) and returning its character result.

ClassAtom (21.2.2.17) includes the following additional evaluation rules:

The production \( \text{ClassAtomInRange} :: - \) evaluates by returning the CharSet containing the one character 

The production \( \text{ClassAtomInRange} :: \text{ClassAtomNoDashInRange} \) evaluates by evaluating \( \text{ClassAtomNoDashInRange} \) to obtain a CharSet and returning that CharSet.

ClassAtomNoDash (21.2.2.18) includes the following additional evaluation rules:

The production \( \text{ClassAtomNoDashInRange} :: \text{SourceCharacter} \) but not one of \( \text{\ or \} \) or \(-\) evaluates by returning a one-element CharSet containing the character represented by \( \text{SourceCharacter} \).

The production \( \text{ClassAtomNoDashInRange} :: \text{\ ClassEscape} \) but only if..., evaluates by evaluating \( \text{ClassEscape} \) to obtain a CharSet and returning that CharSet.

The production \( \text{ClassAtomNoDashInRange} :: \text{IdentityEscape} \) evaluates by returning the character represented by \( \text{IdentityEscape} \).

**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 \( \text{escape} \) 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 \( \text{%xx} \) is used. For those characters being replaced whose code unit value is greater than \( 0xFF \), a four-digit escape sequence of the form \( \text{%uxxxx} \) is used.

When the \( \text{escape} \) function is called with one argument \( \text{string} \), the following steps are taken:

1. Let \( \text{string} \) be ToString(\( \text{string} \)).
2. ReturnIfAbrupt(\( \text{string} \)).
3. Let \( \text{length} \) be the number of code units in \( \text{string} \).
4. Let \( R \) be the empty string.
5. Let \( k \) be \( 0 \).
6. Repeat, while \( k < \text{length} \),
   a. Let \( \text{char} \) be the code unit (represented as a 16-bit unsigned integer) at position \( k \) within \( \text{string} \).
b. If char is the code point of one of the 69 nonblank characters
   "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789@*_+-./",
   then
   i. Let S be a String containing the single character char.
   c. Else if char > 256,
      i. Let S be a String containing six characters "%uwxtyz" where wxyz are four hexadecimal
digits encoding the value of char.
   d. Else, char < 256
      i. Let S be a String containing three characters "%xy" where xy are two hexadecimal digits
encoding the value of char.
   e. Let R be a new String value computed by concatenating the previous value of R and S.
   f. Increase k by 1.

7. Return R.

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. Let string be ToString(string).
2. ReturnIfAbrupt(string).
3. Let length be the number of code units in string.
4. Let R be the empty String.
5. Let k be 0.
6. Repeat, while k ≠ length
   a. Let c be the code unit at position k within string.
   b. If c is %,
      i. If k ≤ length-6 and the code unit at position k+1 within string is u and the four code units at
positions k+2, k+3, k+4, and k+5 within string are all hexadecimal digits, then
         1. Let c be the code unit whose value is the integer represented by the four
hexadecimal digits at positions k+2, k+3, k+4, and k+5 within string.
         2. Increase k by 5.
      ii. Else if k ≤ length-3 and the two code units at positions k+1 and k+2 within string are both
hexadecimal digits, then
         1. Let c be the code unit whose value is the integer represented by two zeroes plus the
two hexadecimal digits at positions k+1 and k+2 within string.
         2. Increase k by 2.
   b. Let R be a new String value computed by concatenating the previous value of R and c.
   c. Increase k by 1.
5. Return R.

B.2.2 Additional Properties of the Object.prototype Object

B.2.2.1 Object.prototype.__proto__
Object.prototype.__proto__ is an accessor property with attributes {[Enumerable]: false, [Configurable]:
true}. The [[Get]] and [[Set]] attributes are defined as follows
B.2.2.1.1 get Object.prototype.__proto__

The value of the [[Get]] attribute is a built-in function that requires no arguments. It performs the following steps:

1. Let O be the result of calling ToObject passing the this value as the argument.
2. ReturnIfAbrupt(O).
3. Return the result of calling the [[GetPrototypeOf]] internal method of O.

B.2.2.1.2 set Object.prototype.__proto__

The value of the [[Set]] attribute is a built-in function that takes an argument proto. It performs the following steps:

1. Let O be CheckObjectCoercible(this value).
2. ReturnIfAbrupt(O).
3. If Type(proto) is neither Object nor Null, then return proto.
4. If Type(O) is not Object, then return proto.
5. Let status be the result of calling the [[SetPrototypeOf]] internal method of O with argument proto.
6. ReturnIfAbrupt(status).
7. If status is false, then throw a TypeError exception.
8. Return proto.

B.2.3 Additional Properties of the String.prototype Object

B.2.3.1 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. Let O be CheckObjectCoercible(this value).
2. Let S be ToString(O).
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.2.3.2 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. Let str be CheckObjectCoercible(string).
2. Let S be ToString(str).
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 "&quot;".
   d. Let p1 be the string value that is the concatenation of the following string values:
      1. p1
      2. a single space code unit 0x0020
      3. attribute
      4. "="
      5. escapedV
      6. ","
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 and S.
8. Let p4 be the string value that is the concatenation of p2, ">", tag, and ">".

B.2.3.3 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.2.3.4 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.2.3.5 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.2.3.6 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.2.3.7 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.2.3.8 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.2.3.9 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.2.3.10 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.2.3.11 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.2.3.12 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.2.3.13 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.2.3.14 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 \( S, \text{"sup"}, \text{""} \) and \( \text{""} \).

### B.2.4 Additional Properties of the Date.prototype Object

#### B.2.4.1 Date.prototype.getYear ( )

The `getYear` 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. Return `IfAbrupt(\( t \))`.
3. If \( t \) is \( \text{NaN} \), return \( \text{NaN} \).
4. Return YearFromTime(\( \text{LocalTime}(t) \) − 1900).

#### B.2.4.2 Date.prototype.setYear (year)

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 \( \text{LocalTime}(\text{this time value}) \); but if this time value is \( \text{NaN} \), let \( t \) be +0.
2. Let \( y \) be \( \text{ToNumber}(\text{year}) \).
3. If \( y \) is not \( \text{NaN} \) and 0 ≤ \( \text{ToInteger}(y) \) ≤ 99 then let \( yyyy \) be \( \text{ToInteger}(y) + 1900 \). Otherwise, let \( yyyy \) be \( y \).
4. Let \( d \) be MakeDay(\( yyyy \), \( \text{MonthFromTime}(t) \), \( \text{DateFromTime}(t) \)).
5. Let \( date \) be \( \text{UTC}(\text{MakeDate}(d, \text{TimeWithinDay}(t))) \). Set the \([\text{DateValue}]\) internal slot of this Date object to \( \text{TimeClip}(\text{date}) \). Return the value of the \([\text{DateValue}]\) internal slot of this Date object.

#### B.2.4.3 Date.prototype.toGMTString ( )

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.2.5 Additional Properties of the RegExp.prototype Object

#### B.2.5.1 RegExp.prototype.compile (pattern, flags )

When the `compile` method is called with arguments `pattern` and `flags`, the following steps are taken:

1. Let \( O \) be the `this` value.
2. If `Type(O)` is not `Object` or `Type(O)` is `Object` and \( O \) does not have a \([\text{RegExpMatcher}]\) internal slot, then
   a. Throw a `TypeError` exception.
3. Let `extensible` be the result of calling the \([\text{IsExtensible}]\) internal method of \( O \).
4. If `extensible` is `false`, then throw a `TypeError` exception.
5. If `Type(pattern)` is `Object` and `pattern` has a \([\text{RegExpMatcher}]\) internal slot, then
   a. If the value of `pattern`’s \([\text{RegExpMatcher}]\) internal slot is `undefined`, then throw a `TypeError` exception.
   b. If `flags` is not `undefined`, then throw a `TypeError` exception.
   c. Let \( P \) be the value of `pattern`’s \([\text{OriginalSource}]\) internal slot.
   d. Let \( P \) be the value of `pattern`’s \([\text{OriginalFlags}]\) internal slot.
6. Else,
   a. let \( P \) be pattern.
   b. let \( F \) be flags.
7. Return the result of the abstract operation RegExpInitialise with arguments \( O \), \( P \), and \( F \).

NOTE The `compile` method completely reinitialises the this object RegExp with a new pattern and flags. An implementation may interpret use of this method as an assertion that the resulting RegExp object will be used multiple times and hence is a candidate for extra optimization.

B.3 Other Additional Features

B.3.1 __proto__ Property Names in Object Initialisers

In 12.1.5 the PropertyDefinitionEvaluation algorithm for the production `PropertyDefinition : PropertyName : AssignmentExpression` is replaced with the following:

```
PropertyDefinition : PropertyName : AssignmentExpression
```

1. Let `propKey` be the result of evaluating `PropertyName`.
2. ReturnIfAbrupt(`propKey`).
3. Let `exprValueRef` be the result of evaluating `AssignmentExpression`.
4. Let `propValue` be GetValue(`exprValueRef`).
5. ReturnIfAbrupt(`propValue`).
6. If `propKey` is the string value "__proto__" and if IsComputedPropertyKey(`propKey`) is false, then
   a. If Type(`propValue`) is either Object or Null, then
      i. Return the result of calling the `[[SetPrototypeOf]]` internal method of object with argument `propValue`.
   b. Return NormalCompletion(empty).
7. Return the Record{[[key]]: `propKey`, [[descriptor]]: PropertyDescriptor{[[Value]]: `propValue`, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}}.

B.3.2 Web Legacy Compatibility for Block-Level Function Declarations

Prior to the Sixth Edition, the ECMAScript specification did not define the occurrence of a `FunctionDeclaration` as an element of a `Block` statement's `StatementList`. However, support for that form of `FunctionDeclaration` was an allowable extension and most browser-hosted ECMAScript implementations permitted them. However, the semantics of such declarations differ among those implementations. Because of these semantic differences, existing web ECMAScript code that uses Block-level function declarations is only portable among browser implementations if the usage only depends upon the semantic intersection of all of the browser implementations for such declarations. The following are the use cases that fall within that intersection semantics:

1. A function is declared and only referenced within a single block
   • A function declaration with the name \( f \) is declared exactly once within the function code of an enclosing function \( g \) and that declaration is nested within a `Block`.
   • No other declaration of \( f \) that is not a `var` declaration occurs within the function code of \( g \).
   • All references to \( f \) occur within the `StatementList` of the `Block` containing the declaration of \( f \).

2. A function is declared and possibly used within a single `Block` but also referenced by an inner function definition that is not contained within that same `Block`.
   • A function declaration with the name \( f \) is declared exactly once within the function code of an enclosing function \( g \) and that declaration is nested within a `Block`.
   • No other declaration of \( f \) that is not a `var` declaration occurs within the function code of \( g \).
   • References to \( f \) may occur within the `StatementList` of the `Block` containing the declaration of \( f \).
   • References to \( f \) occur within the function code of \( g \) that lexically follows the `Block` containing the declaration of \( f \).
3. A function is declared and possibly used within a single block but also referenced within subsequent blocks.
   • A function declaration with the name \( f \) is declared exactly once within the function code of an enclosing function \( g \) and that declaration is nested within a Block.
   • No other declaration of \( f \) that is not a \texttt{var} declaration occurs within the function code of \( g \).
   • References to \( f \) may occur within the \texttt{StatementList} of the Block containing the declaration of \( f \).
   • References to \( f \) occur within another function \( h \) that is nested within \( g \) and no other declaration of \( f \) shadows the references to \( f \) from within \( h \).
   • All invocations of \( h \) occur after the declaration of \( f \) has been evaluated.

The first use case is interoperable with the inclusion of Block level function declarations in the sixth edition. Any pre-existing ECMAScript code that employees that use case will operate using the Block level function declarations semantics defined by clauses 10 and 13 of this specification.

Sixth edition interoperability for the second and third use cases requires the following extensions to the clauses 10 and 14 semantics. These extensions are applied to a non-strict mode functions \( g \) if the above pre-conditions of use cases 2 and 3 above exist at the time of static semantic analysis of \( g \). However, the last pre-condition of use case 3 is not included in this determination and the determination is not applied to any function declaration that is nested within syntactic constructs that are specified in the Fifth edition of this specification.

1. Let \( B \) be environment record for the construct within \( g \) that introduces a new environment contour and which most closely encloses the declaration of \( f \), all function code references to \( f \), and the definitions of all nested functions that contain unshadowed references to \( f \). This syntactic construct may be the definition of \( g \) itself, in which case \( B \) is the function environment record for \( g \).
2. As part of the instantiation of \( B \), its CreateMutableBinding concrete method is called with arguments “\( f \)” (the string name of the function) and \texttt{false}. This creates an uninitialised binding for the name \( f \). Any reference that resolves to that binding prior to step 3 below will throw a \texttt{ReferenceError} exception.
3. When the InitialiseBinding concrete method is used to initialise the binding for the function declaration \( f \) also invoke InitialiseBind on \( B \) using the same arguments.

If an ECMAScript implication has a mechanism that produces diagnostic warning messages, a warning should be produced for each function \( g \) for which the above steps are performed.
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. (11.6.2.2).
- A conforming implementation, when processing strict mode code, may not extend the syntax of NumericLiteral (11.8.3) to include LegacyOctalIntegerLiteral as described in B.1.1.
- A conforming implementation, when processing strict mode code (see 10.2.1), may not extend the syntax of EscapeSequence to include LegacyOctalEscapeSequence 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 (6.2.3.2). 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 slot has the value false. In these cases a TypeError exception is thrown (12.13).
- The identifier eval or arguments may not appear as the LeftHandSideExpression of an Assignment operator (12.13) or of a PostfixExpression (12.13) or as the UnaryExpression operated upon by a Prefix Increment (12.4.7) or a Prefix Decrement (12.4.8) operator.
- Arguments objects for strict mode functions define non-configurable accessor properties named "caller" and "callee" which throw a TypeError exception on access (9.2.8).
- Arguments objects for strict mode functions do not dynamically share their array indexed property values with the corresponding formal parameter bindings of their functions. (9.4.4.1).
- 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. (9.4.4.1).
- It is a SyntaxError if strict mode code contains an ObjectLiteral with more than one definition of any data property (12.1.5).
- It is a SyntaxError if the Identifier "eval" or the Identifier "arguments" occurs as the Identifier in a PropertySetParameterList or a PropertyDefinition that is contained in strict code or if its FunctionBody is strict code (12.1.5).
- 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 (18.2.1).
- 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 (8.3.2, 12.1.1, 19.2.3.3, 0). 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(12.4.4).
• When a `delete` operator occurs within strict mode code, a `TypeError` is thrown if the property to be deleted has the attribute `{ [[Configurable]]: false }` (12.4.4).

• It is a `SyntaxError` if a `VariableDeclaration` occurs within strict code and its `Identifier` is `eval` or `arguments` (13.2.2).

• Strict mode code may not include a `WithStatement`. The occurrence of a `WithStatement` in such a context is a `SyntaxError` (13.10).

• 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` (13.14)

• It is a `SyntaxError` if the identifier `eval` or `arguments` appears within the `FormalParameters` of a strict mode `FunctionDeclaration` or `FunctionExpression` (0)

• 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` (0, 19.2.1).

• 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 (9.2.3, 9.4.4).

• 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 (0). Attempting to dynamically define such a strict mode function using the `Function` constructor (19.2.1) will throw a `SyntaxError` exception.
Annex D  
Corrections and Clarifications with Possible Compatibility Impact

In Edition 6

20.3.1.14: Previous editions permitted the TimeClip abstract operation to return either +0 or −0 as the representation of a 0 time value. The 6th Edition specifies that +0 always returned. This means that for the 6th Edition the time value of a Date object is never observably −0 and methods that return time values never return −0.

20.3.1.15: 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”.

20.3.4.36: If the year cannot be represented using the Date Time String Format specified in 20.3.1.15 a RangeError exception is thrown. Previous editions did not specify the behavior for that case.

20.3.4.41: Previous editions did not specify the value returned by Date.prototype.toString when this time value is NaN. The 6th Edition specifies the result to be the String value is "Invalid Date".

21.2.3.1, 21.2.3.3.4: If any LineTerminator code points in the value of the source property of an RegExp instance must be expressed using an escape sequence. Edition 5.1 only required the escaping of “/”.

In Edition 5.1

Clause references in this list refer to the clause numbers used in 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.vi.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.

15.3.3.3: 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.

15.4.3.12: In step 9.a, incorrect reference to relativeStart was replaced with a reference to actualStart.

15.4.3.15: Clarified that the default value for fromIndex is the length minus 1 of the array.

15.4.3.18: In step 10 (corresponding to step 8 in 5.1) of the algorithm, undefined is now the specified return value.

15.4.3.22: In step 11.d.iii (corresponding to 9.c.ii in 5.1) 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.iii and 3.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.

15.10.3.2: 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 abstract 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 Edition 5

Clause references in this list refer to the clause numbers used in 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 dynamically resolved value of the correspondingly identifier binding.

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 slot 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

E.1 In the 6th Edition

8: In Edition 6, Function calls are not allowed to return a Reference value.

13.6: In Edition 6, a terminating semi-colon is no longer required at the end of a do-while statement.

13.6: Prior to Edition 6, an initialisation expression could appear as part of the VariableDeclaration that precedes the in keyword. The value of that expression was always discarded. In Edition 6, the ForBind in that same position does not allow the occurrence of such an initialiser.

13.14: In Edition 6, it is an early error for a Catch clause to contain 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 Initialiser value would be assigned to the Catch parameter.

14.3 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.

19.1.2.2 and 19.1.2.3: In Edition 6, all property additions and changes are processed, even if one of them throws an exception. If an exception occurs during such processing, the first such exception is thrown after all properties are processed. In Edition 5, processing of property additions and changes immediately terminated when the first exception occurred.

19.1.2.5: In Edition 6, if the argument to Object.freeze is not an object it is treated as if it was a non-extensible ordinary object with no own properties. In Edition 5, a non-object argument always causes a TypeError to be thrown.

19.1.2.6: In Edition 6, if the argument to Object.getOwnPropertyDescriptor is not an object an attempt is made to coerce the argument using ToObject. If the coercion is successful the result is used in place of the original argument value. In Edition 5, a non-object argument always causes a TypeError to be thrown.

19.1.2.7: In Edition 6, if the argument to Object.getOwnPropertyNames is not an object an attempt is made to coerce the argument using ToObject. If the coercion is successful the result is used in place of the original argument value. In Edition 5, a non-object argument always causes a TypeError to be thrown.

19.1.2.9: In Edition 6, if the argument to Object.getPrototypeOf is not an object an attempt is made to coerce the argument using ToObject. If the coercion is successful the result is used in place of the original argument value. In Edition 5, a non-object argument always causes a TypeError to be thrown.

19.1.2.11: In Edition 6, if the argument to Object.isExtensible is not an object it is treated as if it was a non-extensible ordinary object with no own properties. In Edition 5, a non-object argument always causes a TypeError to be thrown.

19.1.2.12: In Edition 6, if the argument to Object.isFrozen is not an object it is treated as if it was a non-extensible ordinary object with no own properties. In Edition 5, a non-object argument always causes a TypeError to be thrown.
19.1.2.13: In Edition 6, if the argument to \texttt{Object.isSealed} is not an object it is treated as if it was a non-extensible ordinary object with no own properties. In Edition 5, a non-object argument always causes a \texttt{TypeError} to be thrown.

19.1.2.14: In Edition 6, if the argument to \texttt{Object.keys} is not an object an attempt is made to coerce the argument using \texttt{ToObject}. If the coercion is successful the result is used in place of the original argument value. In Edition 5, a non-object argument always causes a \texttt{TypeError} to be thrown.

19.1.2.16: In Edition 6, if the argument to \texttt{Object.preventExtensions} is not an object it is treated as if it was a non-extensible ordinary object with no own properties. In Edition 5, a non-object argument always causes a \texttt{TypeError} to be thrown.

19.1.2.18: In Edition 6, if the argument to \texttt{Object.seal} is not an object it is treated as if it was a non-extensible ordinary object with no own properties. In Edition 5, a non-object argument always causes a \texttt{TypeError} to be thrown.

19.2.4.1: In Edition 6, the \texttt{length} property of function instances is configurable. In previous editions it was non-configurable.

19.3.3 In Edition 6, the Boolean prototype object is not a Boolean instance. In previous editions it was a Boolean instance whose Boolean value was \texttt{false}.

20.1.3 In Edition 6, the Number prototype object is not a Number instance. In previous editions it was a Number instance whose number value was +0.

20.3.4 In Edition 6, the Date prototype object is not a Date instance. In previous editions it was a Date instance whose TimeValue was NaN.

22.1.3 In Edition 6, the Array prototype object is not an Array instance. In previous editions it was an Array instance with a length property whose value was +0.

21.1.3 In Edition 6, the String prototype object is not a String instance. In previous editions it was a String instance whose String value was the empty string.

21.1.3.22 and 21.1.3.24 In Edition 6, lowercase/upper conversion processing operates on code points. In previous editions such the conversion processing was only applied to individual code units. The only affected code points are those in the Deseret block of Unicode

21.1.3.25 In Edition 6, the \texttt{String.prototype.trim} method is defined to recognize white space code points that may exist outside of the Unicode BMP. However, as of Unicode 6.1 no such code points are defined. In previous editions such code points would not have been recognized as white space.

21.2.5 In Edition 6, the RegEx prototype object is not a RegExp instance. In previous editions it was a RegExp instance whose pattern is the empty string.

21.2.5 In Edition 6, the source, global, ignoreCase, and multiline are accessor properties defined on the RegExp prototype object. In previous editions they were data properties defined on RegExp instances.

22.1.3 In Edition 6, the Array prototype object is not an Array instance. In previous editions it was an Array instance with a length property whose value was +0.

\textbf{E.2 In the 5th Edition}

Clause references in this list refer to the clause numbers used in Edition 5 and 5.1.
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.

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.

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.

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.getOwnPropertyNames, Object.isSealed, Object.isFrozen, Object.preventExtensions, Object.isExtensible, Object.reflect, Function.prototype.bind, Array.prototype.bind, Array.prototypeизма lastIndexOf, Array.prototype.every, Array.prototype.some, Array.prototype.reduce, Array.prototype.reduceRight, String.prototype.trim, Date.now, Date.prototype.toISOString, Date.prototype.toLocaleString, 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.

15.3.3.3, 15.3.3.4: 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.

15.3.4.2: 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

**TODO:** This Table is out of date and incomplete.

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Purpose</th>
<th>Definitions</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoundNames</td>
<td>Produces a list of the identifiers bound by a production. Does not include identifiers that are bound within inner environments associated with the production.</td>
<td>12.2.1, 12.2.2, 12.2.4, 12.6.4, 13.1, 13.2, 13.5</td>
<td></td>
</tr>
<tr>
<td>ConstructorMethod</td>
<td>From a <code>ClassBody</code> return the first <code>ClassElement</code> whose <code>PropName</code> is &quot;constructor&quot;. Returns empty if the <code>ClassBody</code> does not contain one.</td>
<td>13.5</td>
<td></td>
</tr>
<tr>
<td>Contains</td>
<td>Determine if a grammar production either directly or indirectly includes a grammar symbol.</td>
<td>5.3, 13.1, 13.2, 13.5</td>
<td></td>
</tr>
<tr>
<td>CoveredFormalsList</td>
<td>Reparse a covered <code>Expression</code> using <code>FormalsList</code> as the goal symbol.</td>
<td>13.2</td>
<td></td>
</tr>
<tr>
<td>CV</td>
<td>Determines the &quot;character value&quot; of a component of a <code>StringLiteral</code>.</td>
<td>7.8.4</td>
<td></td>
</tr>
<tr>
<td>Elision Width</td>
<td>Determine the number of commas in an <code>Elision</code>.</td>
<td>11.1.4.1</td>
<td></td>
</tr>
<tr>
<td>ExpectedArgumentCount</td>
<td>Determine the &quot;length&quot; of an argument list for the purpose of initialising the &quot;length&quot; property of a function object.</td>
<td>13.1, 13.2, 13.3</td>
<td></td>
</tr>
<tr>
<td>HasInitialiser</td>
<td>Determines whether the production contains an Initialiser production.</td>
<td>12.2.4, 13.1</td>
<td></td>
</tr>
<tr>
<td>IsConstantDeclaration</td>
<td>Determines whether the production introduces an immutable environment record binding</td>
<td>12.2, 13.1, 13.5</td>
<td></td>
</tr>
<tr>
<td>IsValidAssignmentPattern</td>
<td>Determines if a <code>LeftHandSideExpression</code> is a valid assignment target. Primarily for dealing with destructuring assignment targets.</td>
<td>11.2</td>
<td></td>
</tr>
<tr>
<td>LexicalDeclarations</td>
<td>Return a List containing the components of a production that are processed as lexical declarations</td>
<td>12.1, 12.11, 12.5</td>
<td></td>
</tr>
<tr>
<td>LexicallyDeclaredNames</td>
<td>Returns a list of the lexically scoped identifiers declared by a production.</td>
<td>12.1, 13.1, 13.2, 13.5</td>
<td></td>
</tr>
<tr>
<td>PrototypeMethodDefinitions</td>
<td>Return a list of the non-static <code>MethodDefinition</code> productions that are part of a <code>ClassElementList</code>.</td>
<td>13.5</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Section(s)</td>
<td></td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------</td>
<td></td>
</tr>
<tr>
<td>MV</td>
<td>Determines the “mathematical value” of a numeric literal or component of a numeric literal.</td>
<td>7.8.3</td>
<td></td>
</tr>
<tr>
<td>PropName</td>
<td>Determines the string value of the property name referenced by a production.</td>
<td>11.1.5.1, 13.3, 13.5</td>
<td></td>
</tr>
<tr>
<td>PropNameList</td>
<td>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.</td>
<td>11.5.1, 13.5</td>
<td></td>
</tr>
<tr>
<td>PrototypeMethodDefinitions</td>
<td>Return a list of the non-static MethodDefinition productions that are part of a ClassElementList.</td>
<td>13.5</td>
<td></td>
</tr>
<tr>
<td>ReferencesSuper</td>
<td>Determine if a MethodDefinition contains any references to the ReservedWord super.</td>
<td>13.3</td>
<td></td>
</tr>
<tr>
<td>SpecialMethod</td>
<td>Determine if a MethodDefinition defines a generator method or an accessor property.</td>
<td>13.3</td>
<td></td>
</tr>
<tr>
<td>StaticMethodDefinitions</td>
<td>Return a list of the static MethodDefinition productions that are part of a ClassElementList.</td>
<td>13.5</td>
<td></td>
</tr>
<tr>
<td>SV</td>
<td>Determines the “string value” of a StringLiteral or component of a StringLiteral.</td>
<td>7.8.4</td>
<td></td>
</tr>
<tr>
<td>VarDeclaredNames</td>
<td>Returns a list of the local top-level scoped identifiers declared by a production. These are identifiers that are scoped as if by a var statement.</td>
<td>12.1, 12.5, 12.6.1, 12.6.2, 12.6.3, 12.6.4, 12.12, 13.1, 13.5</td>
<td></td>
</tr>
</tbody>
</table>
Bibliography


