Contents

Introduction .................................................................................................................. vii
1 Scope .......................................................................................................................... 1
2 Conformance .............................................................................................................. 1
3 Normative references ............................................................................................... 1
4 Overview .................................................................................................................... 2
4.1 Web Scripting .......................................................................................................... 3
4.2 ECMAScript Overview ......................................................................................... 3
4.2.1 Objects ............................................................................................................. 5
4.2.2 The Strict Variant of ECMAScript .................................................................. 5
4.3 Terms and definitions ........................................................................................... 5
4.4 Organization of This Specification ....................................................................... 9
5 Notational Conventions ......................................................................................... 10
5.1 Syntactic and Lexical Grammars ........................................................................ 10
5.1.1 Context-Free Grammars ................................................................................ 10
5.1.2 The Lexical and RegExp Grammars ................................................................ 10
5.1.3 The Numeric String Grammar ....................................................................... 10
5.1.4 The Syntactic Grammar .................................................................................. 11
5.1.5 Grammar Notation .......................................................................................... 11
5.2 Algorithm Conventions ....................................................................................... 16
5.3 Static Semantic Rules ......................................................................................... 18
6 ECMAScript Data Types and Values ...................................................................... 19
6.1 ECMAScript Language Types ............................................................................ 19
6.1.1 The Undefined Type ....................................................................................... 19
6.1.2 The Null Type .................................................................................................. 19
6.1.3 The Boolean Type ............................................................................................ 19
6.1.4 The String Type .............................................................................................. 19
6.1.5 The Symbol Type ............................................................................................ 20
6.1.6 The Number Type ........................................................................................... 21
6.1.7 The Object Type .............................................................................................. 21
6.2 ECMAScript Specification Types ......................................................................... 33
6.2.1 The List and Record Specification Type ......................................................... 33
6.2.2 The Completion Record Specification Type .................................................... 33
6.2.3 The Reference Specification Type .................................................................... 34
6.2.4 The Property Descriptor Specification Type .................................................. 36
6.2.5 The Lexical Environment and Environment Record Specification Types ......... 39
6.2.6 Data Blocks ...................................................................................................... 39
7 Abstract Operations ................................................................................................. 40
7.1 Type Conversion and Testing ............................................................................. 40
7.1.1 ToPrimitive ( input [], PreferredType ) ....................................................... 40
7.1.2 ToBoolean ( argument ) ............................................................................... 41
7.1.3 ToNumber ( argument ) ............................................................................... 42
7.1.4 ToInteger ( argument ) ............................................................................... 45
7.1.5 ToInt32 ( argument ) — Signed 32 Bit Integer ............................................. 45
7.1.6 ToUint32 ( argument ) — Unsigned 32 Bit Integer ........................................ 46
7.1.7 ToInt16 ( argument ) — Signed 16 Bit Integer ............................................. 46
7.1.8 ToUint16 ( argument ) — Unsigned 16 Bit Integer ........................................ 46
7.1.9  ToInt8 ( argument ) — Signed 8 Bit Integer ................................................................. 46
7.1.10 ToUint8 ( argument ) — Unsigned 8 Bit Integer ........................................................... 47
7.1.11 ToUint8Clamp ( argument ) — Unsigned 8 Bit Integer, Clamped .................................... 47
7.1.12 ToString ( argument ) ....................................................................................................... 47
7.1.13 ToObject ( argument ) ....................................................................................................... 49
7.1.14 ToPropertyKey ( argument ) ............................................................................................ 49
7.1.15 ToLength ( argument ) ...................................................................................................... 50
7.1.16 CanonicalNumericString ( argument ) ............................................................................. 50
7.2  Testing and Comparison Operations .................................................................................... 50
7.2.1  RequireObjectCoercible ( argument ) ................................................................................ 50
7.2.2  IsCallable ( argument ) .................................................................................................... 50
7.2.3  SameValue(x, y) ................................................................................................................ 51
7.2.4  SameValueZero(x, y) .......................................................................................................... 51
7.2.5  IsConstructor ( argument ) ............................................................................................... 52
7.2.6  IsPropertyKey ( argument ) ............................................................................................. 52
7.2.7  IsExtensible ( O ) .............................................................................................................. 52
7.2.8  IsInteger ( argument ) ....................................................................................................... 52
7.2.9  Abstract Relational Comparison ....................................................................................... 53
7.2.10 Abstract Equality Comparison ........................................................................................ 54
7.2.11 Strict Equality Comparison ............................................................................................. 54
7.3  Operations on Objects ......................................................................................................... 55
7.3.1  Get ( O, P ) ........................................................................................................................ 55
7.3.2  Put ( O, P, V, Throw ) ...................................................................................................... 55
7.3.3  CreateDataProperty ( O, P, V ) ......................................................................................... 55
7.3.4  CreateDataPropertyOrThrow ( O, P, V ) .......................................................................... 56
7.3.5  DefinePropertyOrThrow ( O, P, desc ) ............................................................................. 56
7.3.6  DeletePropertyOrThrow ( O, P ) ....................................................................................... 56
7.3.7  GetMethod ( O, P ) ............................................................................................................ 56
7.3.8  HasProperty ( O, P ) .......................................................................................................... 57
7.3.9  HasOwnProperty ( O, P ) .................................................................................................. 57
7.3.10 Invoke ( O, P, [args] ) ...................................................................................................... 57
7.3.11 Set IntegrityLevel ( O, level ) ............................................................................................ 58
7.3.12 Test IntegrityLevel ( O, level ) .......................................................................................... 58
7.3.13 CreateArrayFromList ( elements ) ................................................................................. 59
7.3.14 CreateListFromArrayLike ( obj ) ..................................................................................... 59
7.3.15 OrdinaryHasInstance ( C, O ) ........................................................................................ 60
7.3.16 GetPrototypeFromConstructor ( constructor, intrinsicDefaultProto ) ......................... 60
7.3.17 CreateFromConstructor ( F ) .......................................................................................... 60
7.3.18 Construct ( F, argumentsList ) ....................................................................................... 61
7.3.19 GetOption ( options, P ) ................................................................................................. 61
7.3.20 EnumerableOwnProperty ( O ) ........................................................................................... 61
7.3.21 GetFunctionRealm ( obj ) — Abstract Operation ............................................................. 62
7.4  Operations on Iterator Objects .......................................................................................... 62
7.4.1  CheckIterable ( obj ) ....................................................................................................... 62
7.4.2  GetIterator ( obj, method ) .............................................................................................. 62
7.4.3  IteratorNext ( iterator, value ) ........................................................................................ 62
7.4.4  IteratorComplete ( iterator, completion ) ........................................................................ 63
7.4.5  IteratorValue ( iterator ) ................................................................................................ 63
7.4.6  IteratorStep ( iterator ) ................................................................................................... 63
7.4.7  IteratorClose ( iterator ) .................................................................................................. 63
7.4.8  CreateIterResultObject ( value, done ) .......................................................................... 63
7.4.9  CreateListIterator ( list ) ............................................................................................... 64
7.4.10 CreateEmptyIterator ( ) ................................................................................................. 64
7.4.11 CreateCompoundIterator ( iterator1, iterator2 ) .......................................................... 64
7.5  Operations on Promise Objects ......................................................................................... 65
7.5.1 PromiseNew (executor) Abstract Operation ................................................. 66
7.5.2 PromiseBuiltInCapability () Abstract Operation ............................................. 66
7.5.3 PromiseOf (value) Abstract Operation .......................................................... 66
7.5.4 PromiseAll (promiseList) Abstract Operation ................................................. 66
7.5.5 PromiseCatch (promise, rejectedAction) Abstract Operation ......................... 66
7.5.6 PromiseThen (promise, resolvedAction, rejectedAction) Abstract Operation ....... 66

8 Executable Code and Execution Contexts ......................................................... 66
8.1 Lexical Environments ...................................................................................... 66
8.1.1 Environment Records ...................................................................................... 67
8.2 Lexical Environment Operations ...................................................................... 67
8.2.1 CreateRealm () Abstract Operation ............................................................... 82
8.2.2 CreateIntrinsics (realmRec) Abstract Operation ............................................. 82
8.2.3 SetRealmGlobalObj (realmRec, globalObj) Abstract Operation ...................... 83
8.3 Execution Contexts ......................................................................................... 83
8.3.1 ResolveBinding (name) Abstract Operation .................................................... 85
8.3.2 GetThisEnvironment () Abstract Operation ................................................... 85
8.3.3 ResolveThisBinding () Abstract Operation ..................................................... 85
8.3.4 GetGlobalObject () Abstract Operation .......................................................... 86
8.4 Jobs and Job Queues ....................................................................................... 86
8.4.1 EnqueueJob (queueName, job, arguments) Abstract Operation ..................... 87
8.4.2 NextJob result ............................................................................................... 87
8.5 Initialization ..................................................................................................... 88
8.5.1 InitializeFirstRealm (realm) Abstract Operation ......................................... 88

9 Ordinary and Exotic Objects Behaviours ............................................................ 88
9.1 Ordinary Object Internal Methods and Internal Slots ..................................... 88
9.1.1 [[GetPrototypeOf]] () ....................................................................................... 89
9.1.2 [[SetPrototypeOf]] (V) ..................................................................................... 89
9.1.3 [[IsExtensible]] () ............................................................................................. 89
9.1.4 [[PreventExtensions]] () ................................................................................ 89
9.1.5 [[GetProperty]] (P) .......................................................................................... 90
9.1.6 [[DefineOwnProperty]] (P, Desc) ................................................................... 90
9.1.7 [[HasProperty]] (P) ......................................................................................... 92
9.1.8 [[Get]] (P, Receiver) ....................................................................................... 92
9.1.9 [[Set]] (P, V, Receiver) ................................................................................... 92
9.1.10 [[Delete]] (P) .................................................................................................. 93
9.1.11 [[Enumerate]] () ............................................................................................ 93
9.1.12 [[OwnPropertyKeys]] () ................................................................................ 94
9.1.13 ObjectCreate(proto, internalSlotsList) Abstract Operation ......................... 94
9.1.14 OrdinaryCreateFromConstructor (constructor, intrinsicDefaultProto, internalSlotsList) ................................................................. 95
9.2 ECMAScript Function Objects ...................................................................... 95
9.2.1 [[GetOwnProperty]] (P) .................................................................................... 96
9.2.2 [[Call]] (thisArgument, argumentsList) .......................................................... 97
9.2.3 [[Construct]] (argumentsList) ....................................................................... 98
9.2.4 FunctionAllocate (functionPrototype, strict) Abstract Operation ................. 98
9.2.5 FunctionInitialize (F, kind, Strict, ParameterList, Body, Scope) Abstract Operation ................................................................. 98
9.2.6 FunctionCreate (kind, ParameterList, Body, Scope, Strict) Abstract Operation ................................................................. 99
9.2.7 GeneratorFunctionCreate (kind, ParameterList, Body, Scope, Strict) Abstract Operation ................................................................. 99
9.2.8 AddRestrictedFunctionProperties (F, realm) Abstract Operation ................. 99
9.2.9 MakeConstructor (F, writablePrototype, prototype) Abstract Operation ........ 100
9.2.10 MakeMethod (F, methodName, homeObject) Abstract Operation .................. 100
9.2.11 SetFunctionName (F, name, prefix) Abstract Operation .............................. 101
9.2.12 CloneMethod(function, newHome, newName) Abstract Operation ................. 101
12.7.5 Applying the Additive Operators to Numbers ................................................................. 194
12.8 Bitwise Shift Operators ........................................................................................................ 195
12.8.1 Static Semantics: isFunctionDefinition ............................................................................ 195
12.8.2 Static Semantics: isValidSimpleAssignmentTarget ....................................................... 195
12.8.3 The Left Shift Operator ( << ) .......................................................................................... 196
12.8.4 The Signed Right Shift Operator ( >> ) ............................................................................. 196
12.8.5 The Unsigned Right Shift Operator ( >>> ) ....................................................................... 197
12.9 Relational Operators ............................................................................................................ 197
12.9.1 Static Semantics: isFunctionDefinition ............................................................................ 197
12.9.2 Static Semantics: isValidSimpleAssignmentTarget ....................................................... 198
12.9.3 Runtime Semantics: Evaluation ...................................................................................... 198
12.9.4 Runtime Semantics: instanceofOperator(O, C) ........................................................... 199
12.10 Equality Operators ............................................................................................................. 200
12.10.1 Static Semantics: isFunctionDefinition ........................................................................ 200
12.10.2 Static Semantics: isValidSimpleAssignmentTarget ..................................................... 200
12.10.3 Runtime Semantics: Evaluation .................................................................................... 200
12.11 Binary Bitwise Operators .................................................................................................. 202
12.11.1 Static Semantics: isFunctionDefinition ........................................................................ 202
12.11.2 Static Semantics: isValidSimpleAssignmentTarget ..................................................... 202
12.11.3 Runtime Semantics: Evaluation .................................................................................... 202
12.12 Binary Logical Operators .................................................................................................. 203
12.12.1 Static Semantics: isFunctionDefinition ........................................................................ 203
12.12.2 Static Semantics: isValidSimpleAssignmentTarget ..................................................... 203
12.12.3 Runtime Semantics: Evaluation .................................................................................... 203
12.13 Conditional Operator ( ? : ) ............................................................................................. 204
12.13.1 Static Semantics: isFunctionDefinition ........................................................................ 204
12.13.2 Static Semantics: isValidSimpleAssignmentTarget ..................................................... 204
12.13.3 Runtime Semantics: Evaluation .................................................................................... 204
12.14 Assignment Operators ...................................................................................................... 205
12.14.1 Static Semantics: Early Errors ..................................................................................... 205
12.14.2 Static Semantics: isFunctionDefinition ........................................................................ 205
12.14.3 Static Semantics: isValidSimpleAssignmentTarget ..................................................... 206
12.14.4 Runtime Semantics: Evaluation .................................................................................... 206
12.15 Destructuring Assignment .................................................................................................. 207
12.15.1 Static Semantics: isFunctionDefinition ........................................................................ 212
12.15.2 Static Semantics: isValidSimpleAssignmentTarget ..................................................... 212
12.15.3 Runtime Semantics: Evaluation .................................................................................... 212
13 ECMAScript Language: Statements and Declarations .......................................................... 213
13.0 Statement Semantics ........................................................................................................... 213
13.0.1 Static Semantics: DeclarationPart .................................................................................. 213
13.0.2 Static Semantics: VarDeclaredNames ............................................................................. 213
13.0.3 Static Semantics: VarScopedDeclarations ..................................................................... 214
13.0.4 Runtime Semantics: LabelledEvaluation ...................................................................... 214
13.0.5 Runtime Semantics: Evaluation .................................................................................... 215
13.1 Block .................................................................................................................................. 215
13.1.1 Static Semantics: Early Errors ..................................................................................... 215
13.1.2 Static Semantics: LexicallyDeclaredNames .................................................................... 215
13.1.3 Static Semantics: LexicallyScopedDeclarations ............................................................. 216
13.1.4 Static Semantics: TopLevelLexicallyDeclaredNames ..................................................... 216
13.1.5 Static Semantics: TopLevelLexicallyScopedDeclarations .............................................. 216
13.1.6 Static Semantics: TopLevelVarDeclaredNames ............................................................... 217
13.1.7 Static Semantics: TopLevelVarScopedDeclarations ...................................................... 217
13.1.8 Static Semantics: VarDeclaredNames ............................................................................ 218
15.1.7 Runtime Semantics: ScriptEvaluation ................................................................. 295
15.1.8 Runtime Semantics: GlobalDeclarationInstantiation ........................................ 295
15.1.9 Runtime Semantics: ScriptEvaluationJob ( source ) ........................................ 297
15.2 Modules .................................................................................................................. 297
15.2.0 Module Static Semantics ..................................................................................... 298
15.2.1 Imports ................................................................................................................ 303
15.2.2 Exports ................................................................................................................ 306
15.2.3 Runtime Semantics: Loader State ....................................................................... 310
15.2.4 Runtime Semantics: Module Loading ............................................................... 312
15.2.5 Runtime Semantics: Module Linking ............................................................... 319
15.2.6 Runtime Semantics: Module Evaluation ........................................................... 328

16 Error Handling and Language Extensions .............................................................. 329
16.1 Forbidden Extensions ........................................................................................... 330

17 ECMAScript Standard Built-in Objects .................................................................. 330
18 The Global Object ..................................................................................................... 332
18.1 Value Properties of the Global Object .................................................................. 332
18.1.1 Infinity ............................................................................................................... 332
18.1.2 NaN ................................................................................................................. 332
18.1.3 undefined ....................................................................................................... 332
18.2 Function Properties of the Global Object ......................................................... 332
18.2.1 eval ( x ) ......................................................................................................... 332
18.2.2 isFinite ( number ) ......................................................................................... 334
18.2.3 isNaN ( number ) ........................................................................................... 334
18.2.4 parseFloat ( string ) ....................................................................................... 334
18.2.5 parseInt ( string , radix ) ................................................................................ 334
18.2.6 URI Handling Functions .................................................................................. 335
18.3 Constructor Properties of the Global Object .................................................... 340
18.3.1 Array ( . . . ) .................................................................................................... 341
18.3.2 ArrayBuffer ( . . . ) ........................................................................................ 341
18.3.3 Boolean ( . . . ) ................................................................................................. 341
18.3.4 DataView ( . . . ) ............................................................................................. 341
18.3.5 Date ( . . . ) ....................................................................................................... 341
18.3.6 Error ( . . . ) ..................................................................................................... 341
18.3.7 EvalError ( . . . ) ............................................................................................. 341
18.3.8 Float32Array ( . . . ) ....................................................................................... 341
18.3.9 Float64Array ( . . . ) ....................................................................................... 341
18.3.10 Function ( . . . ) ........................................................................................... 341
18.3.11 Int16Array ( . . . ) ......................................................................................... 341
18.3.12 Int32Array ( . . . ) ........................................................................................ 341
18.3.13 Int32Array ( . . . ) ........................................................................................ 341
18.3.14 Map ( . . . ) ................................................................................................... 341
18.3.15 Number ( . . . ) .............................................................................................. 342
18.3.16 Object ( . . . ) ................................................................................................. 342
18.3.17 Promise ( . . . ) ............................................................................................. 342
18.3.18 RangeError ( . . . ) ......................................................................................... 342
18.3.19 ReferenceError ( . . . ) .................................................................................. 342
18.3.20 RegExp ( . . . ) ............................................................................................... 342
18.3.21 Set ( . . . ) ...................................................................................................... 342
18.3.22 String ( . . . ) ................................................................................................ 342
18.3.23 Symbol ( . . . ) .............................................................................................. 342
18.3.24 SyntaxError ( . . . ) ......................................................................................... 342
18.3.25 TypeError ( . . . ) .......................................................................................... 342
18.3.26 Uint8Array ( . . . ) ........................................................................................ 342
18.3.27 Uint8ClampedArray ( . . . ) .......................................................................... 342

X © Ecma International 2014
26.2.3 Properties of the Reflect.Loader Prototype Object ................................................................. 581
26.2.4 Properties of Reflect.Loader Instances ......................................................................................... 589
26.2.5 Loader Iterator Objects .................................................................................................................. 589
26.3 The System Object ............................................................................................................................ 590
26.4 Proxy Objects .................................................................................................................................... 590
26.4.1 The Proxy Constructor Function .................................................................................................... 590
26.4.2 Properties of the Proxy Constructor Function .................................................................................. 591

Annex A (informative) Grammar Summary ............................................................................................... 592
A.1 Lexical Grammar ................................................................................................................................. 592
A.2 Expressions .......................................................................................................................................... 599
A.3 Statements ........................................................................................................................................... 603
A.4 Functions and Scripts ........................................................................................................................... 605
A.5 Number Conversions ............................................................................................................................ 606
A.6 Universal Resource Identifier Character Classes ............................................................................... 607
A.7 Regular Expressions .............................................................................................................................. 608

Annex B (normative) Additional ECMAScript Features for Web Browsers .............................................. 612
B.1 Additional Syntax ................................................................................................................................. 612
B.1.1 Numeric Literals .............................................................................................................................. 612
B.1.2 String Literals .................................................................................................................................. 612
B.1.3 HTML-like Comments ..................................................................................................................... 613
B.1.4 Regular Expressions Patterns ........................................................................................................ 613
B.2 Additional Built-in Properties ............................................................................................................. 617
B.2.1 Additional Properties of the Global Object ...................................................................................... 617
B.2.2 Additional Properties of the Object.prototype Object ................................................................. 618
B.2.3 Additional Properties of the String.prototype Object .................................................................. 619
B.2.4 Additional Properties of the Date.prototype Object .................................................................... 621
B.2.5 Additional Properties of the RegExp.prototype Object ............................................................. 622
B.3 Other Additional Features .................................................................................................................. 623
B.3.1 __proto__ Property Names in Object Initializers ........................................................................ 623
B.3.2 Labelled Function Declarations ..................................................................................................... 624
B.3.3 Block-Level Function Declarations Web Legacy Compatibility Semantics ................................ 624
B.3.4 FunctionDeclarations in IfStatement Statement Clauses .......................................................... 625
B.3.5 VariableStatements in Catch blocks ............................................................................................. 626

Annex C (informative) The Strict Mode of ECMAScript .......................................................................... 627

Annex D (informative) Corrections and Clarifications with Possible Compatibility Impact .................. 629
D.1 In Edition 6 ........................................................................................................................................... 629
D.2 In Edition 5.1 ....................................................................................................................................... 629
D.3 In Edition 5 ........................................................................................................................................... 631

Annex E (informative) Additions and Changes That Introduce Incompatibilities with Prior Editions ........................................................................................................................................................................... 633
E.1 In the 6th Edition ................................................................................................................................. 633
E.2 In the 5th Edition .................................................................................................................................. 635
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.


After publication of the third edition, ECMAScript 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 published1 as the fourth edition of ECMAScript, it informs continuing evolution of the language. The fifth edition of ECMAScript (published as ECMA-262 5th edition) codified de facto interpretations of the language specification that have become common among browser implementations and added support for new features that had 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.


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 source code input in conformance with the Unicode Standard, Version 5.1.0 or later and ISO/IEC 10646. If the adopted ISO/IEC 10646-1 subset is not otherwise specified, it is presumed to be the Unicode set, collection 10646.

A conforming implementation of ECMAScript 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
4 Overview

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

ECMA-Script is an object-oriented programming language for performing calculations and manipulating computational objects within a host environment. ECMA-Script, 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 ECMA-Script 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 ECMA-Script program.

A scripting language is a programming language that is used to manipulate, customize, 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. ECMA-Script was originally designed to be used as a scripting language, but has become widely used as a general purpose programming language.

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

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

![Figure 1 — Object/Prototype Relationships](image)

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

All objects that do not directly contain a particular property that their prototype contains share that property and its value. Figure 1 illustrates this:
CF is a constructor (and also an object). Five objects have been created by using `new` expressions: `cf1`, `cf2`, `cf3`, `cf4`, and `cf5`. Each of these objects contains properties named `q1` and `q2`. The dashed lines represent the implicit prototype relationship; so, for example, `cf3`'s prototype is `CFp`. The constructor, `CF`, has two properties itself, named `P1` and `P2`, which are not visible to `CFp`, `cf1`, `cf2`, `cf3`, or `cf5`. The property named `CFP1` in `CFp` is shared by `cf1`, `cf2`, `cf3`, and `cf4` (but not by `CF`), as are any properties found in `CFp`'s implicit prototype chain that are not named `q1`, `q2`, or `CFP1`. Notice that there is no implicit prototype link between `CF` and `CFp`.

Unlike most class-based object languages, properties can be added to objects dynamically by assigning values to them. That is, constructors are not required to name or assign values to all or any of the constructed object's properties. In the above diagram, one could add a new shared property for `cf1`, `cf2`, `cf3`, `cf4`, and `cf5` by assigning a new value to the property in `CFp`.

Although ECMAScript objects are not inherently class-based, it is often convenient to define class-like abstractions based upon a common pattern of constructor functions, prototype objects, and methods. The ECMAScript built-in object themselves follow such a class-like pattern. The ECMAScript language includes syntactic class definitions that permit programmers to concisely define objects that conform to the same class-like abstraction pattern used by the built-in objects.

### 4.2.2 The Strict Variant of ECMAScript

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

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

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

### 4.3 Terms and definitions

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

#### 4.3.1 type

set of data values as defined in clause 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 initializes 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 does not have the default 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 specified and supplied by an ECMAScript implementation

NOTE Standard built-in objects are defined in this specification. An ECMAScript implementation may specify and supply additional kinds of built-in objects. 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 (21.1.1.1).

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 (20.1.1.1).

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 6–8.5.1 define the execution environment within which ECMAScript programs operate.

Clauses 10–16 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 Unicode code points that conform to the rules for SourceCharacter defined in 10.1. It defines a set of productions, starting from the goal symbolInputElementDiv or InputElementRegExp, that describe how sequences of such code points 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 code points as defined by SourceCharacter. It defines a set of productions, starting from the goal symbolPattern, that describe how sequences of code points 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, 14, and 15. 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 code points 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 line terminator character appears in certain “awkward” places.

In certain cases in order to avoid ambiguities the syntactic grammar uses generalized productions that permit token sequences that are not valid ECMAScript scripts. For example, this technique is used for 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 exactly as written. All terminal symbol code points 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 code points 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 Initializer<opt>

is a convenient abbreviation for:

VariableDeclaration:  
BindingIdentifier  
BindingIdentifier Initializer

and that:

IterationStatement:  
for ( LexicalDeclaration Expression<opt> ; Expression<opt> ) Statement

is a convenient abbreviation for:

IterationStatement:  
for ( LexicalDeclaration ; Expression<opt> ) Statement  
for ( LexicalDeclaration Expression ; Expression<opt> ) Statement

which in turn is an abbreviation for:

IterationStatement:  
for ( LexicalDeclaration ; ) Statement  
for ( LexicalDeclaration ; Expression ) Statement  
for ( LexicalDeclaration Expression ; ) Statement  
for ( LexicalDeclaration Expression ; Expression ) Statement

so, in this example, the nonterminal `IterationStatement` actually has four alternative right-hand sides.
A production may be parameterized by a subscripted annotation of the form \([\text{parameters}]\), which may appear as a suffix to the nonterminal symbol defined by the production. \(\text{parameters}\) may be either a single name or a comma separated list of names. A parameterized production is shorthand for a set of productions defining all combinations of the parameter names, preceded by an underscore, appended to the parameterized nonterminal symbol. This means that:

\[
\text{StatementList[Return]} : \\
\text{ReturnStatement} \\
\text{ExpressionStatement}
\]

is a convenient abbreviation for:

\[
\text{StatementList} : \\
\text{ReturnStatement} \\
\text{ExpressionStatement}
\]

\[
\text{StatementList_Return} : \\
\text{ReturnStatement} \\
\text{ExpressionStatement}
\]

and that:

\[
\text{StatementList[Return, In]} : \\
\text{ReturnStatement} \\
\text{ExpressionStatement}
\]

is an abbreviation for:

\[
\text{StatementList} : \\
\text{ReturnStatement} \\
\text{ExpressionStatement}
\]

\[
\text{StatementList_Return} : \\
\text{ReturnStatement} \\
\text{ExpressionStatement}
\]

\[
\text{StatementList_In} : \\
\text{ReturnStatement} \\
\text{ExpressionStatement}
\]

\[
\text{StatementList_Return_In} : \\
\text{ReturnStatement} \\
\text{ExpressionStatement}
\]

Multiple parameters produce a combinatory number of productions, not all of which are necessarily referenced in a complete grammar.

References to nonterminals on the right hand side of a production can also be parameterized. For example:

\[
\text{StatementList} : \\
\text{ReturnStatement} \\
\text{ExpressionStatement[In]}
\]
is equivalent to saying:

\[
\text{StatementList} : \\
\quad \text{ReturnStatement} \\
\quad \text{ExpressionStatement}_\text{In}
\]

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

\[
\text{VariableDeclaration} : \\
\quad \text{BindingIdentifier} \text{Initializer}_{\text{opt}}
\]

is an abbreviation for:

\[
\text{VariableDeclaration} : \\
\quad \text{BindingIdentifier} \text{BindingIdentifier} \text{Initializer}_{\text{In}}
\]

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:

\[
\text{VariableDeclaration}_? : \\
\quad \text{BindingIdentifier} \text{Initializer}_{?}\text{opt}
\]

is an abbreviation for:

\[
\text{VariableDeclaration} : \\
\quad \text{BindingIdentifier} \text{Initializer}
\]

\[
\text{VariableDeclaration}_{\text{In}} : \\
\quad \text{BindingIdentifier} \text{BindingIdentifier} \text{Initializer}_{\text{In}}
\]

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:

\[
\text{StatementList}_{\text{[return]}} : \\
\quad [+\text{Return}] \text{ReturnStatement} \\
\quad \text{ExpressionStatement}
\]

is an abbreviation for:

\[
\text{StatementList} : \\
\quad \text{ExpressionStatement}
\]

\[
\text{StatementList\_Return} : \\
\quad \text{ReturnStatement} \\
\quad \text{ExpressionStatement}
\]

and that
StatementList::= Return~ Return
ExpressionStatement

is an abbreviation for:

StatementList::= Return
ExpressionStatement
StatementList_Return::= 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 ≠ terminal]" 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. If the set consists of a single terminal the phrase "[lookahead ≠ terminal]" may be used.

For example, given the definitions

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

DecimalDigits::=
DecimalDigit
DecimalDigits DecimalDigit

the definition
LookaheadExample ::
  n [lookahead ϵ {1, 3, 5, 7} \(\frac{DecimalDigits}{\text{DecimalDigit} \ [\text{lookahead} \ \&\ \text{DecimalDigit}]\)}

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

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

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

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

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

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

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

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

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

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

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

```
SourceCharacter ::
  any Unicode code 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 parameterized, in which case the names and usage of the parameters must be provided as part of the algorithm's definition. In order to facilitate their use in multiple parts of this specification, some algorithms, called abstract operations, are named and written in parameterized 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 corresponding 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 an Evaluation algorithm exists of the form:

**Runtime Semantics: Evaluation**

```
Block : { StatementList }
```

1. Return the result of evaluating StatementList.

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

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

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 \( \text{min}(x_1, x_2, \ldots, x_n) \) produces the mathematically smallest of \( x_1 \) through \( x_n \). The mathematical function \( \text{max}(x_1, x_2, \ldots, x_n) \) produces the mathematically largest 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 form 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 \( \text{Contains} \) which takes an argument named \( \text{symbol} \) whose value is a terminal or nonterminal of the grammar that includes the associated production. The default definition of \( \text{Contains} \) is:

1. For each terminal and nonterminal grammar symbol, \( \text{sym} \), in the definition of this production do
   a. If \( \text{sym} \) is the same grammar symbol as \( \text{symbol} \), return \text{true}.
   b. If \( \text{sym} \) is a nonterminal, then
Let contained be the result of sym Contains symbol.

i. If contained is true, return true.

ii. Return false.

The above definition is explicitly overridden 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 associated 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 subclassed 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 ensure that Strings are in a normalized form. Only operations that are explicitly specified to be language or locale sensitive produce language-sensitive results.

NOTE The rationale behind this design was to keep the implementation of Strings as simple and high-performing as possible. If ECMAScript source code is in Normalized Form C, string literals are guaranteed to also be normalized, 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.

Each Symbol value immutably holds an associated value called [[Description]] that 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 (8.1.2.5).

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>&quot;Symbol.create&quot;</td>
<td>A method used to allocate an object. Called from the [[Construct]] internal method.</td>
</tr>
<tr>
<td>@@hasInstance</td>
<td>&quot;Symbol.hasInstance&quot;</td>
<td>A method that determines if a constructor object recognizes an object as one of the constructor's instances. Called by the semantics of the instanceof operator.</td>
</tr>
<tr>
<td>@@isConcatSpreadable</td>
<td>&quot;Symbol.isConcatSpreadable&quot;</td>
<td>A Boolean value that if true indicates that an object should be flatten to its array elements by Array.prototype.concat.</td>
</tr>
<tr>
<td>@@isRegExp</td>
<td>&quot;Symbol.isRegExp&quot;</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>&quot;Symbol.iterator&quot;</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>&quot;Symbol.toPrimitive&quot;</td>
<td>A method that converts an object to a corresponding primitive value. Called by the ToPrimitive abstract operation.</td>
</tr>
<tr>
<td>@@toStringTag</td>
<td>&quot;Symbol.toStringTag&quot;</td>
<td>A string value that is used in the creation of the default string description of an object. Called by the built-in method Object.prototype.toString.</td>
</tr>
<tr>
<td>@@unscopables</td>
<td>&quot;Symbol.unscopables&quot;</td>
<td>An Object whose own property names 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 18,437,368,744,548,106,274 (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 18,437,368,744,548,106,264 (that is, $2^{64} - 2^{53}$) values are called the finite numbers. Half of these are positive numbers and half are negative numbers; for every finite positive Number value there is a corresponding negative value having the same magnitude.
Note that there is both a positive zero and a negative zero. For brevity, these values are also referred to for expository purposes by the symbols \(+0\) and \(-0\), respectively. (Note that these two different zero Number values are produced by the program expressions \(+0\) (or simply \(0\)) and \(-0\).)

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

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

\[ s \times m \times 2^e \]

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

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

\[ s \times m \times 2^e \]

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

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

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

In this specification, the phrase “the Number value for \(x\)” where \(x\) represents an exact nonzero real mathematical quantity (which might even be an irrational number such as \(\pi\)) means a Number value chosen in the following manner. Consider the set of all finite values of the Number type, with \(-0\) removed and with two additional values added to it that are not representable in the Number type, namely \(2^{1024}\) (which is \(+1 \times 2^{31} \times 2^{31}\)) and \(-2^{1024}\) (which is \(-1 \times 2^{31} \times 2^{31}\)). 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.

An integer index is a String-valued property key that is a canonical numeric String (see 7.1.16) and whose numeric value is either +0 or a positive integer ≤ 2^{53}−1. An array index is an integer index whose numeric value i is in the range +0 ≤ i < 2^{32}−1.

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 language type</td>
<td>The value retrieved by a get access of the property.</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 13.6.3.6). 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 3 — Attributes of an Accessor Property

<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 13.6.3.6). 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.

### Table 4 — Default Attribute Values

<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>

### 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 ECMA 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 summarizes 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 parenthesized 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 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)→Undefined or 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 identified 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 identified 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 keys of the string-keyed enumerable properties of the object.</td>
</tr>
<tr>
<td>[[OwnPropertyKeys]]</td>
<td>()→List of propertyKey</td>
<td>Returns a List whose elements are all of the own property keys for the object.</td>
</tr>
</tbody>
</table>

Table 6 summarizes 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 this 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 new operator. The arguments to the internal method are the arguments passed to the new 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 [[Construct]] internal method.</td>
</tr>
</tbody>
</table>

The semantics of the essential internal method for ordinary objects and standard exotic objects are specified in clause 8.5.1. 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.

6.1.7.3 Invariants of the Essential Internal Methods

The Internal Methods of Objects of an ECMAScript engine must conform to the list of invariants specified below. Ordinary ECMAScript Objects as well as all standard exotic objects in this specification maintain these invariants. ECMAScript Proxy objects maintain these invariants by means of runtime checks on the result of traps invoked on the [[ProxyHandler]] object.

Any implementation provided exotic objects must also maintain these invariants for those objects. Violation of these invariants may cause ECMAScript code to have unpredictable behaviour and create security issues. However, violation of these invariants must never compromise the memory safety of an implementation.

Definitions:
- The target of an internal method is the object the internal method is called upon.
- A target is non-extensible if it has been observed to return false from its [[IsExtensible]] internal method, or true from its [[PreventExtensions]] internal method.
- A non-existent property is a property that does not exist as an own property on a non-extensible target.
- All references to SameValue are according to the definition of SameValue algorithm specified in 7.2.3.

[[GetPrototypeOf]] ()
- The Type of the return value must be either Object or Null.
- If target is non-extensible, and [[GetPrototypeOf]] returns a value v, then any future calls to [[GetPrototypeOf]] should return the SameValue as v.
- An object’s prototype chain must have finite length (that is, starting from any object, recursively applying the [[GetPrototypeOf]] internal method to its result must eventually lead to the value null.

[[SetPrototypeOf]] (V)
- The Type of the return value must be Boolean.
- If target is non-extensible, [[SetPrototypeOf]] must return false, unless V is the SameValue as the target’s observed [[GetPrototypeOf]] value.
[[PreventExtensions]] ( )

- The Type of the return value must be Boolean.
- If [[PreventExtensions]] returns true, all future calls to [[IsExtensible]] on the target must return false and the target is now considered non-extensible.

[[GetOwnProperty]] (P)

- The Type of the return value must be either Object or Undefined.
- If the Type of the return value is Object, that object must be a complete property descriptor (see 6.2.4.6).
- If a property is described as a data property and it may return different values over time, then either or both of the Desc.[[Writable]] and Desc.[[Configurable]] attributes must be true even if no mechanism to change the value is exposed via the other internal methods.
- If a property P is described as a data property with Desc.[[Value]] equal to v and Desc.[[Writable]] and Desc.[[Configurable]] are both false, then the SameValue must be returned for the Desc.[[Value]] attribute of the property on all future calls to [[GetOwnProperty]] (P).
- If P’s attributes other than [[Writable]] may change over time or if the property might disappear, then P’s [[Configurable]] attribute must be true.
- If the [[Writable]] attribute may change from false to true, then the [[Configurable]] attribute must be true.
- If the target is non-extensible and P is non-existent, then all future calls to [[GetOwnProperty]] (P) on the target must describe P as non-existent (i.e. [[GetOwnProperty]] (P) must return undefined)

[[DefineOwnProperty]] (P, Desc)

- The Type of the return value must be Boolean.
- [[DefineOwnProperty]] must return false if P has previously been observed as a non-configurable own property of the target, unless either:
  1. P is a non-configurable writable own data property. A non-configurable writable data property can be changed into a non-configurable non-writable data property.
  2. All attributes in Desc are the SameValue as P’s attributes.
- [[DefineOwnProperty]] (P, Desc) must return false if target is non-extensible and P is a non-existent own property. That is, a non-extensible target object cannot be extended with new properties.

[[HasProperty]] (P)

- The Type of the return value must be Boolean.
- If P was previously observed as a non-configurable data or accessor own property of the target, [[HasProperty]] must return true.

[[Get]] (P, Receiver)

- If P was previously observed as a non-configurable, non-writable own data property of the target with value v, then [[Get]] must return the SameValue.
- If P was previously observed as a non-configurable own accessor property of the target whose [[Get]] attribute is undefined, the [[Get]] operation must return undefined.

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

- The Type of the return value must be Boolean.
- If P was previously observed as a non-configurable, non-writable own data property of the target, then [[Set]] must return false unless V is the SameValue as P’s [[Value]] attribute.
- If P was previously observed as a non-configurable own accessor property of the target whose [[Set]] attribute is undefined, the [[Set]] operation must return false.

[[Delete]] (P)
- The Type of the return value must be Boolean.
- If P was previously observed to be a non-configurable own data or accessor property of the target, [[Delete]] must return false.

[[Enumerate]] ()
- The Type of the return value must be Object.

[[OwnPropertyKeys]] ()
- The return value must be a List.
- The Type of each element of the returned List is either String or Symbol.
- The returned List must contain at least the keys of all non-configurable own properties that have previously been observed.
- If the object is non-extensible, the returned List must contain only the keys of all own properties of the object that are observable using [[GetOwnProperty]].

[[Construct]] ()
- The Type of the return value must be Object.

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 8.1.2.5. The well-known intrinsics are listed in Table 7.
### Table 7 — Well-known Intrinsic Objects

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Global Name</th>
<th>ECMAScript Language Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ObjectPrototype%</td>
<td></td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %Object%. (19.1.3)</td>
</tr>
<tr>
<td>%ThrowTypeError%</td>
<td></td>
<td>A function that, when called, throws a TypeError exception.</td>
</tr>
<tr>
<td>%FunctionPrototype%</td>
<td></td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %Function%.</td>
</tr>
<tr>
<td>%Object%</td>
<td>&quot;Object&quot;</td>
<td>The Object constructor (19.1.1)</td>
</tr>
<tr>
<td>%ObjectProto_toString%</td>
<td></td>
<td>The initial value of the &quot;toString&quot; data property of the intrinsic %ObjectPrototype%. (19.1.3.6)</td>
</tr>
<tr>
<td>%Function%</td>
<td>&quot;Function&quot;</td>
<td>The Function constructor (19.2.1)</td>
</tr>
<tr>
<td>%Array%</td>
<td>&quot;Array&quot;</td>
<td>The Array constructor (22.1.1)</td>
</tr>
<tr>
<td>%ArrayPrototype%</td>
<td></td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %Array%.</td>
</tr>
<tr>
<td>%ArrayProto_values%</td>
<td></td>
<td>The initial value of the &quot;values&quot; data property of the intrinsic %ArrayPrototype%. (22.1.3.29)</td>
</tr>
<tr>
<td>%ArrayIteratorPrototype%</td>
<td></td>
<td>The prototype object used for Iterator objects created by the CreateArrayIterator abstract operation.</td>
</tr>
<tr>
<td>%String%</td>
<td>&quot;String&quot;</td>
<td>The String constructor (21.1.1)</td>
</tr>
<tr>
<td>%StringPrototype%</td>
<td></td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %String%.</td>
</tr>
<tr>
<td>%StringIteratorPrototype%</td>
<td></td>
<td>The prototype object used for Iterator objects created by the CreateStringIterator abstract operation.</td>
</tr>
<tr>
<td>%Boolean%</td>
<td>&quot;Boolean&quot;</td>
<td>The initial value of the global object property named &quot;Boolean&quot;.</td>
</tr>
<tr>
<td>%BooleanPrototype%</td>
<td></td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %Boolean%.</td>
</tr>
<tr>
<td>%Number%</td>
<td>&quot;Number&quot;</td>
<td>The initial value of the global object property named &quot;Number&quot;.</td>
</tr>
<tr>
<td>%NumberPrototype%</td>
<td></td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %Number%.</td>
</tr>
<tr>
<td>%Date%</td>
<td>&quot;Date&quot;</td>
<td>The initial value of the global object property named &quot;Date&quot;.</td>
</tr>
<tr>
<td>%DatePrototype%</td>
<td></td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %Date%.</td>
</tr>
<tr>
<td>%RegExp%</td>
<td>&quot;RegExp&quot;</td>
<td>The initial value of the global object property named &quot;RegExp&quot;.</td>
</tr>
<tr>
<td>%RegExpPrototype%</td>
<td></td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic %RegExp%.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><code>%Map%</code></td>
<td>&quot;Map&quot; The initial value of the global object property named &quot;Map&quot;.</td>
<td></td>
</tr>
<tr>
<td><code>%MapPrototype%</code></td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic <code>%Map%</code>.</td>
<td></td>
</tr>
<tr>
<td><code>%MapIteratorPrototype%</code></td>
<td>The prototype object used for Iterator objects created by the CreateMapIterator abstract operation.</td>
<td></td>
</tr>
<tr>
<td><code>%WeakMap%</code></td>
<td>&quot;WeakMap&quot; The initial value of the global object property named &quot;WeakMap&quot;.</td>
<td></td>
</tr>
<tr>
<td><code>%WeakMapPrototype%</code></td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic <code>%WeakMap%</code>.</td>
<td></td>
</tr>
<tr>
<td><code>%Set%</code></td>
<td>&quot;Set&quot; The initial value of the global object property named &quot;Set&quot;.</td>
<td></td>
</tr>
<tr>
<td><code>%SetPrototype%</code></td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic <code>%Set%</code>.</td>
<td></td>
</tr>
<tr>
<td><code>%WeakSet%</code></td>
<td>&quot;WeakSet&quot; The initial value of the global object property named &quot;WeakSet&quot;.</td>
<td></td>
</tr>
<tr>
<td><code>%WeakSetPrototype%</code></td>
<td>The initial value of the &quot;prototype&quot; data property of the intrinsic <code>%WeakSet%</code>.</td>
<td></td>
</tr>
<tr>
<td><code>%SetIteratorPrototype%</code></td>
<td>The prototype object used for Iterator objects created by the CreateSetIterator abstract operation.</td>
<td></td>
</tr>
<tr>
<td>%GeneratorFunction%</td>
<td>The constructor of generator functions.</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------</td>
<td></td>
</tr>
<tr>
<td>%Generator%</td>
<td>The initial value of the <code>prototype</code> property of the %GeneratorFunction% intrinsic</td>
<td></td>
</tr>
<tr>
<td>%GeneratorPrototype%</td>
<td>The initial value of the <code>prototype</code> property of the %Generator% intrinsic</td>
<td></td>
</tr>
<tr>
<td>%Error%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%EvalError%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%RangeError%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ReferenceError%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SyntaxError%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%TypeError%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%URIError%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ErrorPrototype%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%EvalErrorPrototype%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%RangeErrorPrototype%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ReferenceErrorPrototype%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%SyntaxErrorPrototype%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%TypeErrorPrototype%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%URIErrorPrototype%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ArrayBuffer%</td>
<td>The initial value of the &quot;<code>prototype</code>&quot; data property of the intrinsic %ArrayBuffer%.</td>
<td></td>
</tr>
<tr>
<td>%TypedArray%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%TypedArrayPrototype%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%Int8Array%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%Int8ArrayPrototype%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%DataView%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%DataViewPrototype%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ThrowTypeError%</td>
<td>A function object that unconditionally throws a new instance of %TypeError%.</td>
<td></td>
</tr>
<tr>
<td>%Promise%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%PromisePrototype%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%Loader%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%LoaderPrototype%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%LoaderIteratorPrototype%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ReturnUndefined%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%Symbol%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%IteratorPrototype%</td>
<td>An object that all standard built-in iterator objects indirectly inherit from.</td>
<td></td>
</tr>
</tbody>
</table>
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 12.3.6) 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, ```PropertyDescriptor{[[Value]: 42, [[Writable]: false, [[Configurable]: true]}}``` defines a Record value that has three fields each of which is initialized to a specific value. Field name order is not significant. Any fields that are not explicitly listed are considered to be absent.

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

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 8 — Completion Record Fields

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

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:

1. 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 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
1. ReturnIfAbrupt(argument).

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

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

- GetBase(V). Returns the base value component of the reference V.
- GetReferencedName(V). Returns the referenced name component of the reference V.
- IsStrictReference(V). Returns the strict reference 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 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.
      ii. Let base be ToObject(base).
   b. Return the result of calling the [[Get]] internal method of base passing GetReferencedName(V) and GetThisValue(V) as the arguments.
5. 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 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 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.
   a. Return the result of calling the SetMutableBinding (8.1.1) concrete method of base, passing
      GetReferencedName(V), W, and IsStrictReference(V) as arguments.

**NOTE** The object that may be created in step 6.a.i 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. Assert: IsPropertyReference(V) is **true**.
2. If IsSuperReference(V), then
   a. Return the value of the thisValue component of the reference 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. 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.
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. Let obj be ObjectCreate(%ObjectPrototype%).
3. Assert: obj is an extensible ordinary object with no own properties.
4. 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).
5. If Desc has a [[Writable]] field, then
   a. Call OrdinaryDefineOwnProperty with arguments obj, "writable", and PropertyDescriptor([[Value]]: Desc.[[Value]], [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true).
6. 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).
7. 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).
8. 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 }
9. 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 }
10. 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 a new Property Descriptor that initially has no fields.
4. If HasProperty(Obj, "enumerable") is true, then
   a. Let enum be Get(Obj, "enumerable").
   b. ReturnIfAbrupt(enum).
   c. Set the [[Enumerable]] field of desc to ToBoolean(enum).
5. If HasProperty(Obj, "configurable") is true, then
   a. Let conf be Get(Obj, "configurable").
   b. ReturnIfAbrupt(conf).
   c. Set the [[Configurable]] field of desc to ToBoolean(conf).
6. If HasProperty(Obj, "value") is true, then
   a. Let value be Get(Obj, "value").
   b. ReturnIfAbrupt(value).
   c. Set the [[Value]] field of desc to value.
7. If HasProperty(Obj, "writable") is true, then
   a. Let writable be Get(Obj, "writable").
   b. ReturnIfAbrupt(writable).
   c. Set the [[Writable]] field of desc to Boolean(writable).
8. If HasProperty(Obj, "get") is true, then
   a. Let getter be 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 HasProperty(Obj, "set") is true, then
   a. Let setter be 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. Return desc.

6.2.4.6 CompletePropertyDescriptor ( Desc )

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

1. ReturnIfAbrupt(Desc).
2. Assert: Desc is a Property Descriptor
3. Let like be Record{[[Value]]: undefined, [[Writable]]: false, [[Get]]: undefined, [[Set]]: undefined, [[Enumerable]]: false, [[Configurable]]: false}.
4. If either IsGenericDescriptor(Desc) or IsDataDescriptor(Desc) is true, then
   a. If Desc does not have a [[Value]] field, then set Desc.[[Value]] to like.[[Value]].
   b. If Desc does not have a [[Writable]] field, then set Desc.[[Writable]] to like.[[Writable]].
5. Else,
   a. If Desc does not have a [[Get]] field, then set Desc.[[Get]] to like.[[Get]].
   b. If Desc does not have a [[Set]] field, then set Desc.[[Set]] to like.[[Set]].
6. If Desc does not have an [[Enumerable]] field, then set Desc.[[Enumerable]] to like.[[Enumerable]].
7. If Desc does not have a [[Configurable]] field, then set Desc.[[Configurable]] to like.[[Configurable]].
8. 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 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≥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 ≤ fromSize.
5. Let toSize be the number of bytes in toBlock.
6. Assert: toIndex+count ≤ toSize.
7. Repeat, while count>0
7 Abstract Operations

These operations are not a part of the ECMAScript language; they are defined here 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 (input [, PreferredType])

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>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completion Record</td>
<td>If input is an abrupt completion, return input. Otherwise return ToPrimitive(input.[[value]]) also passing the optional hint PreferredType.</td>
</tr>
<tr>
<td>Undefined</td>
<td>Return input (no conversion).</td>
</tr>
<tr>
<td>Null</td>
<td>Return input (no conversion).</td>
</tr>
<tr>
<td>Boolean</td>
<td>Return input (no conversion).</td>
</tr>
<tr>
<td>Number</td>
<td>Return input (no conversion).</td>
</tr>
<tr>
<td>String</td>
<td>Return input (no conversion).</td>
</tr>
<tr>
<td>Symbol</td>
<td>Return input (no conversion).</td>
</tr>
<tr>
<td>Object</td>
<td>Perform the steps following this table.</td>
</tr>
</tbody>
</table>

When Type(input) 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 GetMethod(input, @@toPrimitive).
5. ReturnIfAbrupt(exoticToPrim).
6. If exoticToPrim is not undefined, then
   a. Let result be the result of calling the [[Call]] internal method of exoticToPrim, with input as thisArgument and (hint) as argumentsList.
   b. ReturnIfAbrupt(result).
c. If `Type(result)` is not `Object`, then return `result`.

d. Throw a `TypeError` exception.

7. If `hint` is "default", then, let `hint` be "number".

8. Return `OrdinaryToPrimitive(input, hint)`.

When the abstract operation `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 `methodNames` be the List ("toString", "valueOf").

4. Else,
   a. Let `methodNames` be the List ("valueOf", "toString").

5. For each `name` in `methodNames` in List order, do
   a. Let `method` be `Get(O, name)`.
   b. ReturnIfAbrupt(`method`).
   c. 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`).
      iii. 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 override this behaviour by defining a `@@toPrimitive` method. Of the objects defined in this specification only `Date` objects (see 20.3) and `Symbol` objects (see 19.4.3.5) override the default `ToPrimitive` behaviour. `Date` objects treat no `hint` as if the `hint` were `String`.

### 7.1.2 ToBoolean (argument)

The abstract operation `ToBoolean` converts `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 <code>argument</code> is an abrupt completion, return <code>argument</code>. Otherwise return <code>ToBoolean(argument.[[value]])</code></td>
</tr>
<tr>
<td>Undefined</td>
<td>Return <code>false</code></td>
</tr>
<tr>
<td>Null</td>
<td>Return <code>false</code></td>
</tr>
<tr>
<td>Boolean</td>
<td>Return <code>argument</code> (no conversion).</td>
</tr>
<tr>
<td>Number</td>
<td>Return <code>false</code> if <code>argument</code> is <code>+0</code>, <code>-0</code>, or <code>NaN</code>; otherwise return <code>true</code>.</td>
</tr>
<tr>
<td>String</td>
<td>Return <code>false</code> if <code>argument</code> is the empty string (its length is zero); otherwise return <code>true</code>.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Return <code>true</code></td>
</tr>
<tr>
<td>Object</td>
<td>Return <code>true</code></td>
</tr>
</tbody>
</table>
7.1.3  ToNumber ( argument )

The abstract operation ToNumber converts 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 conversion algorithm below.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Object</td>
<td>Apply the following steps:</td>
</tr>
<tr>
<td></td>
<td>1.  Let primValue be ToPrimitive(argument, hint Number).</td>
</tr>
<tr>
<td></td>
<td>2.  Return ToNumber(primValue)</td>
</tr>
</tbody>
</table>

7.1.3.1  ToNumber Applied to the String Type

ToNumber applied to Strings applies the following grammar to the input String interpreted as a sequence of UTF-16 encoded code points. If the grammar cannot interpret the String as an expansion of StringNumericLiteral, then the result of ToNumber is NaN.

NOTE The terminal symbols of this grammar are all composed of Unicode BMP code points so the result will be NaN if the string contains the UTF-16 encoding of any supplementary code points or any unpaired surrogate code points.

Syntax

StringNumericLiteral ::= StrWhiteSpaceopt StrWhiteSpaceopt StrDecimalLiteral StrWhiteSpaceopt
StrWhiteSpace ::= StrWhiteSpaceChar StrWhiteSpaceopt
StrWhiteSpaceChar ::= WhiteSpace LineTerminator
StringNumericLiteral ::= StrDecimalLiteral HexIntegerLiteral
StrDecimalLiteral ::=
  StrUnsignedDecimalLiteral
  + StrUnsignedDecimalLiteral
  - StrUnsignedDecimalLiteral

StrUnsignedDecimalLiteral ::=
  Infinity
  DecimalDigits . DecimalDigits opt ExponentPart opt
  . DecimalDigits ExponentPart opt
  DecimalDigits ExponentPart opt

DecimalDigits ::=
  DecimalDigit
  DecimalDigits DecimalDigit

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

ExponentPart ::=
  ExponentIndicator SignedInteger

ExponentIndicator :: one of
  e E

SignedInteger ::=
  DecimalDigits
  + DecimalDigits
  - DecimalDigits

HexIntegerLiteral ::=
  0x HexDigit
  0X HexDigit
  HexIntegerLiteral HexDigit

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

NOTE
  Some differences should be noted between the syntax of a StringNumericLiteral and a NumericLiteral (see
  11.8.3):
  • A StringNumericLiteral may include leading and/or trailing 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 include a + or – to indicate its sign.
  • A StringNumericLiteral that is empty or contains only white space is converted to +0.
  • Infinity and –Infinity are recognized as a StringNumericLiteral but not as a NumericLiteral.

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 StrWhiteSpace is the MV of StringNumericLiteral, no matter whether white space is present or not.
The MV of StrNumericLiteral :: StrDecimalLiteral is the MV of StrDecimalLiteral.
The MV of StrNumericLiteral :: StrUnsignedDecimalLiteral is the MV of StrUnsignedDecimalLiteral.
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 \(10^{9999}\) (a value so large that it will round to \(+\infty\)).
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 code points in the second DecimalDigits.
The MV of StrUnsignedDecimalLiteral::: DecimalDigits . DecimalDigits ExponentPart is the MV of DecimalDigits times \(10^{n}\), where \(e\) is the MV of ExponentPart.
The MV of StrUnsignedDecimalLiteral::: DecimalDigits . DecimalDigits ExponentPart is (the MV of the first DecimalDigits plus (the MV of the second DecimalDigits times \(10^{-n}\))) times \(10^{e}\), where \(n\) is the number of code points 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 code points in DecimalDigits.
The MV of StrUnsignedDecimalLiteral::: DecimalDigits ExponentPart is the MV of DecimalDigits times \(10^{n}\), where \(n\) is the number of code points in DecimalDigits and \(e\) is the MV of ExponentPart.
The MV of StrUnsignedDecimalLiteral::: DecimalDigits is the MV of DecimalDigits.
The MV of StrUnsignedDecimalLiteral::: DecimalDigits ExponentPart is the MV of DecimalDigits times \(10^{n}\), where \(n\) is the number of code points in DecimalDigits and \(e\) is the MV of ExponentPart.
The MV of StrUnsignedDecimalLiteral::: DecimalDigits is the MV of DecimalDigits.
The MV of DecimalDigits :: DecimalDigit DecimalDigit is (the MV of DecimalDigits times \(10^{n}\)) plus the MV of DecimalDigit.
The MV of ExponentPart :: ExponentIndicator SignedInteger is the MV of SignedInteger.
The MV of SignedInteger :: DecimalDigits is the MV of DecimalDigits.
The MV of SignedInteger :: + DecimalDigits is the MV of DecimalDigits.
The MV of SignedInteger :: - DecimalDigits is the negative of the MV of DecimalDigits.
The MV of DecimalDigit :: 0 or of HexDigit :: 0 is 0.
The MV of DecimalDigit :: 1 or of HexDigit :: 1 is 1.
The MV of DecimalDigit :: 2 or of HexDigit :: 2 is 2.
The MV of DecimalDigit :: 3 or of HexDigit :: 3 is 3.
The MV of DecimalDigit :: 4 or of HexDigit :: 4 is 4.
The MV of DecimalDigit :: 5 or of HexDigit :: 5 is 5.
The MV of DecimalDigit :: 6 or of HexDigit :: 6 is 6.
The MV of DecimalDigit :: 7 or of HexDigit :: 7 is 7.
The MV of DecimalDigit :: 8 or of HexDigit :: 8 is 8.
The MV of DecimalDigit :: 9 or of HexDigit :: 9 is 9.
The MV of HexDigit :: a or of HexDigit :: A is 10.
• The MV of `HexDigit ::: b` or of `HexDigit ::: B` is 11.
• The MV of `HexDigit ::: c` or of `HexDigit ::: C` is 12.
• The MV of `HexDigit ::: d` or of `HexDigit ::: D` is 13.
• The MV of `HexDigit ::: e` or of `HexDigit ::: E` is 14.
• The MV of `HexDigit ::: f` or of `HexDigit ::: F` is 15.
• The MV of `HexIntegerLiteral ::: 0x HexDigit` is the MV of `HexDigit`.
• The MV of `HexIntegerLiteral ::: 0X HexDigit` is the MV of `HexDigit`.
• The MV of `HexIntegerLiteral ::: HexIntegerLiteral HexDigit` is (the MV of `HexIntegerLiteral` times 16) plus the MV of `HexDigit`.

Once the exact MV for a String numeric literal has been determined, it is then rounded to a value of the Number type. If the MV is 0, then the rounded value is +0 unless the first non-white space code point 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; or
• 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 ( argument )
The abstract operation ToInteger converts argument to an integral numeric value. This abstract operation functions as follows:
1. Let `number` be ToNumber(argument).
2. ReturnIfAbrupt(number).
3. If `number` is NaN, return +0.
4. If `number` is +0, −0, +∞, or −∞, return number.
5. Return the result of computing sign(number) × floor(abs(number)).

7.1.5 ToInt32 ( argument ) — Signed 32 Bit Integer
The abstract operation ToInt32 converts argument to one of $2^{32}$ integer values in the range $-2^{31}$ through $2^{31}-1$, inclusive. This abstract operation functions as follows:
1. Let `number` be ToNumber(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 $2^{32}$.
6. If int32bit ≥ $2^{31}$, return `int32bit` − $2^{32}$, otherwise return `int32bit`.

NOTE
Given the above definition of ToInt32:
• The ToInt32 abstract operation is idempotent: if applied to a result that it produced, the second application leaves that value unchanged.
• ToInt32(ToUint32(x)) is equal to ToInt32(x) for all values of x. (It is to preserve this latter property that +∞ and −∞ are mapped to +0.)
• ToInt32 maps −0 to +0.
7.1.6 ToUint32 (argument) — Unsigned 32 Bit Integer

The abstract operation ToUint32 converts 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 ToNumber(argument).
2. ReturnIfAbrupt(number).
3. If number is NaN, +0, -0, +∞, or -∞, return +0.
4. Let int be sign(number) \times 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 $+\infty$ and $-\infty$ are mapped to $+0$.)
- ToUint32 maps $-0$ to $+0$.

7.1.7 ToInt16 (argument) — Signed 16 Bit Integer

The abstract operation ToInt16 converts 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 ToNumber(argument).
2. ReturnIfAbrupt(number).
3. If number is NaN, +0, -0, +∞, or -∞, return +0.
4. Let int be sign(number) \times 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 (argument) — Unsigned 16 Bit Integer

The abstract operation ToUint16 converts 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 ToNumber(argument).
2. ReturnIfAbrupt(number).
3. If number is NaN, +0, -0, +∞, or -∞, return +0.
4. Let int be sign(number) \times 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 (argument) — Signed 8 Bit Integer

The abstract operation ToInt8 converts 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 ToNumber(argument).
2. ReturnIfAbrupt(number).
3. If number is NaN, +0, −0, +∞, or −∞, return +0.
4. Let int be sign(number) \times \text{floor}(|\text{number}|).
5. Let int\text{8bit} be int modulo 2^8.
6. If int\text{8bit} ≥ 2^7, return int\text{8bit} − 2^8, otherwise return int\text{8bit}.

7.1.10 ToUint8 (argument) — Unsigned 8 Bit Integer

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

1. Let number be ToNumber(argument).
2. ReturnIfAbrupt(number).
3. If number is NaN, +0, −0, +∞, or −∞, return +0.
4. Let int be sign(number) \times \text{floor}(|\text{number}|).
5. Let int\text{8bit} be int modulo 2^8.
6. Return int\text{8bit}.

7.1.11 ToUint8Clamp (argument) — Unsigned 8 Bit Integer, Clamped

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

1. Let number be ToNumber(argument).
2. ReturnIfAbrupt(number).
3. If number ≤ 0, return +0.
4. If number ≥ 255, return 255.
5. Let f be floor(number).
6. If f + 0.5 < number, then return f + 1.
7. If number < f + 0.5, then return f.
8. If f is odd, then return f + 1.
9. Return f.

NOTE: Note that unlike the other ECMAScript integer conversion abstract operation, ToUint8Clamp rounds rather than truncates non-integer values and does not convert +∞ to 0. ToUint8Clamp does “round half to even” tie-breaking. This differs from Math\text{round} which does “round half up” tie-breaking.

7.1.12 ToString (argument) 

The abstract operation ToString converts 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 argument is an abrupt completion, return argument. Otherwise return ToString(argum[nt][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 argument is true, then return &quot;true&quot;. If argument is 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 argument (no conversion)</td>
</tr>
<tr>
<td>Symbol</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Object</td>
<td>Apply the following steps: 1. Let primValue be ToPrimitive(argument, hint String). 2. Return ToString(primValue).</td>
</tr>
</tbody>
</table>

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 +∞, return the String "Infinity".
5. Otherwise, let n, k, and s be integers such that k ≥ 1, 10^{k−1} ≤ s < 10^k, the Number value for s × 10^{−k} is m, and k is as small as possible. Note that k is the number of digits in the decimal representation of s, that s is not divisible by 10, and that the least significant digit of s is not necessarily uniquely determined by these criteria.
6. If k ≤ n ≤ 21, return the String consisting of the code points of the k digits of the decimal representation of s (in order, with no leading zeroes), followed by n−k occurrences of the code point U+0030 (DIGIT ZERO).
7. If 0 ≤ n ≤ 21, return the String consisting of the code points of the most significant n digits of the decimal representation of s, followed by the code point U+002E (FULL STOP), followed by the code points of the remaining k−n digits of the decimal representation of s.
8. If −6 ≤ n ≤ 0, return the String consisting of the code point U+0030 (DIGIT ZERO), followed by the code point U+002E (FULL STOP), followed by −n occurrences of the code point U+0030 (DIGIT ZERO), followed by the code points of the k digits of the decimal representation of s.
9. Otherwise, if k = 1, return the String consisting of the code point of the single digit of s, followed by code point U+0065 (LATIN SMALL LETTER E), followed by the code point U+002B (PLUS SIGN) or the code point U+002D (HYPHEN-MINUS) according to whether n−1 is positive or negative, followed by the code points of the decimal representation of the integer abs(n−1) (with no leading zeroes).
10. Return the String consisting of the code point of the most significant digit of the decimal representation of s, followed by code point U+002E (FULL STOP), followed by the code points of the remaining k−1 digits of the decimal representation of s, followed by code point U+0065 (LATIN SMALL LETTER E), followed by code point U+002B (PLUS SIGN) or the code point U+002D (HYPHEN-MINUS) according to whether n−1 is positive or negative, followed by the code points of the decimal representation of the integer abs(n−1) (with no leading zeroes).
NOTE 1  The following observations may be useful as guidelines for implementations, but are not part of the normative requirements of this Standard:

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

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

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

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


7.1.13 ToObject ( argument )

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

<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([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 ( argument )

The abstract operation ToPropertyKey converts 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 (argument)

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

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

7.1.16 CanonicalNumericIndexString (argument)

The abstract operation CanonicalNumericIndexString returns argument converted to a numeric value if it is a String representation of a Number that would be produced by ToString, or the string "-0". Otherwise, it returns undefined. This abstract operation functions as follows:

1. Assert: Type(argument) is String.
2. If argument is "-0", then return -0.
3. Let n be ToNumber(argument).
4. If SameValue(ToString(n), argument) is false, then return undefined.
5. Return n.

A canonical numeric string is any String value for which the CanonicalNumericIndexString abstraction operation does not return undefined.

7.2 Testing and Comparison Operations

7.2.1 RequireObjectCoercible (argument)

The abstract operation RequireObjectCoercible throws an error if argument is a value that cannot be converted to an Object using ToObject. It is defined by Table 14:

<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 RequireObjectCoercible(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 (argument)

The abstract operation IsCallable determines if 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 \textbf{true}.
b. If $x$ is +0 and $y$ is -0, return \textbf{true}.
c. If $x$ is -0 and $y$ is +0, return \textbf{true}.
d. If $x$ is the same Number value as $y$, return \textbf{true}.
e. Return \textbf{false}.

7. If $\text{Type}(x)$ 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 \textbf{true}; otherwise, return \textbf{false}.
8. If $\text{Type}(x)$ is Boolean, then
   a. If $x$ and $y$ are both \textbf{true} or both \textbf{false}, then return \textbf{true}; otherwise, return \textbf{false}.
9. If $\text{Type}(x)$ is Symbol, then
   a. If $x$ and $y$ are both the same Symbol value, then return \textbf{true}; otherwise, return \textbf{false}.
10. Return \textbf{true} if $x$ and $y$ are the same Object value. Otherwise, return \textbf{false}.

\textbf{NOTE} SameValueZero differs from SameValue only in its treatment of +0 and -0.

\textbf{7.2.5 IsConstructor} (argument)

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

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

\textbf{7.2.6 IsPropertyKey} (argument)

The abstract operation \textbf{IsPropertyKey} determines if 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 \textbf{true}.
3. If Type(argument) is Symbol, return \textbf{true}.
4. Return \textbf{false}.

\textbf{7.2.7 IsExtensible} (O)

The abstract operation \textbf{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 \texttt{[[IsExtensible]]} internal method of $O$.

\textbf{7.2.8 IsInteger} (argument)

The abstract operation \textbf{IsInteger} determines if argument is a finite integer numeric value.

1. ReturnIfAbrupt(argument).
2. If Type(argument) is not Number, return \textbf{false}.
3. If argument is NaN, $\pm\infty$, or $-\infty$, return \textbf{false}.
4. If floor(abs(argument)) $\neq$ abs(argument), then return \textbf{false}.
5. Return \textbf{true}.
7.2.9 Abstract Relational Comparison

The comparison \( x < y \), where \( x \) and \( y \) are values, produces true, false, or undefined (which indicates that at least one operand is NaN). In addition to \( x \) and \( y \) the algorithm takes a Boolean flag named \( \text{LeftFirst} \) 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 \( \text{LeftFirst} \) is true and indicates that the \( x \) parameter corresponds to an expression that occurs to the left of the \( y \) parameter's corresponding expression. If \( \text{LeftFirst} \) is false, the reverse is the case and operations must be performed upon \( y \) before \( x \). Such a comparison is performed as follows:

1. ReturnIfAbrupt(\( x \)).
2. ReturnIfAbrupt(\( y \)).
3. If the \( \text{LeftFirst} \) flag is true, then
   a. Let \( px \) be ToPrimitive(\( x \), hint Number).
   b. ReturnIfAbrupt(\( px \)).
   c. Let \( py \) be ToPrimitive(\( y \), hint Number).
   d. ReturnIfAbrupt(\( py \)).
4. Else the order of evaluation needs to be reversed to preserve left to right evaluation
   a. Let \( py \) be ToPrimitive(\( y \), hint Number).
   b. ReturnIfAbrupt(\( py \)).
   c. Let \( px \) be ToPrimitive(\( x \), hint Number).
   d. ReturnIfAbrupt(\( px \)).
5. If both \( px \) and \( py \) are Strings, then
   a. If \( py \) is a prefix of \( px \), return false. (A String value \( p \) is a prefix of String value \( q \) if \( q \) can be the result of concatenating \( p \) and some other String \( r \). Note that any String is a prefix of itself, because \( r \) may be the empty String.)
   b. If \( px \) is a prefix of \( py \), return true.
   c. Let \( k \) be the smallest nonnegative integer such that the code unit at position \( k \) within \( px \) is different from the code unit 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 at position \( k \) within \( px \).
   e. Let \( n \) be the integer that is the code unit value at position \( k \) within \( py \).
   f. If \( m < n \), return true. Otherwise, return false.
6. Else,
   a. Let \( nx \) be ToNumber(\( px \)). Because \( px \) and \( py \) are primitive values evaluation order is not important.
   b. ReturnIfAbrupt(\( nx \)).
   c. Let \( ny \) be ToNumber(\( py \)).
   d. ReturnIfAbrupt(\( ny \)).
   e. If \( nx \) is NaN, return undefined.
   f. If \( ny \) is NaN, return undefined.
   g. If \( nx \) and \( ny \) are the same Number value, return false.
   h. If \( nx \) is +0 and \( ny \) is −0, return false.
   i. If \( nx \) is −0 and \( ny \) is +0, return false.
   j. If \( nx \) is +∞, return false.
   k. If \( ny \) is +∞, return true.
   l. If \( ny \) is −∞, return false.
   m. If \( nx \) is −∞, return true.
   n. If the mathematical value of \( nx \) is less than the mathematical value of \( ny \) — note that these mathematical values are both finite and not both zero — return true. Otherwise, return false.
NOTE 1  Step 5 differs from step 11 in the algorithm for the addition operator + (12.7.3) 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 normalized 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.

7.2.10 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. ReturnIfAbrupt(\( x \)).
2. ReturnIfAbrupt(\( y \)).
3. If Type(\( x \)) is the same as Type(\( y \)), then
   a. Return the result of performing Strict Equality Comparison \( x === y \).
4. If \( x \) is null and \( y \) is undefined, return true.
5. If \( x \) is undefined and \( y \) is null, return true.
6. If Type(\( x \)) is Number and Type(\( y \)) is String, return the result of the comparison \( x == \) ToNumber(\( y \)).
7. If Type(\( x \)) is String and Type(\( y \)) is Number, return the result of the comparison ToNumber(\( x \)) == \( y \).
8. If Type(\( x \)) is Boolean, return the result of the comparison ToNumber(\( x \)) == \( y \).
9. If Type(\( y \)) is Boolean, return the result of the comparison \( x == \) ToNumber(\( y \)).
10. If Type(\( x \)) is either String, Number, or Symbol and Type(\( y \)) is Object, then return the result of the comparison \( x == \) ToPrimitive(\( y \)).
11. If Type(\( x \)) is Object and Type(\( y \)) is either String, Number, or Symbol, then return the result of the comparison ToPrimitive(\( x \)) == \( y \).
12. Return false.

7.2.11 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 code units (same length and same code units 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 thePropertyDescriptor{[[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 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 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 [[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 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 either undefined or null, then return undefined.
6. If IsCallable(func) is false, then throw a TypeError exception.
7. Return func.

7.3.8 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.

7.3.9 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.

7.3.10 Invoke(O, P, [args])

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

1. Assert: P is a valid property key.
2. If args was not passed, then let args be a new empty List.
3. Let obj be ToObject(O).
4. ReturnIfAbrupt(obj).
5. Let func be the result of calling the [[Get]] internal method of obj passing P and O as the arguments.
6. ReturnIfAbrupt(func).
7. If IsCallable(func) is false, then throw a TypeError exception.
8. Return the result of calling the [[Call]] internal method of func passing O as thisArgument and args as argumentsList.
7.3.11 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 DefinePropertyOrThrow(O, k, PropertyDescriptor{ [[Configurable]]: false }).
      ii. If status is an abrupt completion, then
          1. If pendingException is undefined, then set pendingException to status.
6. Else level is "frozen",
   a. Repeat for each element k of keys,
      i. Let status be the result of calling the [[GetOwnProperty]] internal method of O with k.
      ii. If status is an abrupt completion, then
          1. If pendingException is undefined, then set pendingException to status.
      iii. Else,
          1. Let currentDesc be status.[[value]].
          2. If currentDesc is not undefined, then
             a. If IsAccessorDescriptor(currentDesc) is true, then
                i. Let desc be the PropertyDescriptor{ [[Configurable]]: false }.
             b. Else,
                i. Let desc be the PropertyDescriptor{ [[Configurable]]: false, [[Writable]]: false }.
             c. Let status be 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.

7.3.12 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 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 [[GetOwnProperty]] internal method of O with k.
    b. If status is an abrupt completion, then
i. If `pendingException` is `undefined`, then set `pendingException` to `status`.

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`.

### 7.3.13 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 `ArrayCreate(0)` (see 9.4.2.2).
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`.

### 7.3.14 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. The optional argument `elementTypes` is a List containing the names of ECMAScript Language Types that are allowed for element values of the List that is created. This abstract operation performs the following steps:

1. ReturnIfAbrupt(`obj`).
2. If `elementTypes` was not passed, then let `elementTypes` be (Undefined, Null, Boolean, String, Symbol, Number, Object).
3. If Type(`obj`) is not Object, then throw a `TypeError` exception.
4. Let `len` be Get(`obj`, "length").
5. Let `n` be ToLength(`len`).
6. ReturnIfAbrupt(`n`).
7. Let `list` be an empty List.
8. Let `index` be 0.
9. Repeat while `index` < `n`
   a. Let `indexName` be `ToString(index)`.
   b. Let `next` be Get(`obj`, `indexName`).
   c. ReturnIfAbrupt(`next`).
   d. If Type(`next`) is not an element of `elementTypes`, then throw a `TypeError` exception.
   e. Append `next` as the last element of `list`.
   f. Set `index` to `index` + 1.
10. Return `list`.
7.3.15 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 InstanceofOperator(O, BC) (see 12.9.4).
3. If Type(O) is not Object, return false.
4. Let P be 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.

7.3.16 GetPrototypeOfFromConstructor (constructor, intrinsicDefaultProto)

The abstract operation GetPrototypeOfFromConstructor 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 Get(constructor, "prototype").
4. ReturnIfAbrupt(proto).
5. If Type(proto) is not Object, then
   a. Let realm be GetFunctionRealm(constructor).
   b. 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.

7.3.17 CreateFromConstructor (F)

When the abstract operation CreateFromConstructor is called with Object F the following steps are taken:

1. Assert: Type(F) is Object.
2. Let creator be GetMethod(F, @@create).
3. ReturnIfAbrupt(creator).
4. If creator is undefined, then return undefined.
5. Let obj be the result of calling the [[Call]] internal method of creator with arguments F and an empty List.
6. ReturnIfAbrupt(obj).
7. If Type(obj) is not Object, then throw a TypeError exception.
8. Return obj.

NOTE This operation is equivalent to: F[Symbol.create]() followed by an error check.

7.3.18 Construct (F, argumentsList)

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

1. Assert: Type(F) is Object.
2. Let obj be CreateFromConstructor(F).
3. ReturnIfAbrupt(obj).
4. If obj is undefined, then
   a. Let obj be OrdinaryCreateFromConstructor(F, ObjectPrototype).
   b. ReturnIfAbrupt(obj).
   c. Assert: Type(obj) is Object.
5. Let result be the result of calling the [[Call]] internal method of F, providing obj and argumentsList as the arguments.
6. ReturnIfAbrupt(result).
7. If Type(result) is Object then return result.
8. Return obj.

NOTE This operation is equivalent to: new F(...argumentsList)

7.3.19 GetOption (options, P)

The abstract operation GetOption is used to retrieve the value of a specific property of an object in situation where the object may not be present. The operation is called with arguments options and P where options is the object and P is the property key. This abstract operation performs the following steps:

1. Assert: IsPropertyKey(P) is true.
2. If options is undefined, then return undefined.
3. If Type(options) is not Object, then throw a TypeError exception.
4. Return the result of calling the [[Get]] internal method of options passing P and options as the arguments.

7.3.20 EnumerableOwnNames (O)

When the abstract operation EnumerableOwnNames is called with Object O the following steps are taken:

1. Assert: Type(O) is Object.
2. Let ownKeys be the result of calling the [[OwnPropertyKeys]] internal method of O with no arguments.
3. ReturnIfAbrupt(ownKeys).
4. Let names be a new empty List.
5. Repeat, for each element key of ownKeys in List order
   a. If Type(key) is String, then
      i. Let desc be the result of calling the [[GetOwnProperty]] internal method of O with argument key.
      ii. ReturnIfAbrupt(desc).
      iii. If desc is not undefined, then
         1. If desc.[[Enumerable]] is true, then append key to names.
6. Return names.
7.3.21 GetFunctionRealm ( obj ) Abstract Operation

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

1. Assert: obj is a callable object.
2. If obj has a [[Realm]] internal slot, then
   a. Return obj’s [[Realm]] internal slot.
3. If obj is a Bound Function exotic object, then
   a. Let target be obj’s [[BoundTargetFunction]] internal slot.
   b. Return GetFunctionRealm(target).
4. If obj is a Proxy exotic object, then
   a. Let proxyTarget be the value of obj’s [[ProxyTarget]] internal slot.
   b. If proxyTarget is not null, return GetFunctionRealm(proxyTarget).
5. Return the running execution context’s Realm.

NOTE Step 5 will only be reached if target is a revoked proxy functions or a non-standard exotic function object that does not have a [[Realm]] internal slot.

7.4 Operations on Iterator Objects

See Common Iteration Interfaces (25.1).

7.4.1 CheckIterable ( obj )

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

1. If Type(obj) is not Object, then return undefined.
2. Return Get(obj, @@iterator).

7.4.2 GetIterator ( obj, method )

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

1. ReturnIfAbrupt(obj).
2. If method was not passed, then
   a. Let method be CheckIterable(obj).
   b. ReturnIfAbrupt(method).
3. If IsCallable(method) is false, then throw a TypeError exception.
4. Let iterator be the result of calling the [[Call]] internal method of method with obj as thisArgument and an empty List as argumentsList.
5. ReturnIfAbrupt(iterator).
6. If Type(iterator) is not Object, then throw a TypeError exception.
7. Return iterator.

7.4.3 IteratorNext ( iterator, value )

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

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

7.4.4 IteratorComplete (iterResult)

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

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

7.4.5 IteratorValue (iterResult)

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

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

7.4.6 IteratorStep (iterator)

The abstract operation IteratorStep with argument iterator 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. Let result be IteratorNext(iterator).
2. ReturnIfAbrupt(result).
3. Let done be IteratorComplete(result).
4. ReturnIfAbrupt(done).
5. If done is true, then return false.
6. Return result.

7.4.7 IteratorClose (iterator, completion)

The abstract operation IteratorClose with arguments iterator and completion is used to notify an iterator that should perform any actions it would normally perform when it has reached its completed state:

1. Assert: Type(iterator) is Object.
2. Assert: completion is a Completion Record.
3. Let hasReturn be HasProperty(iterator, "return").
4. ReturnIfAbrupt(hasReturn).
5. If hasReturn is true, then
   a. Let innerResult be Invoke(iterator, "return", ()).
   b. If completion.[[type]] is not throw and innerResult.[[type]] is throw, then .
      i. Return innerResult.
6. Return completion.

7.4.8 CreateIteratorResultObject (value, done)

The abstract operation CreateIteratorResultObject 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 ObjectCreate(%ObjectPrototype%).
3. Perform CreateDataProperty(obj, "value", value).
4. Perform CreateDataProperty(obj, "done", done).
5. Return obj.

7.4.9 CreateListIterator (list)

The abstract operation CreateListIterator with argument list creates an Iterator (25.1.1.2) object whose next method returns the successive elements of list. It performs the following steps:

1. Let iterator be ObjectCreate(\%IteratorPrototype\%, (\[\[IteratorNext\]\], \[\[IteratedList\]\], \[\[ListIteratorNextIndex\]\])).
2. Set iterator’s [[IteratedList]] internal slot to list.
3. Set iterator’s [[ListIteratorNextIndex]] internal slot to 0.
4. Let next be a new built-in function object as defined in ListIterator next (7.4.9.1).
5. Set iterator’s [[IteratorNext]] internal slot to next.
6. Let status be the result of CreateDataProperty(iterator, "next", next).
7. Return iterator.

7.4.9.1 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. Let f be the active function object.
3. If O does not have a [[IteratorNext]] internal slot, then throw a TypeError exception.
4. Let next be the value of the [[IteratorNext]] internal slot of O.
5. If SameValue(f, next) is false, then throw a TypeError exception.
6. If O does not have a [[IteratedList]] internal slot, then throw a TypeError exception.
7. Let list be the value of the [[IteratedList]] internal slot of O.
8. Let index be the value of the [[ListIteratorNextIndex]] internal slot of O.
9. Let len be the number of elements of list.
10. If index ≥ len, then:
    a. Return CreateIterResultObject(\undefined, true).
11. Set the value of the [[ListIteratorNextIndex]] internal slot of O to index+1.
12. Return CreateIterResultObject(list[index], false).

NOTE A ListIterator next method will throw an exception if applied to any object other than the one with which it was originally associated.

7.4.10 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 CreateListIterator(empty).

7.4.11 CreateCompoundIterator (iterator1, iterator2)

The abstract operation CreateCompoundIterator with arguments iterator1 and iterator2 creates an Iterator (25.1.1.2) object whose next method returns the successive elements of iterator1 followed by the successive elements of iterator2. It performs the following steps:
1. Let iterator be ObjectCreate(%IteratorPrototype%, %[[Iterator1]], %[[Iterator2]], %[[State]], %[[IteratorNext]]).
2. Set iterator's %[[Iterator1]] internal slot to iterator1.
3. Set iterator's %[[Iterator2]] internal slot to iterator2.
4. Set iterator's %[[State]] internal slot to 1.
5. Let next be a new built-in function object as defined in CompoundIterator next (7.4.11.1).
6. Set iterator's %[[IteratorNext]] internal slot to next.
7. Let status be the result of CreateDataProperty(iterator, "next", next).
8. Return iterator.

7.4.11.1 CompoundIterator next()

The CompoundIterator next method is a standard built-in function object that performs the following steps:

1. Let O be the this value.
2. Let f be the active function object.
3. If O does not have a %[[IteratorNext]] internal slot, then throw a TypeError exception.
4. Let next be the value of the %[[IteratorNext]] internal slot of O.
5. If SameValue(f, next) is false, then throw a TypeError exception.
6. If O does not have a %[[Iterator1]] internal slot, then throw a TypeError exception.
7. Assert: O is an object created and initialized by CreateCompoundIterator.
8. Let state be the value of O's %[[State]] internal slot.
9. If state = 1, then
   a. Let iterator1 be the value of O's %[[Iterator1]] internal slot.
   b. Let result1 be IteratorStep(iterator1).
   c. If result1 is not false, then,
      i. Return result1.
   d. Set O's %[[State]] internal slot to 2.
10. Let iterator2 be the value of O's %[[Iterator2]] internal slot.
11. Return IteratorNext(iterator2).

NOTE A CompoundIterator next method will throw an exception if applied to any object other than the one with which it was originally associated.

7.5 Operations on Promise Objects

Promise Objects (25.4) serve as a placeholder for the eventual result of a deferred (and possibly asynchronous) computation.

Within this specification the adjective "eventual" mean a value or a Promise object that will ultimately resolves to the value. For example, “Returns an eventual String” is equivalent to “Returns either a String or a Promise object that will eventually resolves to a String”. A "resolved value" is the final value of an "eventual value".

NOTE The Promise related abstract operations defined in this subclause are used by specification algorithms when they perform or respond to asynchronous operations. They ensure that the actual built-in Promise operations are used by the algorithms, even if ECMAScript code has modified the properties of %Promise% or %PromisePrototype%.
7.5.1 PromiseNew (executor) Abstract Operation

The abstract operation PromiseNew allocates and initializes a new promise object for use by specification algorithm. The executor argument initiates the deferred computation.

1. Let promise be AllocatePromise(%Promise%).
2. Return InitializePromise(promise, executor).

7.5.2 PromiseBuiltinCapability () Abstract Operation

The abstract operation PromiseBuiltinCapability allocates a PromiseCapability record (25.4.1.1) for a builtin promise object for use by specification algorithm.

1. Let promise be AllocatePromise(%Promise%).
2. Return CreatePromiseCapabilityRecord(promise, %Promise%).

NOTE This abstract operation is the same as the default built-in behaviour of NewPromiseCapability abstract operation (25.4.1.4).

7.5.3 PromiseOf (value) Abstract Operation

The abstract operation PromiseOf returns a new Promise that resolves to the argument value.

1. Assert: IsPromise(value) is false.
2. Let capability be PromiseBuiltinCapability().
3. ReturnIfAbrupt(capability).
4. Let resolveResult be the result of calling the [[Call]] internal method of capability.[[Resolve]] with undefined as thisArgument and (value) as argumentsList.
5. ReturnIfAbrupt(resolveResult).
6. Return capability.[[Promise]].

NOTE This abstract operation is the same as the default built-in behaviour of the Promise.resolve method (25.4.4.5). However, PromiseOf does not accept a Promise as its argument.

7.5.4 PromiseAll (promiseList) Abstract Operation

7.5.5 PromiseCatch (promise, rejectedAction) Abstract Operation

7.5.6 PromiseThen (promise, resolvedAction, rejectedAction) Abstract Operation

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 `FunctionEnvironment` 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 <strong>true</strong> if it does and <strong>false</strong> 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 uninitialized mutable binding in an environment record. The String value ( N ) is the text of the bound name. If the optional Boolean argument ( D ) is <strong>true</strong> the binding is may be subsequently deleted.</td>
</tr>
<tr>
<td>CreateImmutableBinding(N)</td>
<td>Create a new but uninitialized immutable binding in an environment record. The String value ( N ) is the text of the bound name.</td>
</tr>
<tr>
<td>InitializeBinding(N,V)</td>
<td>Set the value of an already existing but uninitialized 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 <strong>true</strong> and the binding cannot be set throw a <strong>TypeError</strong> 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 <strong>true</strong> and the binding does not exist throw a <strong>ReferenceError</strong> exception. If the binding exists but is uninitialized a <strong>ReferenceError</strong> is thrown, regardless of the value of ( S ).</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 <strong>true</strong>. If the binding exists but cannot be removed return <strong>false</strong>. If the binding does not exist return <strong>true</strong>.</td>
</tr>
<tr>
<td>HasThisBinding()</td>
<td>Determine if an environment record establishes a <code>this</code> binding. Return <strong>true</strong> if it does and <strong>false</strong> if it does not.</td>
</tr>
<tr>
<td>HasSuperBinding()</td>
<td>Determine if an environment record establishes a <code>super</code> method binding. Return <strong>true</strong> if it does and <strong>false</strong> if it does not.</td>
</tr>
<tr>
<td>WithBaseObject ()</td>
<td>If this environment record is associated with a <code>with</code> statement, return the with object. Otherwise, return <strong>undefined</strong>.</td>
</tr>
</tbody>
</table>

### 8.1.1 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 \( \text{envRec} \) be the declarative environment record for which the method was invoked.
2. If \( \text{envRec} \) has a binding for the name that is the value of \( N \), return \text{true}.
3. Return \text{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 uninitialized. A binding must not already exist in this Environment Record for \( N \). If Boolean argument \( D \) is provided and has the value \text{true} the new binding is marked as being subject to deletion.

1. Let \( \text{envRec} \) be the declarative environment record for which the method was invoked.
2. Assert: \( \text{envRec} \) does not already have a binding for \( N \).
3. Create a mutable binding in \( \text{envRec} \) for \( N \) and record that it is uninitialized. If \( D \) is \text{true} record that the newly created binding may be deleted by a subsequent DeleteBinding call.
4. Return NormalCompletion(\text{empty}).

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 uninitialized. A binding must not already exist in this environment record for \( N \).

1. Let \( \text{envRec} \) be the declarative environment record for which the method was invoked.
2. Assert: \( \text{envRec} \) does not already have a binding for \( N \).
3. Create an immutable binding in \( \text{envRec} \) for \( N \) and record that it is uninitialized.

8.1.1.1.4 InitializeBinding(N,V)

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

1. Let \( \text{envRec} \) be the declarative environment record for which the method was invoked.
2. Assert: \( \text{envRec} \) must have an uninitialized binding for \( N \).
3. Set the bound value for \( N \) in \( \text{envRec} \) to \( V \).
4. Record that the binding for \( N \) in \( \text{envRec} \) has been initialized.

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 \text{TypeError} is thrown if \( S \) is \text{true}.

1. Let \( \text{envRec} \) 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 initialized 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(empty)`.

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`. If the binding exists but is uninitialized a `ReferenceError` is thrown, regardless of the value of `S`.

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 uninitialized binding, then throw a `ReferenceError` exception.
4. 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 `envRec` be the object environment record for which the method was invoked.
2. Let `bindings` be the binding object for `envRec`.
3. Let `foundBinding` be HasProperty(`bindings`, `N`).
4. ReturnIfAbrupt(`foundBinding`).
5. If `foundBinding` is `false`, then return `false`.
6. If the `withEnvironment` flag of `envRec` is `false`, then return `true`.
7. Let `unscopables` be Get(`bindings`, @@unscopables`).
8. ReturnIfAbrupt(`unscopables`).
9. If Type(`unscopables`) is Object, then
   a. Let `blocked` be Get(`unscopables`, `N`).
   b. ReturnIfAbrupt(`blocked`).
   c. If `blocked` is not `undefined`, then return `false`.
10. Return `true`.

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

**NOTE** Normally `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 **InitializeBinding** \((N,V)\)

The concrete Environment Record method InitializeBinding for object environment records is used to set the bound value of the current binding of the identifier whose name is the value of the argument \(N\) to the value of argument \(V\). An uninitialized binding for \(N\) must already exist.

1. Let envRec be the object environment record for which the method was invoked.
2. Assert: envRec must have an uninitialized binding for \(N\).
3. Record that the binding for \(N\) in envRec has been initialized.
4. Return the result of calling the SetMutableBinding concrete method of envRec with \(N\), \(V\), 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 envRec be the object environment record for which the method was invoked.
2. Let bindings be the binding object for envRec.
3. Return \(\text{Put}(\text{bindings}, N, V, S)\).

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 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 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 outermost 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>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GetThisBinding()</code></td>
<td>Return the value of this environment record’s <code>this</code> binding.</td>
</tr>
<tr>
<td><code>GetSuperBase()</code></td>
<td>Return the object that is the base for <code>super</code> property accesses bound in this environment record. The object is derived from this environment record’s <code>HomeObject</code> binding. If the value is <code>Empty</code>, return <code>undefined</code>.</td>
</tr>
<tr>
<td><code>GetMethodName()</code></td>
<td>Return the value of this environment record’s <code>MethodName</code> 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 (8.1.2.5). A global environment record 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`, `GeneratorDeclaration`, 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`, `GeneratorDeclaration`, or `VariableDeclaration` 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>ObjectRecord</td>
<td>An Object Environment Record whose base object is the global object. It contains global built-in bindings as well as FunctionDeclaration, GeneratorDeclaration, and VariableDeclaration bindings in global code for the associated Realm.</td>
</tr>
<tr>
<td>DeclarativeRecord</td>
<td>A Declarative Environment Record that contains bindings for all declarations in global code for the associated Realm code except for FunctionDeclaration, GeneratorDeclaration, and VariableDeclaration bindings.</td>
</tr>
<tr>
<td>VarNames</td>
<td>A List containing the string names bound by FunctionDeclaration, GeneratorDeclaration, and VariableDeclaration 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 VariableDeclaration, FunctionDeclaration, or GeneratorDeclaration.</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 ObjectRecord component of a global 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. 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>
</tbody>
</table>
| CreateGlobalFunctionBinding(N, V, D) | Used to create and initialize global function bindings in the ObjectRecord component of a global 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. 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 followed by a SetMutableBinding but it allows function declarations to
receive special treatment.

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

8.1.1.4.1 HasBinding(N)

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

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

8.1.1.4.2 CreateMutableBinding (N, D)

The concrete environment record method CreateMutableBinding for global environment records creates a new mutable binding for the name N that is uninitialized. The binding is created in the associated DeclarativeRecord. A binding for N must not already exist in the DeclarativeRecord. If Boolean argument D is provided and has the value true the new binding is marked as being subject to deletion.

1. Let envRec be the global environment record for which the method was invoked.
2. Let DclRec be envRec's DeclarativeRecord.
3. Let alreadyThere be the result of calling the HasBinding concrete method of DclRec with argument N.
4. ReturnIfAbrupt(alreadyThere).
5. If alreadyThere is true, then throw a TypeError exception.
6. Return the result of calling the CreateMutableBinding concrete method of DclRec with arguments N and D.

8.1.1.4.3 CreateImmutableBinding [N]

The concrete Environment Record method CreateImmutableBinding for global environment records creates a new immutable binding for the name N that is uninitialized. 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 DeclarativeRecord.
3. Let alreadyThere be the result of calling the HasBinding concrete method of DclRec with argument N.
4. ReturnIfAbrupt(alreadyThere).
5. If alreadyThere is true, then throw a TypeError exception.
6. Return the result of calling the CreateImmutableBinding concrete method of DclRec with argument N.

8.1.1.4.4 InitializeBinding (N,V)

The concrete Environment Record method InitializeBinding for global environment records is used to set the bound value of the current binding of the identifier whose name is the value of the argument N to the value of argument V. An uninitialized 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 DeclarativeRecord.
3. If the result of calling $DclRec$'s HasBinding concrete method with argument $N$ is $true$, then
   a. Return the result of calling $DclRec$'s InitializeBinding concrete method with arguments $N$ and $V$.
4. Assert: If the binding exists it must be in the object environment record.
5. Let $ObjRec$ be $envRec$'s ObjectRecord.
6. Return the result of calling $ObjRec$'s InitializeBinding concrete method with arguments $N$ and $V$.

**8.1.1.4.5 SetMutableBinding (N,V,S)**

The concrete Environment Record method SetMutableBinding for global environment records attempts to change the bound value of the current binding of the identifier whose name is the value of the argument $N$ to the value of argument $V$. If the binding is an immutable binding, a TypeError is thrown if $S$ is $true$. A property named $N$ normally already exists but if it does not or is not currently writable, error handling is determined by the value of the Boolean argument $S$.

1. Let $envRec$ be the global environment record for which the method was invoked.
2. Let $DclRec$ be $envRec$'s DeclarativeRecord.
3. If the result of calling $DclRec$'s HasBinding concrete method with argument $N$ is $true$, then
   a. Return the result of calling the SetMutableBinding concrete method of $DclRec$ with arguments $N$, $V$, and $S$.
4. Let $ObjRec$ be $envRec$'s ObjectRecord.
5. Return the result of calling the SetMutableBinding 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 the binding is an uninitialized 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 global environment record for which the method was invoked.
2. Let $DclRec$ be $envRec$'s DeclarativeRecord.
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 $N$, $V$, and $S$.
4. Let $ObjRec$ be $envRec$'s ObjectRecord.
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 global environment record for which the method was invoked.
2. Let $DclRec$ be $envRec$'s DeclarativeRecord.
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 ObjectRecord.
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 ObjectRecord.
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 DelRec be envRec's DeclarativeRecord.
3. Return the result of calling DelRec'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 ObjectRecord.
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 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 ObjectRecord.
3. Let globalObject be the binding object for ObjRec.
4. Let extensible be 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. ReturnIfAbrupt(existingProp).
9. If existingProp is undefined, then return extensible.
10. If existingProp.[[Configurable]] is true, then return true.
11. If IsDataDescriptor(existingProp) is true and existingProp has attribute values {[[Writable]]: true, [[Enumerable]]: true}, then return true.
12. 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 global environment record for which the method was invoked.
2. Let ObjRec be envRec’s ObjectRecord.
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).

Commented [AWB1111]: Carry over from ES5, but perhaps unnecessary
8.1.4.17 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 global environment record for which the method was invoked.
2. Let ObjRec be envRec's ObjectRecord.
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. ReturnIfAbrupt(existingProp).
6. If existingProp is undefined or existingProp.[[Configurable]] is true, then
   a. Let desc be the PropertyDescriptor([[Value]]:V, [[Writable]]:true, [[Enumerable]]:true, [[Configurable]]:D).
7. Else,
   a. Let desc be the PropertyDescriptor([[Value]]:V).
8. Let status be DefinePropertyOrThrow(globalObject, N, desc).
9. ReturnIfAbrupt(status).
10. Let varDeclaredNames be envRec's VarNames List.
11. If varDeclaredNames does not contain the value of N, then
    a. Append N to varDeclaredNames.
12. 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) Abstract Operation

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 concrete method of envRec passing name as the argument.
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 GetIdentifierReference(outer, name, strict).
8.1.2.2 NewDeclarativeEnvironment (E) Abstract Operation

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) Abstract Operation

When the abstract operation NewObjectEnvironment is called with an Object O and a Lexical Environment E (or null) as arguments, the following steps are performed:
   1. Let env be a new Lexical Environment.
   2. Let envRec be a new object environment record containing O as the binding object.
   3. Set env’s environment record to envRec.
   4. Set the outer lexical environment reference of env to E.
   5. Return env.

8.1.2.4 NewFunctionEnvironment (F, T) Abstract Operation

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’s [[NeedsSuper]] internal slot is true, then
      a. Let home be the value of F’s [[HomeObject]] internal slot.
      b. If home is undefined, then throw a ReferenceError exception.
      c. Set envRec’s HomeObject to home.
      d. 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 [[Environment]] internal slot.

8.1.2.5 NewGlobalEnvironment (G) Abstract Operation

When the abstract operation NewGlobalEnvironment is called with an ECMAScript Object G as its argument, the following steps are performed:
   1. Let env be a new Lexical Environment.
   2. Let objRec be a new object environment record containing G as the binding object.
   3. Set objRec’s unscopables to an empty List.
   4. Let delRec be a new declarative environment record containing no bindings.
   5. Let globalRec be a new global environment record.
   6. Set globalRec’s ObjectRecord to objRec.
   7. Set globalRec’s DeclarativeRecord to delRec.
8. Set `globalRec`'s VarNames to a new empty List.
9. Set `env`'s environment record to `globalRec`.
10. Set the outer lexical environment reference of `env` to `null`.

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>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[intrinsics]]</code></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><code>[[globalThis]]</code></td>
<td>An object</td>
<td>The global object for this Realm</td>
</tr>
<tr>
<td><code>[[globalEnv]]</code></td>
<td>An ECMAScript environment</td>
<td>The global environment for this Realm</td>
</tr>
<tr>
<td><code>[[directEvalTranslate]]</code></td>
<td><code>undefined</code> or an object that is callable as a function.</td>
<td></td>
</tr>
<tr>
<td><code>[[nonEvalFallback]]</code></td>
<td><code>undefined</code> or an object that is callable as a function.</td>
<td></td>
</tr>
<tr>
<td><code>[[indirectEval]]</code></td>
<td><code>undefined</code> or an object that is callable as a function.</td>
<td></td>
</tr>
<tr>
<td><code>[[loader]]</code></td>
<td>any ECMAScript identifier or <code>empty</code></td>
<td>The Loader object that can associate ECMAScript code with this Realm</td>
</tr>
</tbody>
</table>
4. Set `intrinsic. [[%ObjectPrototype%]]` to `objProto`.
5. Let `throwerSteps` be the algorithm steps of the `%ThrowTypeError%` function (9.2.8.1).
6. Let `thrower` be `CreateBuiltinFunction(realmRec, throwerSteps, null)`.
7. Set `intrinsic. [[%ThrowTypeError%]]` to `thrower`.
8. Let `noSteps` be an empty sequence of algorithm steps.
9. Let `funcProto` be `CreateBuiltinFunction(realmRec, noSteps, objProto)`.
10. Set `intrinsic. [[%FunctionPrototype%]]` to `funcProto`.
11. Call the `[[SetPrototypeOf]]` internal method of `thrower` with argument `funcProto`.
12. Perform `AddRestrictedFunctionProperties(funcProto, realmRec)`.
13. Set fields of `intrinsic` with the values listed in Table 7 that have not already been handled above. The field names are the names listed in column one of the table. The value of each field is a new object value fully and recursively populated with property values as defined by the specification of each object in clauses 18-26. All object property values are newly created object values. All values that are built-in function objects are created by performing `CreateBuiltinFunction(realmRec, <steps>, <prototype>, <slots>)` where `<steps>` is the definition of that function provided by this specification, `<prototype>` is the specified value of the function’s `[[Prototype]]` internal slot and `<slots>` is a list of the names, if any, of the functions specified internal slots. The creation of the intrinsics and their properties must be ordered to avoid any dependencies upon objects that have not yet been created.

### 8.2.3 SetRealmGlobalObj (realmRec, globalObj) Abstract Operation

The abstract operation `SetRealmGlobalObj` with arguments `realmRec` and `globalObj` performs the following steps:

1. If `globalObj` is `undefined`, then
   a. Let `intrinsic` be `realmRec. [[intrinsic]]`.
   b. Let `globalObj` be `ObjectCreate(intrinsic. [[%ObjectPrototype%]])`.
2. Assert: `Type(globalObj)` is `Object`.
3. Set `realmRec. [[globalThis]]` to `new Global`.
4. Let `newGlobalEnv` be `NewGlobalEnvironment(new Global)`.
5. Set `realmRec. [[globalEnv]]` to `newGlobalEnv`.
6. Return `realmRec`.

### 8.2.4 SetDefaultGlobalBindings (realmRec) Abstract Operation

The abstract operation `SetDefaultGlobalBindings` with argument `realmRec` performs the following steps:

1. Let `global` be `realmRec. [[globalThis]]`.
2. For each property of the Global Object specified in clause 18, do
   a. Let `name` be the string value of the property name.
   b. Let `desc` be the fully populated data property descriptor for the property containing the specified attributes for the property. For properties whose values are functions, the value of the `[[Value]]` attribute is the corresponding intrinsic function object from `realmRec`.
   c. Let `status` be `DefinePropertyOrThrow(global, name, desc)`.
   d. ReturnIfAbrupt(status).
3. Return `global`.

### 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 at least 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>Function</td>
<td>If this execution context is evaluating the code of a function object, then the value of this is the function. If the context is evaluating the code of a <code>Script</code> the value is <code>null</code>.</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 later 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 of the Realm component of the running execution context is also called the *current Realm*. The value of the Function component of the running execution context is also called the *active Function*.

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

### Table 22 — Additional State Components for ECMAScript Code Execution Contexts

<table>
<thead>
<tr>
<th>Component</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>LexicalEnvironment</td>
<td>Identifies the Lexical Environment used to resolve identifier references made by code within this execution context.</td>
</tr>
<tr>
<td>VariableEnvironment</td>
<td>Identifies the Lexical Environment whose environment record holds bindings created by <code>VariableStatements</code> within this execution context.</td>
</tr>
</tbody>
</table>

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>

Table 23 — Additional State Components for Generator Execution Contexts

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 ECMAScript code to directly access or observe an execution context.

8.3.1 ResolveBinding ( name ) Abstract Operation

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 GetIdentifierReference(env, name, strict).

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

8.3.2 GetThisEnvironment () Abstract Operation

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 () Abstract Operation

The abstract operation ResolveThisBinding determines the binding of the keyword this using the LexicalEnvironment of the running execution context. ResolveThisBinding performs the following steps:

1. Let env be GetThisEnvironment( ).
2. Return the result of calling the GetThisBinding concrete method of env.
8.3.4 GetGlobalObject ( ) Abstract Operation

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 \( currentRealm.[[globalThis]] \).

8.4 Jobs and Job Queues

A Job is an abstract operation that initiates an ECAMScript computation when no other ECAMScript computation is currently in progress. A Job abstract operation may be defined to accept an arbitrary set of job parameters.

Execution of a Job can be initiated only when there is no running execution context and the execution context stack is empty. A PendingJob is a request for the future execution of a Job. A PendingJob is an internal Record whose fields are specified in Table 24.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{Job}])</td>
<td>The name of a Job abstract operation</td>
<td>This is the abstract operation that is performed when execution of this PendingJob is initiated. Jobs are abstract operations that use NextJob rather than Return to indicate that they have completed.</td>
</tr>
<tr>
<td>([\text{Arguments}])</td>
<td>A List.</td>
<td>The List of argument values that are to be passed to ([\text{Job}]) when it is activated.</td>
</tr>
<tr>
<td>([\text{Realm}])</td>
<td>A Realm Record</td>
<td>The Realm for the initial execution context when this Pending Job is initiated.</td>
</tr>
<tr>
<td>([\text{HostDefined}])</td>
<td>Any, default value is undefined.</td>
<td>Field reserved for use by host environments that need to associate additional information with a pending Job.</td>
</tr>
</tbody>
</table>

A Job Queue is a FIFO queue of PendingJob records. Each Job Queue has a name and the full set of available Job Queues are defined by an ECAMScript implementation. Every ECAMScript implementation has at least the Job Queues defined in Table 25.

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScriptJobs</td>
<td>Jobs that validate and evaluate ECAMScript ( \text{Script} ) and ( \text{Module} ) code units. See clauses 10 and 15.</td>
</tr>
<tr>
<td>PromiseJobs</td>
<td>Jobs that are responses to the settlement of a Promise (see 25.4).</td>
</tr>
</tbody>
</table>

A request for the future execution of a Job is made by enqueueing, on a Job Queue, a PendingJob record that includes a Job abstract operation name and any necessary argument values. When there is no running execution context and the execution context stack is empty, the ECAMScript implementation removes the first PendingJob from a Job Queue and uses the information contained in it to create an execution context and starts execution of the associated Job abstract operation.
The PendingJob records from a single Job Queue are always initiated in FIFO order. This specification does not define the order in which multiple Job Queues are serviced. An ECMAScript implementation may interweave the FIFO evaluation of the PendingJob records of a Job Queue with the evaluation of the PendingJob records of one or more other Job Queues. An implementation must define what occurs when there are no running execution context and all Job Queues are empty.

NOTE Typically an ECMAScript implementation will have its Job Queues pre-initialized with at least one PendingJob and one of those Jobs will be the first to be executed. An implementation might choose to free all resources and terminate if the current Job completes and all Job Queues are empty. Alternatively, it might choose to wait for a some implementation specific agent or mechanism to enqueue new PendingJob requests.

The following abstract operations are used to create and manage Jobs and Job Queues:

8.4.1 EnqueueJob (queueName, job, arguments) Abstract Operation

The abstract operation requires three arguments: queueName, job, and arguments. It performs the following steps:

1. Assert: Type(queueName) is String and its value is the name of a Job Queue recognized by this implementation.
2. Assert: job is the name of a Job.
3. Assert: arguments is a List that has the same number of elements as the number of parameters required by job.
4. Let callerContext be the running execution context.
5. Let callerRealm be callerContext’s Realm.
6. Let pending be PendingJob{ [[Job]]: job, [[Arguments]]: arguments, [[Realm]]: callerRealm, [[HostDefined]]: undefined }.
7. Perform any implementation or host environment defined processing of pending. This may including modify the [[HostDefined]] field or any other field of pending.
8. Add pending at the back of the Job Queue named by queueName.
9. Return NormalCompletion(empty).

8.4.2 NextJob result

An algorithm step such as:

1. NextJob result.

is used in Job abstract operations in place of:

1. Return result.

Job abstract operations must not contain a Return step or a ReturnIfAbrupt step. The NextJob result operation is equivalent to the following steps:

1. If result is an abrupt completion, then perform implementation defined unhandled exception processing.
2. Suspend the running execution context and remove it from the execution context stack.
3. Assert: The execution context stack is now empty.
4. Let nextQueue be a non-empty Job Queue chosen in an implementation defined manner. If all Job Queues are empty, the result is implementation defined.
5. Let nextPending be the PendingJob record at the front of nextQueue. Remove that record from nextQueue.
6. Let newContext be a new execution context.
7. Set newContext’s Realm to nextPending.\([\text{Realm}]\).
8. Push newContext onto the execution context stack; newContext is now the running execution context.
9. Perform any implementation or host environment defined job initialization using nextPending.
10. Perform the abstract operation named by nextPending.\([\text{Job}]\) using the elements of nextPending.\([\text{Arguments}]\) as its arguments.

8.5 Initialization

An ECMAScript implementation performs the following steps prior to the execution of any Jobs or the evaluation of any ECMAScript code:

1. Let realm be CreateRealm().
2. Let newContext be a new execution context.
3. Set the Function of newContext to null.
4. Set the Realm of newContext to realm.
5. Push newContext onto the execution context stack; newContext is now the running execution context.
6. Let status be InitializeFirstRealm(realm).
7. If status is an abrupt completion, then
   a. Assert: The first realm could not be created.
   b. Terminate ECMAScript execution.
8. In an implementation dependent manner, obtain the SourceCharacter sequence (see 10) for zero or more ECMAScript scripts. For each such sequence source do,
   a. EnqueueJob("ScriptJob", ScriptEvaluationJob, (source)),
9. NextJob NormalCompletion(\text{undefined}).

8.5.1 InitializeFirstRealm ( realm ) Abstract Operation

The abstract operation InitializeFirstRealm with parameter realm performs the following steps:

1. Let intrinsics be CreateIntrinsics(realm).
2. If this implementation requires use of an exotic object to serve as realm’s global object, then let global be such an object created in an implementation defined manner. Otherwise, let global be \text{undefined} indicating that an ordinary object should be created as the global object.
3. Perform SetRealmGlobalObject(realm, global).
4. Let globalObj be SetDefaultGlobalBindings(realm).
5. ReturnIfAbrupt(globalObj).
6. Create any implementation defined global object properties on globalObj.
7. Return NormalCompletion(\text{undefined}).

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

Commented \([\text{AWB2514}]\): TODO: probably also need to obtain and schedule any module load tasks.
properties may not be added to the object. In addition, if [[Extensible]] is false the value of the [[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 to true.

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 null, then
   a. Let p be V.
   b. Repeat, while p is not null
      i. If SameValue(p, O) is true, then return false.
      ii. Let nextp be the result of calling the [[GetPrototypeOf]] internal method of p with no arguments.
      iii. ReturnIfAbrupt(nextp).
   c. Let p be nextp.
7. Let extensible be the value of the [[Extensible]] internal slot of O.
8. If extensible is false, then
   a. Let current2 be the value of the [[Prototype]] internal slot of O.
   b. If SameValue(V, current2) is true, then return true.
   c. Return false.
9. Set the value of the [[Prototype]] internal slot of O to V.
10. Return true.

9.1.3 [[IsExtensible]]()

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

1. Return the value of the [[Extensible]] internal slot of O.

9.1.4 [[PreventExtensions]]()

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

1. Set the value of the [[Extensible]] internal slot of O to false.
2. Return true.
9.1.5 [[GetOwnProperty]] (P)

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

1. Return OrdinaryGetOwnProperty(O, P).

9.1.5.1 OrdinaryGetOwnProperty (O, P)

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

1. Assert: IsPropertyKey(P) is true.
2. If O does not have an own property with key P, return undefined.
3. Let D be a newly created Property Descriptor with no fields.
4. Let X be O’s own property whose key is P.
5. If X is a data property, then
   a. Set D.[[Value]] to the value of X’s [[Value]] attribute.
   b. Set D.[[Writable]] to the value of X’s [[Writable]] attribute.
6. Else X is an accessor property, so
   a. Set D.[[Get]] to the value of X’s [[Get]] attribute.
   b. Set D.[[Set]] to the value of X’s [[Set]] attribute.
7. Set D.[[Enumerable]] to the value of X’s [[Enumerable]] attribute.
8. Set D.[[Configurable]] to the value of X’s [[Configurable]] attribute.
9. Return D.

9.1.6 [[DefineOwnProperty]] (P, Desc)

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

1. Return OrdinaryDefineOwnProperty(O, P, 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 the [[GetOwnProperty]] internal method of O with argument P.
2. Return IfAbrupt(current).
3. Let extensible be the value of the [[Extensible]] internal slot of O.
4. Return ValidateAndApplyPropertyDescriptor(O, P, extensible, Desc, 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 ValidateAndApplyPropertyDescriptor(undefined, undefined, Extensible, Desc, Current).

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

When the abstract operation ValidateAndApplyPropertyDescriptor is called with Object O, property key P, Boolean value extensible, and Property Descriptors Desc, and current the following steps are taken:
This algorithm contains steps that test various fields of the Property Descriptor \( \text{Desc} \) for specific values. The fields that are tested in this manner need not actually exist in \( \text{Desc} \). If a field is absent then its value is considered to be \( \text{false} \).

**NOTE** If \( \text{undefined} \) is passed as the \( O \) argument only validation is performed and no object updates are performed.

1. Assert: If \( O \) is not \( \text{undefined} \) then \( P \) is a valid property key.
2. If \( \text{current} \) is \( \text{undefined} \), then
   a. If \( \text{extensible} \) is \( \text{false} \), then return \( \text{false} \).
   b. Assert: \( \text{extensible} \) is \( \text{true} \).
   c. If IsGenericDescriptor(\( \text{Desc} \)) or IsDataDescriptor(\( \text{Desc} \)) is \( \text{true} \), then
      i. If \( O \) is not \( \text{undefined} \), then create an own data property named \( P \) of object \( O \) whose [[Value]], [[Writable]], [[Enumerable]] and [[Configurable]] attribute values are described by \( \text{Desc} \). If the value of an attribute field of \( \text{Desc} \) is absent, the attribute of the newly created property is set to its default value.
   d. Else \( \text{Desc} \) must be an accessor Property Descriptor,
      i. If \( O \) is not \( \text{undefined} \), then create an own accessor property named \( P \) of object \( O \) whose [[Get]], [[Set]], [[Enumerable]] and [[Configurable]] attribute values are described by \( \text{Desc} \). If the value of an attribute field of \( \text{Desc} \) is absent, the attribute of the newly created property is set to its default value.
   e. Return \( \text{true} \).
3. Return \( \text{true} \), if every field in \( \text{Desc} \) is absent.
4. Return \( \text{true} \), if every field in \( \text{Desc} \) also occurs in \( \text{current} \) and the value of every field in \( \text{Desc} \) is the same value as the corresponding field in \( \text{current} \) when compared using the SameValue algorithm.
5. If the [[Configurable]] field of \( \text{current} \) is \( \text{false} \) then
   a. Return \( \text{false} \), if the [[Configurable]] field of \( \text{Desc} \) is \( \text{true} \).
   b. Return \( \text{false} \), if the [[Enumerable]] field of \( \text{Desc} \) is present and the [[Enumerable]] fields of \( \text{current} \) and \( \text{Desc} \) are the Boolean negation of each other.
6. If IsGenericDescriptor(\( \text{Desc} \)) is \( \text{true} \), then no further validation is required.
7. Else if IsDataDescriptor(\( \text{current} \)) and IsDataDescriptor(\( \text{Desc} \)) have different results, then
   a. Return \( \text{false} \), if the [[Configurable]] field of \( \text{current} \) is \( \text{false} \).
   b. If IsDataDescriptor(\( \text{current} \)) is \( \text{true} \), then
      i. If \( O \) is not \( \text{undefined} \), 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 \( \text{undefined} \), 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(\( \text{current} \)) and IsDataDescriptor(\( \text{Desc} \)) are both \( \text{true} \), then
   a. If the [[Configurable]] field of \( \text{current} \) is \( \text{false} \), then
      i. Return \( \text{false} \), if the [[Writable]] field of \( \text{current} \) is \( \text{false} \) and the [[Writable]] field of \( \text{Desc} \) is \( \text{true} \).
      ii. If the [[Writable]] field of \( \text{current} \) is \( \text{false} \), then
           1. Return \( \text{false} \), if the [[Value]] field of \( \text{Desc} \) is present and SameValue(\( \text{Desc}.[[\text{Value}]]) \) is \( \text{false} \).
   b. Else the [[Configurable]] field of \( \text{current} \) is \( \text{true} \), so any change is acceptable.
9. Else IsAccessorDescriptor(\( \text{current} \)) and IsAccessorDescriptor(\( \text{Desc} \)) are both \( \text{true} \), then
   a. If the [[Configurable]] field of \( \text{current} \) is \( \text{false} \), then
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 not undefined, then return true.
5. Let parent be the result of calling the [[GetPrototypeOf]] internal method of O.
6. ReturnIfAbrupt(parent).
7. If parent is not null, then
   a. Return the result of calling the [[HasProperty]] internal method of parent with argument P.
8. Return false.

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 [[GetProperty]] 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]].
7. If getter is undefined, return undefined.
8. Return the result of calling the [[Call]] internal method of getter with Receiver as the thisArgument and an empty List as argumentsList.

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

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

9.1.10 [[Delete]] (P)

When the [[Delete]] internal method of O is called with property key P 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(delete).
4. If desc is undefined, then return true.
5. If desc.[[Configurable]] is true, then
   a. Remove the own property with name P from O.
   b. Return true.
6. Return false.

9.1.11 [[Enumerate]] ()

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

1. Return an Iterator object (25.1.1.2) whose next method iterates over all the String-valued keys of enumerable properties of O. The mechanics and order of enumerating the properties is not specified but must conform to the rules specified below.
The iterator's `next` method processes object properties to determine whether the property key should be returned as an iterator value. Processed properties do not include properties whose property key is a Symbol. Properties of the object being enumerated may be deleted during enumeration. A property that is deleted before it is processed by the iterator's `next` method is ignored. If new properties are added to the object being enumerated during enumeration, the newly added properties are not guaranteed to be processed in the active enumeration. A property name will be returned by the iterator's `next` method at most once in any enumeration.

Enumerating the properties of an object includes processing properties of its prototype, and the prototype of the prototype, and so on, recursively; but a property of a prototype is not processed if it has the same name as a property that has already been processed by the iterator's `next` method. The values of `[Enumerable]` attributes are not considered when determining if a property of a prototype object has already been processed.

The following is an informative definition of an ECMAScript generator function that conforms to these rules:

```javascript
function* enumerate(obj) {
    if (Object(obj) !== obj) return undefined;
    let visited = new Set;
    while (obj !== null) {
        for (let name of Object.getOwnPropertyNames(obj)) {
            if (!visited.has(name)) {
                let desc = Object.getOwnPropertyDescriptor(obj, name);
                if (desc) {
                    visited.add(name);
                    if (desc.enumerable) yield name;
                }
            }
        }
        obj = Object.getPrototypeOf(obj);
    }
}
```

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` that is an integer index, in ascending numeric index order
   a. Add `P` as the last element of `keys`.
3. For each own property key `P` of `O` that is a String but is not an integer index, in property creation order
   a. Add `P` as the last element of `keys`.
4. For each own property key `P` of `O` that is a Symbol, in property creation order
   a. Add `P` as the last element of `keys`.
5. Return `keys`.

9.1.13 `ObjectCreate(proto, internalSlotsList)` 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 `internalSlotsList` is a List of the names of additional internal slots that must 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. If `internalSlotsList` was not provided, let `internalSlotsList` be an empty List.
2. Let obj be a newly created object with an internal slot for each name in internalSlotsList.
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.

9.1.14 OrdinaryCreateFromConstructor (constructor, intrinsicDefaultProto, internalSlotsList)

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 internalSlotsList is a List of the names of additional internal slots that must 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 GetPrototypeFromConstructor(constructor, intrinsicDefaultProto).
3. ReturnIfAbrupt(proto).
4. Return ObjectCreate(proto, internalSlotsList).

9.2 ECMAScript Function Objects

ECMAScript function objects encapsulate parameterized 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 26.

ECMAScript function objects whose code is not strict mode code (10.2.1) provide an alternative definition for the [[GetOwnProperty]] internal method. This alternative 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 this alternative internal method for ECMAScript function objects. ECMAScript function objects are considered to be ordinary objects even though they may use the alternative definition of [[GetOwnProperty]].
Table 26 — Internal Slots of ECMAScript Function Objects

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Environment]]</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 'normal' or 'generator'.</td>
</tr>
<tr>
<td>[[ECMAScriptCode]]</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 a 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>[[NeedsSuper]]</td>
<td>Boolean</td>
<td>true if this function uses super.</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.</td>
</tr>
<tr>
<td>[[MethodName]]</td>
<td>String or Symbol</td>
<td>If the function uses super, this is the property key that is used for unqualified references to 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  [[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 OrdinaryGetProperty(F, P).
2. If IsDataDescriptor(v) is true, then
   a. If P is "callee" then,
      i. Let callerValue be v.[[Value]].
      ii. If callerValue is an ECMAScript Function object, then
         1. If callerValue’s [[Strict]] internal slot is true, then set v.[[Value]] to null.
   3. Return v.

If an implementation extends function objects non-strict ECMAScript function objects with a built-in callee own property then it must not use this definition of [[GetOwnProperty]]. If an implementation does not provide such an extension, the ordinary object [[GetOwnProperty]] internal method must be used instead.
9.2.2 [[Call]] (thisArgument, argumentsList)

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

1. If \( F \)'s [[ECMAScriptCode]] internal slot has the value `undefined`, then throw a `TypeError` exception.
2. Let \( \text{callerContext} \) be the running execution context.
3. If \( \text{callerContext} \) is not already suspended, then `Suspend callerContext`.
4. Let \( \text{calleeContext} \) be a new ECMAScript Code execution context.
5. Set the Function of \( \text{calleeContext} \) to \( F \).
6. Let \( \text{calleeRealm} \) be the value of \( F \)'s [[Realm]] internal slot.
7. Set the Realm of \( \text{calleeContext} \) to \( \text{calleeRealm} \).
8. Let \( \text{thisMode} \) be the value of \( F \)'s [[ThisMode]] internal slot.
9. Let \( \text{needsThisWrapper} \) be `false`.
10. If \( \text{thisMode} \) is lexical, then
    a. Let \( \text{localEnv} \) be the result of calling `NewDeclarativeEnvironment` passing the value of the [[Environment]] internal slot of \( F \) as the argument.
11. Else, If \( \text{thisMode} \) is strict, then let \( \text{thisValue} \) be \( \text{thisArgument} \).
12. Else, if \( \text{thisArgument} \) is \( \text{null} \) or `undefined`, then
    i. Let \( \text{thisValue} \) be \( \text{calleeRealm}.[[\text{globalThis}]] \).
    ii. Else
        i. If `Type(thisArgument)` is not Object, then let \( \text{needsThisWrapper} \) be `true`.
        ii. Let \( \text{thisValue} \) be \( \text{thisArgument} \).
13. Let \( \text{localEnv} \) be `NewFunctionEnvironment(F, thisValue)`.
14. Let \( \text{functionEnv} \) be \( \text{localEnv} \)'s environment record.
15. If \( \text{needsThisWrapper} \) is `true` then
    a. Let \( \text{wrappedThis} \) be `ToObject(thisArgument)`.
    b. Assert: \( \text{wrappedThis} \) is not an abrupt completion.
    c. NOTE Wrapping deferred until \( \text{calleeContext} \) is running so that ToObject produces objects using \( \text{calleeRealm} \).
    d. Let \( \text{functionEnv} \)'s thisValue to \( \text{wrappedThis} \).
16. Let \( \text{status} \) be the result of performing `FunctionDeclarationInstantiation` using the function \( F \), \( \text{argumentsList} \), and \( \text{localEnv} \) as described in 9.2.13.
17. If \( \text{status} \) is an abrupt completion, then
    a. Remove \( \text{calleeContext} \) from the execution context stack and restore \( \text{callerContext} \) as the running execution context.
    b. Return \( \text{status} \).
18. Let \( \text{result} \) be the result of `EvaluateBody` of the production that is the value of \( F \)'s [[ECMAScriptCode]] internal slot passing \( F \) as the argument.
19. Remove \( \text{calleeContext} \) from the execution context stack and restore \( \text{callerContext} \) as the running execution context.
20. Return \( \text{result} \).
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.

9.2.3 [[Construct]] (argumentsList)
The [[Construct]] internal method for an ECMAScript Function object \( F \) is called with a single parameter \( \text{argumentsList} \) which is a possibly empty List of ECMAScript language values. The following steps are taken:

1. If \( F \)'s [[ECMAScriptCode]] internal slot has the value \text{undefined}, then throw a 
   \text{TypeError} exception.
2. Return Construct\( (F, \text{argumentsList}) \).

9.2.4 FunctionAllocate (functionPrototype, strict) Abstract Operation

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

1. Assert: Type(\( \text{functionPrototype} \)) is Object.
2. Assert: If \( \text{functionKind} \) is present, its value is either \text{"normal"}, \text{"non-constructor"} or \text{"generator"}.
3. If \( \text{functionKind} \) is not present, then let \( \text{functionKind} \) be \text{"normal"}.
4. If \( \text{functionKind} \) is \text{"non-constructor"}, then
   a. Let \( \text{functionKind} \) be \text{"normal"}.
   b. Let needsConstruct be false.
5. Else let needsConstruct be true.
6. Let \( F \) be a newly created ECMAScript function object with the internal slots listed in Table 26. All of those internal slots are initialized to \text{undefined}.
7. Set \( F \)'s essential internal methods except for [[GetOwnProperty]] to the default ordinary object definitions specified in 9.1.
8. If \( \text{strict} \) is true, set \( F \)'s [[GetOwnProperty]] internal method to the default ordinary object definition specified in 9.1.5.
9. Else, set \( F \)'s [[GetOwnProperty]] internal method as specified in 9.2.1.
10. Set \( F \)'s [[Call]] internal method to the definition specified in 9.2.2.
11. If needsConstruct is true, then
    a. Set \( F \)'s [[Construct]] internal method to the definition specified in 9.2.3.
12. Set the [[Strict]] internal slot of \( F \) to \( \text{strict} \).
13. Set the [[FunctionKind]] internal slot of \( F \) to \( \text{functionKind} \).
14. Set the [[Prototype]] internal slot of \( F \) to \( \text{functionPrototype} \).
15. Set the [[Extensible]] internal slot of \( F \) to true.
16. Set the [[Realm]] internal slot of \( F \) to the running execution context’s Realm.
17. Return \( F \).

9.2.5 FunctionInitialize (F, kind, Strict, ParameterList, Body, Scope) Abstract Operation

The abstract operation FunctionInitialize requires the arguments: a function object \( F \), \( \text{kind} \) which is one of (Normal, Method, Arrow), a Boolean \( \text{Strict} \), a parameter list production specified by \( \text{ParameterList} \), a body production specified by \( \text{Body} \), a Lexical Environment specified by \( \text{Scope} \). FunctionInitialize performs the following steps:

1. Let len be the ExpectedArgumentCount of \( \text{ParameterList} \).
2. Let realm be the value of F’s [[Realm]] internal slot.
3. Let status be DefinePropertyOrThrow(F, "length", PropertyDescriptor{[[Value]]: len, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true}).
4. ReturnIfAbrupt(status).
5. Set the [[Strict]] internal slot of F to Strict.
6. Set the [[Environment]] internal slot of F to the value of Scope.
7. Set the [[FormalParameters]] internal slot of F to ParameterList.
8. Set the [[ECMAScriptCode]] internal slot of F to Body.
9. If kind is Arrow, then set the [[ThisMode]] internal slot of F to lexical.
10. Else if Strict is true, then set the [[ThisMode]] internal slot of F to strict.
11. Else set the [[ThisMode]] internal slot of F to global.
12. Return F.

9.2.6 FunctionCreate (kind, ParameterList, Body, Scope, Strict) 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. FunctionCreate performs the following steps:

1. If the functionPrototype argument was not passed, then
   a. Let functionPrototype be the intrinsic object %FunctionPrototype%.
2. If kind is not Normal, then let allocKind be "non-constructor".
3. Else let allocKind be "normal".
4. Let F be FunctionAllocate(functionPrototype, Strict, allocKind).
5. Return FunctionInitialize(F, kind, Strict, ParameterList, Body, Scope).

9.2.7 GeneratorFunctionCreate (kind, ParameterList, Body, Scope, Strict) Abstract Operation

The abstract operation GeneratorFunctionCreate requires the arguments: kind which is one of (Normal, Method), a parameter list production specified by ParameterList, a body production specified by Body, a Lexical Environment specified by Scope, and a Boolean flag strict. GeneratorFunctionCreate performs the following steps:

1. Let functionPrototype be the intrinsic object %Generator%.
2. Let F be FunctionAllocate(functionPrototype, strict, "generator").
3. Return FunctionInitialize(F, kind, strict, ParameterList, Body, Scope).

9.2.8 AddRestrictedFunctionProperties (F, realm) Abstract Operation

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

1. Assert: realm.[[intrinsics]].[[%ThrowTypeError%]] exists and has been initialized.
2. Let thrower be realm.[[intrinsics]].[[%ThrowTypeError%]].
3. Let status be DefinePropertyOrThrow(F, "caller", PropertyDescriptor{[[Get]]: thrower, length: [[Get]]: thrower, [[Enumerable]]: false, [[Configurable]]: true}.
4. Assert: status is not an abrupt completion.
5. Return DefinePropertyOrThrow(F, "arguments", PropertyDescriptor{[[Get]]: thrower, [[Set]]: thrower, [[Enumerable]]: false, [[Configurable]]: true}).
6. Assert: The above returned value is not an abrupt completion.
9.2.8.1 `%ThrowTypeError%()`

The `%ThrowTypeError%` intrinsic is an anonymous built-in function object that is defined once for each Realm. When `%ThrowTypeError%` is called it performs the following steps:

1. Throw a `TypeError` exception.

The value of the `[[Extensible]]` internal slot of a `%ThrowTypeError%` function is `false`.

The `length` property of a `%ThrowTypeError%` function has the attributes `{ `[[Writable]]`: `false`, `[[Enumerable]]`: `false`, `[[Configurable]]`: `false` }`.

9.2.9 `MakeConstructor` (F, writablePrototype, prototype) 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. Assert: `F` has a `[[Constructor]]` internal method.
3. Let `installNeeded` be `false`.
4. If the `prototype` argument was not provided, then
   a. Let `installNeeded` be `true`.
   b. Let `prototype` be `ObjectCreate(%ObjectPrototype%)`.
5. If the `writablePrototype` argument was not provided, then
   a. Let `writablePrototype` be `true`.
6. If `installNeeded`, then
   a. Let `status` be `DefinePropertyOrThrow(prototype, "constructor", PropertyDescriptor{[[Value]]: F, [[Writable]]: writablePrototype, [[Enumerable]]: false, [[Configurable]]: writablePrototype })`.
   b. ReturnIfAbrupt(`status`).
7. Let `status` be `DefinePropertyOrThrow(F, "prototype", PropertyDescriptor{[[Value]]: prototype, [[Writable]]: writablePrototype, [[Enumerable]]: false, [[Configurable]]: false})`.
8. ReturnIfAbrupt(`status`).
9. Return NormalCompletion(`undefined`).

9.2.10 `MakeMethod` (F, methodName, homeObject) Abstract Operation

The abstract operation `MakeMethod` with arguments `F`, `methodName` and `homeObject` configures `F` as a method by performing the following steps:

1. Assert: `F` is an ECMAScript function object.
2. Assert: `methodName` is either `undefined` or a property key.
3. Assert: Type(`homeObject`) is either Undefined or Object.
4. Set the `[[NeedsSuper]]` internal slot of `F` to `true`.
5. Set the `[[HomeObject]]` internal slot of `F` to `homeObject`.
6. Set the `[[MethodName]]` internal slot of `F` to `methodName`.
7. Return NormalCompletion(`undefined`).

Commented [AWB1915]: Same as ES5
9.2.11 SetFunctionName (F, name, prefix) 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 object that does not have a name own property.
2. Assert: Type(name) is either Symbol or String.
3. If prefix was passed, Type(prefix) is String.
4. If Type(name) is Symbol, then
   a. Let description be name’s [[Description]] value.
   b. If description is undefined, then let name be the empty String.
   c. Else, let name be the concatenation of " [", description, and "] ".
5. If prefix was passed, then
   a. Let name be the concatenation of prefix, Unicode code point U+0020 (Space), and name.
6. Return DefinePropertyOrThrow(F, "name", PropertyDescriptor{[[Value]]: name, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true}).

9.2.12 CloneMethod (function, newHome, newName) Abstract Operation

The abstract operation CloneMethod is called with a function object function, an object newHome, and a property key newName as its argument. It performs the following steps:

1. Assert: function is an ECMAScript function object or an exotic Built-in function object.
2. Assert: Type(newHome) is Object.
3. Assert: Type(newName) is one of Undefined, String, or Symbol.
4. If function is an ECMAScript function, then
   a. Let new be a new ECMAScript function object that has all of the same internal methods and internal slots as function.
5. Else
   a. Assert: function is an exotic Built-in function object.
   b. Let new be a new exotic Built-in function object that has all of the same internal methods and internal slots as function.
6. Set the value of each of new’s internal slots, except for [[Extensible]], [[HomeObject]] and [[MethodName]] to the value of function’s corresponding internal slot.
7. Set new’s [[Extensible]] internal slot to true.
8. If the value of function’s [[NeedsSuper]] internal slot is true, then
   a. Set the value of new’s [[HomeObject]] internal slot to newHome.
   b. If newName is not undefined, then
      i. Set the value of new’s [[MethodName]] internal slot to newName.
   c. Else, i. Set the value of new’s [[MethodName]] internal slot to the value of function’s [[MethodName]] internal slot.
9. 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. However, properties of the function object, except for the restricted function properties, are not created or copied.

9.2.13 FunctionDeclarationInstantiation (func, argumentsList, env) Abstract Operation

NOTE When an execution context is established for evaluating an ECMAScript function a new Declarative Environment Record is created and bindings for each formal parameter are instantiated in that environment record.
Each declaration in the function body is also instantiated. If the function’s formal parameters do not include any default value initializers then the body declarations are instantiated in the same environment record as the parameters. If default value parameter initializers exist, a second environment record is created for the body declarations. Formal parameters and functions are initialized as part of FunctionDeclarationInstantiation. All other bindings are initialized during evaluation of the function body.

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

1. Let `code` be the value of the `[[ECMAScriptCode]]` 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. If `parameterNames` has any duplicate entries, let `hasDuplicates` be `true`. Otherwise, let `hasDuplicates` be `false`.
7. Let `hasParameterExpressions` be `ContainsExpression of formals`.
8. Let `varNames` be the VarDeclaredNames of `code`.
9. Let `varDeclarations` be the VarScopedDeclarations of `code`.
10. Let `lexicalNames` be the LexicallyDeclaredNames of `code`.
11. Let `functionNames` be an empty List.
12. Let `functionsToInitialize` be an empty List.
13. For each `d` in `varDeclarations`, in reverse list order do
   a. If `d` is neither a VariableDeclaration or a ForBinding, then
      i. Assert: `d` is either a FunctionDeclaration or a GeneratorDeclaration.
      ii. Let `fn` be the sole element of the BoundNames of `d`.
      iii. If `fn` is not an element of `functionNames`, then
          1. Insert `fn` as the first element of `functionNames`.
          2. NOTE If there are multiple FunctionDeclarations or GeneratorDeclarations for the same name, the last declaration is used.
          3. Insert `d` as the first element of `functionsToInitialize`.
   14. Let `argumentsObjectNeeded` be `true`.
15. If the value of the `[[ThisMode]]` internal slot of `func` is lexical, then
   a. NOTE Arrow functions never have an arguments objects.
   b. Let `argumentsObjectNeeded` be `false`.
16. Else if `arguments` is an element of `parameterNames`, then
   a. Let `argumentsObjectNeeded` be `false`.
17. Else if `hasParameterExpressions` is `false`, then
   a. If `arguments` is an element of `functionNames` or if `arguments` is an element of `lexicalNames`, then
      i. Let `argumentsObjectNeeded` be `false`.
18. For each `paramName` in `parameterNames`, do
   a. Let `alreadyDeclared` be the result of calling `env`’s `HasBinding` concrete method passing `paramName` as the argument.
   b. NOTE Early errors ensure that duplicate parameter names can only occur in non-strict functions that do not have parameter default values or rest parameters.
   c. If `alreadyDeclared` is `false`, then
      i. Let `status` be the result of calling `env`’s `CreateMutableBinding` concrete method passing `paramName` as the argument.
      ii. If `hasDuplicates` is `true`, then
          1. Let `status` be the result of calling `env`’s `InitializeBinding` concrete method passing `paramName` and `undefined` as the argument.
iii. Assert: status is never an abrupt completion for either of the above operations.

19. If argumentsObjectNeeded is true, then
   a. If strict is true or if simpleParameterList is false, then
      i. Let ao be CreateUnmappedArgumentsObject(argumentsList).
   b. Else,
      i. NOTE mapped argument object is only provided for non-strict functions that don’t have a
         rest parameter, any parameter default value initializers, or any destructured parameters.
      ii. Let ao be CreateMappedArgumentsObject(func, formals, argumentsList, env).
   c. ReturnIfAbrupt(ao).
   d. If strict is true, then
      i. Let status be the result of calling env’s CreateImmutableBinding concrete method passing
         “arguments” as the argument.
      e. Else,
         i. Let status be the result of calling env’s CreateMutableBinding concrete method passing
            “arguments” as the argument.
      f. Assert: status is never an abrupt completion.
   e. Call env’s InitializeBinding concrete method passing “arguments” and ao as arguments.
   f. Append ”arguments” to parameterNames.

20. If hasDuplicates is true, then
   a. Let formalStatus be the result of performing IteratorBindingInitialization for formals with
      CreateListIterator(argumentsList) and undefined as arguments.
   21. Else,
      a. Let formalStatus be the result of performing IteratorBindingInitialization for formals with
         CreateListIterator(argumentsList) and env as arguments.
   b. ReturnIfAbrupt(formalStatus).
   22. If hasParameterExpressions is false, then
      a. NOTE Only a single environment record is needed.
      b. Let bodyEnv be env.
      c. Let instantiatedVarNames be a copy of the List parameterNames.
      d. For each n in varNames, do
         i. If n is not an element of instantiatedVarNames, then
            1. Append n to instantiatedVarNames.
            2. Let status be the result of calling bodyEnv’s CreateMutableBinding concrete method
               passing n as the argument.
            3. Assert: status is never an abrupt completion.
            4. Call bodyEnv’s InitializeBinding concrete method passing n and undefined as arguments.
      23. Else,
         a. NOTE A separate environment record is needed to ensure that closures created by expressions
            in the formal parameter list do not have visibility of declarations in the function body.
         b. Let bodyEnv be NewDeclarativeEnvironment(env).
         c. Let calleeContext be the running execution context.
         d. Set the LexicalEnvironment of calleeContext to bodyEnv.
         e. Set the VariableEnvironment of calleeContext to bodyEnv.
         f. Let instantiatedVarNames be a new emptyList.
         g. For each n in varNames, do
            i. If n is not an element of instantiatedVarNames, then
               1. Append n to instantiatedVarNames.
               2. Let status be the result of calling bodyEnv’s CreateMutableBinding concrete method
                  passing n as the argument.
               3. Assert: status is never an abrupt completion.
               4. If n is not an element of parameterNames or if n is an element of functionNames, then
                  let initialValue be undefined.
5. else,
   a. Let initialValue be the result of calling env’s GetBindingValue concrete method passing n and false as the arguments.
   b. ReturnIfAbrupt(initialValue).
6. Call bodyEnv’s InitializeBinding concrete method passing n and initialValue as arguments.

7. NOTE vars whose names are the same as a formal parameter, initially have the same
   value as the corresponding initialized parameter.

25. Let lexDeclarations be the LexicallyScopedDeclarations of code.
26. For each element d in lexDeclarations do
   a. NOTE A lexically declared name cannot be the same as a function/generator declaration, formal
      parameter, or a var name. Lexically declared names are only instantiated here but not initialized.
   b. For each element dn of the BoundNames of d do
      i. If IsConstantDeclaration of d is true, then
         1. Let status be the result of calling bodyEnv’s CreateImmutableBinding concrete method
            passing dn as the argument.
      ii. Else,
         1. Let status be the result of calling bodyEnv’s CreateMutableBinding concrete method
            passing dn and false as the arguments.
   c. Assert: status is never an abrupt completion.
27. For each production 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 bodyEnv.
   c. Let status be the result of calling bodyEnv’s SetMutableBinding concrete method passing fn, fo
      and false as the arguments.
   d. Assert: status is never an abrupt completion.
28. Return NormalCompletion(empty).

NOTE B.3.2 provides an extension to the above algorithm that is necessary for backwards compatibility
with web browser implementations of ECAMScript that predate the sixth edition of ECMA-262.

9.3 Built-in Function Objects

The built-in function objects defined in this specification may be implemented as either ECAMScript
function objects (9.2) whose behaviour is provided using ECAMScript 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. An implementation may also provide additional
built-in function objects that are not defined in this specification.

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 9.2.1. All such exotic function
objects also have [[Prototype]] and [[Extensible]] internal slots.

Unless otherwise specified every built-in function object initially has the %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 ECAMScript function object then this specified behaviour must be implemented by the
ECAMScript code that is the body of the function. Built-in functions that are ECAMScript 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. It must also have 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 \( \text{thisArgument} \) and \( \text{argumentsList} \), a List of ECMAScript language values. The following steps are taken:

1. Let \( \text{calleeContext} \) be the running execution context.
2. If \( \text{calleeContext} \) is not already suspended, then Suspend \( \text{calleeContext} \).
3. Let \( \text{calleeContext} \) be a new execution context.
4. Set the Function of \( \text{calleeContext} \) to \( F \).
5. Set the Realm of \( \text{calleeContext} \) to \( \text{calleeRealm} \).
6. Perform any necessary implementation defined initialization of \( \text{calleeContext} \).
7. Push \( \text{calleeContext} \) onto the execution context stack; \( \text{calleeContext} \) is now the running execution context.
8. Let \( \text{result} \) 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 \( \text{calleeContext} \) from the execution context stack and restore \( \text{calleeContext} \) as the running execution context.
10. Return \( \text{result} \).

NOTE 1 When \( \text{calleeContext} \) 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(realm, steps, prototype, internalSlotsList) Abstract Operation

The abstract operation CreateBuiltInFunction takes arguments \( \text{realm} \), \( \text{prototype} \), and \( \text{steps} \). The optional argument \( \text{internalSlotsList} \) is a List of the names of additional internal slot that must be defined as part of the object. If the list is not provided, an empty List is used. CreateBuiltInFunction returns a built-in function object created by the following steps:

1. Assert: \( \text{realm} \) is a Realm Record.
2. Assert: \( \text{steps} \) is either a set of algorithm steps or other definition of a functions behaviour provided in this specification.
3. Let \( \text{func} \) be a new built-in function object that when called performs the action described by \( \text{steps} \). The new function object has internal slots whose names are the elements of \( \text{internalSlotsList} \). The initial value of each of those internal slots is undefined.
4. Set the [[Realm]] internal slot of \( \text{func} \) to \( \text{realm} \).
5. Call the [[SetPrototypeOf]] internal method of \( \text{func} \) with argument \( \text{prototype} \).
6. 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 [[Call]] internal method and may have a [[Construct]] internal method). Calling a bound function generally results in a call of its wrapped function.

Bound function objects do not have the internal slots of ECMAScript function objects defined in Table 26. Instead they have the internal slots defined in Table 27.

Table 27 — 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>[[BoundTargetFunction]]</td>
<td>Callable Object</td>
<td>The wrapped function object.</td>
</tr>
<tr>
<td>[[BoundThis]]</td>
<td>Any</td>
<td>The value that is always passed as the this value when calling the wrapped function.</td>
</tr>
<tr>
<td>[[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 wrapped function.</td>
</tr>
</tbody>
</table>

Unlike ECMAScript function objects, bound function objects do not use alternative definitions of the [[Get]] and [[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 [[Call]]

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

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

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 (targetFunction, boundThis, boundArgs) 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.

### 9.4.1.4 BoundFunctionClone (function) Abstract Operation

The abstract operation BoundFunctionClone is called with argument function it performs the following steps:

1. Assert: function is a Bound Function exotic object.
2. Let new be a new Bound Function exotic object that has all of the same internal methods and internal slots as function.
3. Set the value of each of new’s internal slots, except for [[Extensible]] to the value of function’s corresponding internal slot.
4. Set new’s [[Extensible]] internal slot to true.
5. Return new.

### 9.4.2 Array Exotic Objects

An Array object is an exotic object that gives special treatment to array index property keys (see 6.1.7). 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.

**NOTE** A String property name $P$ is an array index if and only if `ToString(ToUint32(P))` is equal to $P$ and $\text{ToUint32}(P)$ is not equal to $2^{31} - 1$.

Exotic Array objects have the same internal slots as ordinary objects. They also have an [[ArrayInitializationState]] 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 ArraySetLength(A, Desc).
3. Else if P is an array index, then
   a. Let oldLenDesc be OrdinaryGetOwnProperty(A, "length").
   b. Assert: oldLenDesc 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 OrdinaryDefineOwnProperty(A, "length", oldLenDesc).
      iii. ReturnIfAbrupt(succeeded).
   k. Return true.

9.4.2.2 ArrayCreate(length, proto) 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:

1. Assert: length is either undefined or a integer Number ≥ 0.
2. If length is −0, then let length be +0.
3. If the proto argument was not passed, then let proto be the intrinsic object %ArrayPrototype%.
4. Let A be a newly created Array exotic object.
5. Set A’s essential internal methods except for [[DefineOwnProperty]] to the default ordinary object definitions specified in 9.1.
6. Set the [[DefineOwnProperty]] internal method of A as specified in 9.4.2.1.
7. Set the [[Prototype]] internal slot of A to proto.
8. Set the [[Extensible]] internal slot of A to true.
9. If length is not undefined, then
   a. Set the [[ArrayInitializationState]] internal slot of A to true.
10. Else
    a. Set the [[ArrayInitializationState]] internal slot of A to false.
    b. Let length be +0.
11. If length > 2^{32}−1, then throw a RangeError exception.
12. Call OrdinaryDefineOwnProperty with arguments A, **"length"** and PropertyDescriptor{[[Value]]: length, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false}.

13. 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:

1. If the [[Value]] field of Desc is absent, then
   a. Return OrdinaryDefineOwnProperty(A, **"length"**, Desc).
2. Let newLenDesc be a copy of Desc.
3. Let newLen be ToUint32(Desc.[[Value]]).
4. If newLen is not equal to ToNumber(Desc.[[Value]]), throw a **RangeError** exception.
5. Set newLenDesc.[[Value]] to newLen.
6. Let oldLenDesc be the result of calling the [[GetOwnProperty]] internal method of A passing **"length"** as the argument.
7. ReturnIfAbrupt(oldLenDesc).
8. Assert: oldLenDesc 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. Let oldLen be oldLenDesc.[[Value]].
10. If newLen ≥ oldLen, then
    a. Return OrdinaryDefineOwnProperty(A, **"length"**, newLenDesc).
11. If oldLenDesc.[[Writable]] is false, then return false.
12. If newLenDesc.[[Writable]] is absent or has the value true, let newWritable be true.
13. Else,
   a. Need to defer setting the [[Writable]] attribute to false in case any elements cannot be deleted.
   b. Let newWritabl be false.
   c. Set newLenDesc.[[Writable]] to true.
14. Let succeeded be OrdinaryDefineOwnProperty(A, **"length"**, newLenDesc).
15. ReturnIfAbrupt(succeeded).
16. If succeeded is false, return false.
17. 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 OrdinaryDefineOwnProperty(A, **"length"**, newLenDesc).
      iv. ReturnIfAbrupt(succeeded).
      v. Return false.
18. If newWritable is false, then
   a. Call OrdinaryDefineOwnProperty passing A, **"length"**, and PropertyDescriptor{[[Writable]]: false} as arguments. This call will always return true.
19. Return true.

**NOTE**: In steps 3 and 4, if Desc.[[Value]] is an object then its **valueOf** method is called twice. This is legacy behaviour that was specified with this effect starting with the 2nd Edition of this specification.
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 OrdinaryGetOwnProperty(S, P).
3. If desc is not undefined return desc.
4. If Type(P) is not String, then return undefined.
5. Let index be CanonicalNumericIndexString(P).
6. Assert: index is not an abrupt completion.
7. If index is undefined, then return undefined.
8. If IsInteger(index) is false, then return undefined.
9. If index = ~0, then 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 index < 0 or 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 [[Enumerate]] ()

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

1. Let indexKeys be a new empty List.
2. Let str be the String value of the [[StringData]] internal slot of O, if the value of [[StringData]] is undefined the empty string is used as its value.
3. Let len be the number of elements in str.
4. For each integer i starting with 0 such that i < len, in ascending order,
   a. Add ToString(i) as the last element of indexKeys
5. Let ordinary be the result of calling the default ordinary object [[Enumerate]] internal method (9.1.11) on O.
6. ReturnIfAbrupt(ordinary).
7. Return `CreateCompoundIterator(CreateListIterator(indexKeys), ordinary).

### 9.4.3.3 `[[OwnPropertyKeys]]` ()

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

1. Let `keys` be a new empty List.
2. Let `str` be the String value of the `[[StringData]]` internal slot of `O`, if the value of `[[StringData]]` is `undefined` the empty string is used as its value.
3. Let `len` be the number of elements in `str`.
4. For each integer `i` starting with 0 such that `i < len`, in ascending order,
   a. Add `ToString(i)` as the last element of `keys`.
5. For each own property key `P` of `O` such that `P` is an integer index and `ToInteger(P) ≥ len`, in ascending numeric index order,
   a. Add `P` as the last element of `keys`.
6. For each own property key `P` of `O` such that `Type(P)` is `String` and `P` is not an integer index, in property creation order,
   a. Add `P` as the last element of `keys`.
7. For each own property key `P` of `O` such that `Type(P)` is `Symbol`, in property creation order,
   a. Add `P` as the last element of `keys`.
8. Return `keys`.

### 9.4.3.4 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 `[[Enumerate]]` internal method of `A` as specified in 9.4.3.2.
4. Set the `[[OwnPropertyKeys]]` internal method of `A` as specified in 9.4.3.3.
5. Set the `[[Prototype]]` internal slot of `A` to `prototype`.
6. Set the `[[Extensible]]` internal slot of `A` to `true`.
7. Return `A`.

### 9.4.4 Arguments Exotic Objects

Most ECMAScript functions make an arguments objects available to their code. Depending upon the characteristics of the function definition, its argument object is either an ordinary object or an arguments exotic object. An arguments exotic object is an exotic object whose array index properties map to the formal parameters bindings of an invocation of its associated ECMAScript function.

Arguments exotic objects have the same internal slots as ordinary objects. They also have a `[[ParameterMap]]` internal slot. Ordinary arguments objects also have a `[[ParameterMap]]` internal slot whose value is always `undefined`. For ordinary argument objects the `[[ParameterMap]]` internal slot is only used by `Object.prototype.toString(19.1.3.6)` to identify them as such.

Arguments exotic 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.
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 observable from ECMAScript code. An ECMAScript implementation does not need to actually create or use such objects to implement the specified semantics.

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

9.4.4.1  [[GetOwnProperty]] (P)

The [[GetOwnProperty]] internal method of an arguments exotic object when called with a property name P performs the following steps:

1. Let args be the arguments object.
2. Let desc be OrdinaryGetOwnProperty(args, P).
3. If desc is undefined then return desc.
4. Let map be the value of the [[ParameterMap]] internal slot of the arguments object.
5. Let isMapped be HasOwnProperty(map, P).
6. Assert: isMapped is never an abrupt completion.
7. If the value of isMapped is true, then
   a. Set desc.[[Value]] to Get(map, P).
8. If IsDataDescriptor(desc) is true and P is "caller" and desc.[[Value]] is a strict mode Function object, throw a TypeError exception.
9. Return desc.

If an implementation does not provide a built-in caller property for argument exotic objects then step 8 of this algorithm is must be skipped.

9.4.4.2  [[DefineOwnProperty]] (P, Desc)

The [[DefineOwnProperty]] internal method of an arguments exotic object when called with a property name P and Property Descriptor Desc 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 HasOwnProperty(map, P).
4. Let allowed be OrdinaryDefineOwnProperty(args, P, Desc).
5. ReturnIfAbrupt(allowed).
6. If allowed is false, then return false.
7. If the value of isMapped is true, then
   a. If IsAccessorDescriptor(Desc) is true, then
      i. Call the [[Delete]] internal method of map passing P as the argument.
b. Else
   i. If Desc.[Value] is present, then
      1. Let putStatus be Put(map, P, Desc.[Value], false).
      2. Assert: putStatus is true because formal parameters mapped by argument objects are always writable.
   ii. If Desc.[Writable] is present and its value is false, then
      1. Call the [[Delete]] internal method of map passing P as the argument.

8. Return true.

9.4.4.3  [[Get]] (P, Receiver)

The [[Get]] internal method of an arguments exotic object when called with a property name P and ECMA Script language value Receiver 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 HasOwnProperty(map, P).
4. Assert: isMapped is not an abrupt completion.
5. If the value of isMapped is false, then
   a. Let v be the result of calling the default ordinary object [[Get]] internal method (9.1.8) on args passing P and Receiver as the arguments.
6. Else map contains a formal parameter mapping for P,
   a. Let v be Get(map, P).
7. ReturnIfAbrupt(v).
8. If P is "caller" and v is a strict mode Function object, throw a TypeError exception.

If an implementation does not provide a built-in caller property for argument exotic objects then step 8 of this algorithm must be skipped.

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

The [[Set]] internal method of an arguments exotic object when called with property key P, value V, and ECMA Script language value Receiver performs the following steps:

1. Let args be the arguments object.
2. If SameValue(args, Receiver) is false, then
   a. Let isMapped be undefined.
3. Else,
   a. Let map be the value of the [[ParameterMap]] internal slot of the arguments object.
   b. Let isMapped be HasOwnProperty(map, P).
   c. Assert: isMapped is not an abrupt completion.
4. If the value of isMapped is false, then
   a. Return the result of calling the default ordinary object [[Set]] internal method (9.1.8) on args passing P, V and Receiver as the arguments.
5. Else map contains a formal parameter mapping for P,

9.4.4.5  [[Delete]] (P)

The [[Delete]] internal method of an arguments exotic object when called with a property key P performs the following steps:
1. Let map be the value of the [[ParameterMap]] internal slot of the arguments object.
2. Let isMapped be HasOwnProperty(map, P).
3. Assert: isMapped is not an abrupt completion.
4. Let 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.
5. ReturnIfAbrupt(result).
6. If result is true and the value of isMapped is true, then
   a. Call the [[Delete]] internal method of map passing P as the argument.
7. Return result.

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

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

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

9.4.4.6 CreateUnmappedArgumentsObject(argumentsList) Abstract Operation

The abstract operation CreateUnmappedArgumentsObject called with an argument argumentsList performs the following steps:

1. Let len be the number of elements in argumentsList.
2. Let obj be ObjectCreate(ObjectPrototype%, {}[[ParameterMap]]).
3. Set obj's [[ParameterMap]] internal slot to undefined.
4. Perform DefinePropertyOrThrow(obj, "length", PropertyDescriptor{[[Value]]: len, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true}).
5. Let index be 0.
6. Repeat while index < len.
   a. Let val be the element of argumentsList at 0-originated list position index.
   b. Perform CreateDataProperty(obj, ToString(index), val).
   c. Let index be index + 1
7. Perform DefinePropertyOrThrow(obj, @@iterator, PropertyDescriptor
      {[[Value]]: %ArrayPrototype_values%, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true}).
8. Perform DefinePropertyOrThrow(obj, "caller", PropertyDescriptor [[[Get]]:
      %ThrowTypeError%, [[Set]]: %ThrowTypeError%, [[Enumerable]]: false, [[Configurable]]: false}).
9. Perform DefinePropertyOrThrow(obj, "callee", PropertyDescriptor [[[Get]]:
      %ThrowTypeError%, [[Set]]: %ThrowTypeError%, [[Enumerable]]: false, [[Configurable]]: false}).
9.4.4.7 CreateMappedArgumentsObject (func, formals, argumentsList, env) Abstract Operation

The abstract operation CreateMappedArgumentsObject is called with object func, grammar production formals, List argumentsList, and environment record env. The following steps are performed:

1. Assert: formals does not contain a rest parameter, any binding patterns, or any initializers. It may contain duplicate identifiers.
2. Let len be the number of elements in argumentsList.
3. Let obj be a newly created arguments exotic object with a [[ParameterMap]] internal slot.
4. Set the [[GetOwnProperty]] internal method of obj as specified in 9.4.4.1.
5. Set the [[DefineOwnProperty]] internal method of obj as specified in 9.4.4.2.
6. Set the [[Get]] internal method of obj as specified in 9.4.4.3.
7. Set the [[Set]] internal method of obj as specified in 9.4.4.4.
8. Set the [[Delete]] internal method of obj as specified in 9.4.4.5.
9. Set the remainder of obj’s essential internal methods to the default ordinary object definitions specified in 9.1.
10. Set the [[Prototype]] internal slot of obj to %ObjectPrototype%.
11. Set the [[Extensible]] internal slot of obj to true.
12. Let parameterNames be the BoundNames of formals.
13. Let numberOfParameters be the number of elements in parameterNames
14. Let index be 0.
15. Repeat while index < len,
   a. Let val be the element of argumentsList at 0-originated list position index.
   b. Perform CreateDataProperty(obj, ToString(index), val).
   c. Let index be index + 1
16. Perform DefinePropertyOrThrow(obj, "length", PropertyDescriptor{[[Value]]: len, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true}).
17. Let map be ObjectCreate(null).
18. Let mappedNames be an empty List.
19. Let index be numberOfParameters - 1.
20. Repeat while index ≥ 0,
   a. Let name be the element of parameterNames at 0-originated list position index.
   b. If name is not an element of mappedNames, then
      i. Add name as an element of the list mappedNames.
   c. Let index be index - 1
21. Set the [[ParameterMap]] internal slot of obj to map.
22. Perform DefinePropertyOrThrow(obj, @iterator, PropertyDescriptor
   {[[Value]]:%ArrayProto_values%, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true}).
23. Perform DefinePropertyOrThrow(obj, "callee", PropertyDescriptor
   {[[Value]]: func, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true}).
24. Assert: the above property definitions will not produce an abrupt completion.
25. Return obj
9.4.4.7.1 MakeArgGetter (name, env) Abstract Operation

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

1. Let realm be the current Realm.
2. Let steps be the steps of a ArgGetter function as specified below.
3. Let getter be CreateBuiltInFunction(realm, steps, %FunctionPrototype%, , [[name]], [[env]]) .
4. Set getter’s [[name]] internal slot to name.
5. Set getter’s [[env]] internal slot to env.
6. Return getter.

An ArgGetter function is an anonymous built-in function with [[name]] and [[env]] internal slots. When an ArgGetter function f that expects no arguments is called it performs the following steps:

1. Let name be the value of f’s [[name]] internal slot.
2. Let env be the value of f’s [[env]] internal slot
3. Return the result of calling the GetBindingValue concrete method of env with arguments name and false.

NOTE ArgGetter functions are never directly accessible to ECMAScript code.

9.4.4.7.2 MakeArgSetter (name, env) Abstract Operation

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

1. Let realm be the current Realm.
2. Let steps be the steps of a ArgSetter function as specified below.
3. Let setter be CreateBuiltInFunction(realm, steps, %FunctionPrototype%, , [[name]], [[env]]) .
4. Set setter’s [[name]] internal slot to name.
5. Set setter’s [[env]] internal slot to env.
6. Return setter.

An ArgSetter function is an anonymous built-in function with [[name]] and [[env]] internal slots. When an ArgSetter function f is called with argument value it performs the following steps:

1. Let name be the value of f’s [[name]] internal slot.
2. Let env be the value of f’s [[env]] internal slot
3. Return the result of calling the SetMutableBinding concrete method of env with arguments name, value, and false.

NOTE ArgSetter functions are never directly accessible to ECMAScript code.

9.4.5 Integer Indexed Exotic Objects

An Integer Indexed object is an exotic object that performs special handling of integer index 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 `numericIndex` be CanonicalNumericIndexString(P).
   b. Assert: `numericIndex` is not an abrupt completion.
   c. If `numericIndex` is not undefined, then
      i. Let value be IntegerIndexedElementGet(O, `numericIndex`).
      ii. ReturnIfAbrupt(value).
   d. If value is undefined, then return undefined.
   e. Return a PropertyDescriptor( [[Value]]: value, [[Enumerable]]: true, [[Writable]]: true, [[Configurable]]: false ).
4. Return 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 `numericIndex` be CanonicalNumericIndexString(P).
   b. Assert: `numericIndex` is not an abrupt completion.
   c. If `numericIndex` is not undefined, then
      i. If the value of `O`'s [[ViewedArrayBuffer]] is undefined, then throw a TypeError exception.
   d. If IsInteger(`numericIndex`) is false then return false.
   e. Let `intIndex` be numericIndex.
   f. If `intIndex` is < 0, then return false.
   g. If `intIndex` is ≥ length, then return false.
   h. Let `length` be the value of `O`'s [[ArrayLength]] internal slot.
   i. If `intIndex` is ≥ `length`, then return false.
   j. If `Desc` has a [[Configurable]] field and if `Desc`.[[Configurable]] is true, then return false.
   k. If `Desc` has an [[Enumerable]] field and if `Desc`.[[Enumerable]] is false, then return false.
   l. If `Desc` has a [[Writable]] field and if `Desc`.[[Writable]] is false, then return false.
   m. If `Desc` has a [[Value]] field, then
      1. Let `value` be `Desc`.[[Value]].
      2. Let status be IntegerIndexedElementSet(O, `intIndex`, value).
      3. ReturnIfAbrupt(status).
   n. Return `true`.
5. Return OrdinaryDefineOwnProperty(O, P, Desc).
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 numericIndex be CanonicalNumericIndexString (P).
   b. Assert: numericIndex is not an abrupt completion.
   c. If numericIndex is not undefined, then
      i. Return IntegerIndexedElementGet (O, numericIndex).
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
   a. Let numericIndex be CanonicalNumericIndexString (P).
   b. Assert: numericIndex is not an abrupt completion.
   c. If numericIndex is not undefined, then
      i. Return ToBoolean(IntegerIndexedElementSet (O, numericIndex, 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]] ()

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

1. Let indexKeys be a new empty List.
2. Assert: O is an Object that has [[ViewedArrayBuffer]], [[ArrayLength]], [[ByteOffset]], and [[TypedArrayName]] internal slots.
3. If the value of O’s [[ViewedArrayBuffer]] is undefined, then throw a TypeError exception.
4. Let len be the value of O’s [[ArrayLength]] internal slot.
5. For each integer i starting with 0 such that i < len, in ascending order, a. Add ToString(i) as the last element of indexKeys.
6. Let ordinary be the result of calling the default ordinary object [[Enumerate]] internal method (9.1.11) on O.
7. ReturnIfAbrupt(ordinary).
8. Return CreateCompoundIterator(CreateListIterator(indexKeys), ordinary).

9.4.5.6 [[OwnPropertyKeys]] ()

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

1. Let keys be a new empty List.
2. Assert: O is an Object that has [[ViewedArrayBuffer]], [[ArrayLength]], [[ByteOffset]], and [[TypedArrayName]] internal slots.
3. If the value of \( O \)'s [[ViewedArrayBuffer]] is `undefined`, then throw a `TypeError` exception.
4. Let `len` be the value of \( O \)'s [[ArrayLength]] internal slot.
5. For each integer \( i \) starting with \( 0 \) such that \( i < len \), in ascending order,
   a. Add `ToString\((i)\)` as the last element of `keys`.
6. For each own property key \( P \) of \( O \) such that \( P \) is an integer index and `ToInteger\((P)\) ≥ len`, in ascending numeric index order
   a. Add `P` as the last element of `keys`.
7. For each own property key \( P \) of \( O \) such that `Type\((P)\)` is `String` and \( P \) is not an integer index, in property creation order
   a. Add `P` as the last element of `keys`.
8. For each own property key \( P \) of \( O \) such that `Type\((P)\)` is `Symbol`, in property creation order
   a. Add `P` as the last element of `keys`.
9. Return `keys`.

9.4.5.7 `IntegerIndexedObjectCreate` (prototype, internalSlotsList) Abstract Operation

The abstract operation `IntegerIndexedObjectCreate` with arguments `prototype` and `internalSlotsList` is used to specify the creation of new Integer Indexed exotic objects. The argument `internalSlotsList` is a List of the names of additional internal slots that must be defined as part of the object. `IntegerIndexedObjectCreate` performs the following steps:

1. Let \( A \) be a newly created object with an internal slot for each name in `internalSlotsList`.
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

The abstract operation `IntegerIndexedElementGet` with arguments \( O \) and `index` performs the following steps:

1. Assert: `Type\((index)\)` is `Number`.
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. If `IsDetachedBuffer\((buffer)\)` is `true`, then throw a `TypeError` exception.
6. If `IsInteger\((index)\)` is `false` then return `undefined`.
7. If `index = -0`, then return `undefined`.
8. Let `length` be the value of \( O \)'s `[[ArrayLength]]` internal slot.
9. If `index < 0` or `index ≥ length`, then return `undefined`.
10. Let `offset` be the value of \( O \)'s `[[ByteOffset]]` internal slot.
11. Let `arrayTypeName` be the string value of \( O \)'s `[[TypedArrayName]]` internal slot.
12. Let `elementSize` be the Number value of the Element Size value specified in Table 45 for `arrayTypeName`.
13. Let `indexedPosition = (index × elementSize) + offset`.
14. Let `elementType` be the string value of the Element Type value in Table 45 for `arrayTypeName`.
15. Return `GetValueFromBuffer(buffer, indexedPosition, elementType)`.

9.4.5.9 ArgumentIndexedElementSet( O, index, value ) Abstract Operation

The abstract operation ArgumentIndexedElementSet with arguments `O`, `index`, and `value` performs the following steps:

1. Assert: `Type(index)` is Number.
2. Assert: `O` is an Object that has `[[ViewedArrayBuffer]]`, `[[ArrayLength]]`, `[[ByteOffset]]`, and `[[TypedArrayName]]` internal slots.
3. Let `numValue` be `ToNumber(value)`.
4. ReturnIfAbrupt(`numValue`).
5. Let `buffer` be the value of `O`’s `[[ViewedArrayBuffer]]` internal slot.
6. If `buffer` is `undefined`, then throw a `TypeError` exception.
7. If `IsDetachedBuffer(buffer)` is true, then throw a `TypeError` exception.
8. If `IsInteger(index)` is false, then return false.
9. If `index = -0`, then return false.
10. Let `length` be the value of `O`’s `[[ArrayLength]]` internal slot.
11. If `index < 0` or `index ≥ length`, then return false.
12. Let `offset` be the value of `O`’s `[[ByteOffset]]` internal slot.
13. Let `arrayTypeName` be the string value of `O`’s `[[TypedArrayName]]` internal slot.
14. Let `elementSize` be the Number value of the Element Size value specified in Table 45 for `arrayTypeName`.
15. Let `indexedPosition = (index × elementSize) + offset`.
16. Let `elementType` be the string value of `O`’s `[[TypedArrayName]]` internal slot.
17. Let `status` be `SetValueInBuffer(buffer, indexedPosition, elementType, numValue)`.
18. ReturnIfAbrupt(`status`).
19. Return true.

9.4.6 Module Exotic Objects

A module object is an exotic object that exposes the bindings exported from an ECMAScript Module (See 15.1.9). There is a one-to-one correspondence between the own properties of a module exotic object and the ExportedBindings of the Module. Each own property name is the StringValue of the corresponding exported binding. These are the only properties of a module exotic object. Each such property has the attributes `[[Configurable]]: false`, `[[Enumerable]]: true`. Module objects are not extensible.

Module objects have the internal slots defined in Table 28.

Table 28 — Internal Slots of Module Exotic Objects

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[ModuleEnvironment]]</code></td>
<td>Environment</td>
<td>The Declarative Environment Record that contains all of the declared top-level bindings for the corresponding module.</td>
</tr>
<tr>
<td><code>[[Exports]]</code></td>
<td>List of String</td>
<td>A List containing the bound names exposed as own properties of this object. The list is ordered as if an Array of the same values had been sorted using <code>Array.prototype.sort</code> using <code>SortCompare</code> as <code>compareFn</code>.</td>
</tr>
</tbody>
</table>

Module exotic objects provide alternative definitions for all of the internal methods.
9.4.6.1 \[[\text{GetPrototypeOf}]\] ( )

When the \[[\text{GetPrototypeOf}]\] internal method of a module exotic object \(O\) is called the following steps are taken:
1. Return null.

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

When the \[[\text{SetPrototypeOf}]\] internal method of a module exotic object \(O\) is called with argument \(V\) the following steps are taken:
1. Assert: Either Type(\(V\)) is Object or Type(\(V\)) is Null.
2. Return false.

9.4.6.3 \[[\text{IsExtensible}]\] ( )

When the \[[\text{IsExtensible}]\] internal method of a module exotic object \(O\) is called the following steps are taken:
1. Return false.

9.4.6.4 \[[\text{PreventExtensions}]\] ( )

When the \[[\text{PreventExtensions}]\] internal method of a module exotic object \(O\) is called the following steps are taken:
1. Return true.

9.4.6.5 \[[\text{GetOwnProperty}]\] (P)

When the \[[\text{GetOwnProperty}]\] internal method of a module exotic object \(O\) is called with property key \(P\), the following steps are taken:
1. Throw a TypeError exception.

9.4.6.6 \[[\text{DefineOwnProperty}]\] (P, Desc)

When the \[[\text{DefineOwnProperty}]\] internal method of a module exotic object \(O\) is called with property key \(P\) and Property Descriptor \(Desc\), the following steps are taken:
1. Return false.

9.4.6.7 \[[\text{HasProperty}]\] (P)

When the \[[\text{HasProperty}]\] internal method of a module exotic object \(O\) is called with property key \(P\), the following steps are taken:
1. Let \(exports\) be the value of \(O\)’s \[[\text{Exports}]\] internal slot.
2. If \(P\) is an element of \(exports\), then return true.
3. Return false.
9.4.6.8  [[Get]] (P, Receiver)

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

1.  Assert: IsPropertyKey(P) is \texttt{true}.
2.  Let \texttt{exports} be the value of O’s [[Exports]] internal slot.
3.  If P is not an element of exports, then return \texttt{undefined}.
4.  Let \texttt{env} be the value of O’s [[ModuleEnvironment]] internal slot.
5.  Return the result of calling the GetBindingValue concrete method of env with arguments P and \texttt{true}.

NOTE Attempting to [[Get]] the value of a module export that has not yet been initialized will throw a ReferenceError exception.

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

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

1.  Return \texttt{false}.

9.4.6.10  [[Delete]] (P)

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

1.  Assert: IsPropertyKey(P) is \texttt{true}.
2.  Let \texttt{exports} be the value of O’s [[Exports]] internal slot.
3.  If P is an element of exports, then return \texttt{false}.
4.  Return \texttt{true}.

9.4.6.11  [[Enumerate]] ()

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

1.  Let \texttt{exports} be the value of O’s [[Exports]] internal slot.
2.  Return CreateListIterator(\texttt{exports}).

9.4.6.12  [[OwnPropertyKeys]] ()

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

1.  Let \texttt{exports} be the value of O’s [[Exports]] internal slot.
2.  Return \texttt{exports}.

9.4.6.13  ModuleObjectCreate (environment, exports)

1.  Assert: \texttt{environment} is a Declarative Environment Record.
2.  Assert: \texttt{exports} is a List of string values.
3.  Let M be a newly created object.
4.  Set M’s essential internal methods to the definitions specified in 9.4.6.
5.  Set M’s [[ModuleEnvironment]] internal slot to \texttt{environment}.

\textbf{Commented [AWB2419]:} TODO: somebody needs to call this
6. Set M’s [[Exports]] internal slot to exports.
7. Return M.

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 objects has an internal slot called [[ProxyHandler]]. The value of [[ProxyHandler]] is always an object, called the proxy’s handler object. Methods (see Table 29) 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.

Table 29 — Proxy Handler Methods

<table>
<thead>
<tr>
<th>Internal Method</th>
<th>Handler Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[GetPrototypeOf]]</td>
<td>getPrototypeOf</td>
</tr>
<tr>
<td>[[SetPrototypeOf]]</td>
<td>setPrototypeOf</td>
</tr>
<tr>
<td>[[IsExtensible]]</td>
<td>isExtensible</td>
</tr>
<tr>
<td>[[PreventExtensions]]</td>
<td>preventExtensions</td>
</tr>
<tr>
<td>[[GetOwnProperty]]</td>
<td>getOwnPropertyDescriptor</td>
</tr>
<tr>
<td>[[HasProperty]]</td>
<td>has</td>
</tr>
<tr>
<td>[[Get]]</td>
<td>get</td>
</tr>
<tr>
<td>[[Delete]]</td>
<td>deleteProperty</td>
</tr>
<tr>
<td>[[DefineOwnProperty]]</td>
<td>defineProperty</td>
</tr>
<tr>
<td>[[Enumerate]]</td>
<td>enumerate</td>
</tr>
<tr>
<td>[[OwnPropertyKeys]]</td>
<td>ownKeys</td>
</tr>
<tr>
<td>[[Call]]</td>
<td>apply</td>
</tr>
<tr>
<td>[[Construct]]</td>
<td>construct</td>
</tr>
</tbody>
</table>

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 initialized when the object is created and typically may not be modified. Some proxy objects are created in a manner that permits them to be 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 ECMAScript 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 `[[GetPrototypeOf]]` ()

When the `[[GetPrototypeOf]]` 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. Assert: `Type(handler)` is Object.
4. Let `target` be the value of the `[[ProxyTarget]]` internal slot of `O`.
5. Let `trap` be `GetMethod(handler, 'getPrototypeOf')`.
6. ReturnIfAbrupt(trap).
7. If `trap` is `undefined`, then
   a. Return the result of calling the `[[GetPrototypeOf]]` internal method of `target`.
8. Let `handlerProto` be the result of calling the `[[Call]]` internal method of `trap` with `handler` as the `this` value and a new List containing `target`.
9. ReturnIfAbrupt(handlerProto).
10. If `Type(handlerProto)` is neither Object nor Null, then throw a `TypeError` exception.
11. Let `extensibleTarget` be `IsExtensible(target)`.
12. ReturnIfAbrupt(extensibleTarget).
13. If `extensibleTarget` is `true`, then return `handlerProto`.
14. Let `targetProto` be the result of calling the `[[GetPrototypeOf]]` internal method of `target`.
15. ReturnIfAbrupt(targetProto).
16. If `SameValue(handlerProto, targetProto)` is `false`, then throw a `TypeError` exception.
17. Return `handlerProto`.

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

9.5.2 `[[SetPrototypeOf]]` (V)

When the `[[SetPrototypeOf]]` 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 `null`.
2. Let `handler` be the value of the `[[ProxyHandler]]` internal slot of `O`.
3. If `handler` is `null`, then throw a `TypeError` exception.
4. Assert: `Type(handler)` is Object.
5. Let `target` be the value of the `[[ProxyTarget]]` internal slot of `O`.
7. ReturnIfAbrupt(trap).
8. If `trap` is `undefined`, then
   a. Return the result of calling the `[[SetPrototypeOf]]` internal method of `target` with argument `V`.
9. 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 `V`.
10. Let `booleanTrapResult` be `ToBoolean(trapResult)`.
11. ReturnIfAbrupt(booleanTrapResult).
12. Let `extensibleTarget` be `IsExtensible(target)`.
13. ReturnIfAbrupt(extensibleTarget).
14. If `extensibleTarget` is `true`, then return `booleanTrapResult`.
15. Let `targetProto` be the result of calling the `[[GetPrototypeOf]]` internal method of `target`.
16. ReturnIfAbrupt(targetProto).
17. If booleanTrapResult is **true** and SameValue(V, targetProto) is **false**, then throw a **TypeError** exception.
18. 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. Assert: Type(handler) is Object.
4. Let target be the value of the [[ProxyTarget]] internal slot of O.
5. Let trap be GetMethod(handler, "isExtensible").
6. ReturnIfAbrupt(trap).
7. If trap is **undefined**, then
   a. Return the result of calling the [[IsExtensible]] internal method of target.
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.
9. Let booleanTrapResult be ToBoolean(trapResult).
10. ReturnIfAbrupt(booleanTrapResult).
11. Let targetIsExtensible be the result of calling the [[IsExtensible]] internal method of target.
12. ReturnIfAbrupt(targetIsExtensible).
13. If SameValue(booleanTrapResult, targetIsExtensible) is **false**, then throw a **TypeError** exception.

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 **null**, then throw a **TypeError** exception.
3. Assert: Type(handler) is Object.
4. Let target be the value of the [[ProxyTarget]] internal slot of O.
5. Let trap be GetMethod(handler, "preventExtensions").
6. ReturnIfAbrupt(trap).
7. If trap is **undefined**, then
   a. Return the result of calling the [[PreventExtensions]] internal method of target.
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.
9. Let booleanTrapResult be ToBoolean(trapResult).
10. ReturnIfAbrupt(booleanTrapResult).
11. If booleanTrapResult is **true**, then
    a. Let targetIsExtensible be the result of calling the [[IsExtensible]] internal method of target.
    b. ReturnIfAbrupt(targetIsExtensible).
If `targetIsExtensible` is `true`, then throw a `TypeError` exception.

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

NOTE: `[[GetOwnProperty]]` for proxy objects enforces the following invariants:
- The result of `[[GetOwnProperty]]` must be either an `Object` or `undefined`.
- A property cannot be reported as non-existent, if it exists as a non-configurable own property of the target object.
- A property cannot be reported as non-existent, if it exists as 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 exists as an own property of the target object and the target object is not extensible.

A property cannot be reported as non-configurable, if it does not exists as 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.
2. Let handler be the value of the [[ProxyHandler]] internal slot of O.
3. If handler is null, then throw a TypeError exception.
4. Assert: Type(handler) is Object.
5. Let target be the value of the [[ProxyTarget]] internal slot of O.
6. Let trap be GetMethod(handler, "defineProperty").
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. 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.
10. If booleanTrapResult is false, then return false.
11. Let targetDesc be the result of calling the [[GetOwnProperty]] internal method of target with argument P.
12. If targetDesc is undefined, then
    a. If IsConfigurable(desc) is false, then throw a TypeError exception.
   b. If settingConfigFalse is true, then throw a TypeError exception.
13. If extensibleTarget is 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.

**NOTE**

- A property cannot be added, if the target object is not extensible.
- A property cannot be added or modified to be non-configurable, if it does not exists as a non-configurable own property of the target object.
- A property may not be non-configurable, if 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. Assert: Type(handler) is Object.
5. Let target be the value of the [[ProxyTarget]] internal slot of O.
6. Let trap be GetMethod(handler, "has").
7. ReturnIfAbrupt(trap).
8. If trap is undefined, then
   a. Return the result of calling the [[HasProperty]] internal method of target with argument P.
9. 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.
10. Let booleanTrapResult be ToBoolean(trapResult).
11. ReturnIfAbrupt(booleanTrapResult).
12. 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 IsExtensible(target).
      iii. ReturnIfAbrupt(extensibleTarget).
      iv. If extensibleTarget is false, then throw a TypeError exception.
13. 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. Assert: Type(handler) is Object.
5. Let target be the value of the [[ProxyTarget]] internal slot of O.
6. Let trap be GetMethod(handler, "get").
7. ReturnIfAbrupt(trap).
8. If trap is undefined, then
   a. Return the result of calling the [[Get]] internal method of target with arguments P and Receiver.
9. 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.
10. ReturnIfAbrupt(trapResult).
11. Let targetDesc be the result of calling the [[GetOwnProperty]] internal method of target with argument P.
12. ReturnIfAbrupt(targetDesc).
13. 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.

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 null, then throw a TypeError exception.
4. Assert: Type(handler) is Object.
5. Let target be the value of the [[ProxyTarget]] internal slot of O.
6. Let trap be GetMethod(handler, "set").
7. ReturnIfAbrupt(trap).
8. If trap is undefined, then
   a. Return the result of calling the [[Set]] internal method of target with arguments P, V, and Receiver.
9. 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, V, and Receiver.
10. Let booleanTrapResult be ToBoolean(trapResult).
11. ReturnIfAbrupt(booleanTrapResult).
12. If booleanTrapResult is false, then return false.
13. Let targetDesc be the result of calling the [[GetOwnProperty]] internal method of target with
    argument P.
14. ReturnIfAbrupt(targetDesc).
15. If targetDesc is not undefined, then
   a. If IsDataDescriptor(targetDesc) and targetDesc.[[Configurable]] is false and
      targetDesc.[[Writable]] is false, then
      i. If SameValue(V, targetDesc.[[Value]]) is false, then throw a TypeError exception.
   b. If IsAccessorDescriptor(targetDesc) and targetDesc.[[Configurable]] is false, then
      i. If targetDesc.[[Set]] is undefined, then throw a TypeError exception.
16. Return 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 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 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. Assert: Type(handler) is Object.
5. Let target be the value of the [[ProxyTarget]] internal slot of O.
6. Let trap be GetMethod(handler, "deleteProperty").
7. ReturnIfAbrupt(trap).
8. If trap is undefined, then
   a. Return the result of calling the [[Delete]] internal method of target with argument P.
9. 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.
10. Let booleanTrapResult be ToBoolean(trapResult).
11. ReturnIfAbrupt(booleanTrapResult).
12. If booleanTrapResult is false, then return false.
13. Let targetDesc be the result of calling the [[GetOwnProperty]] internal method of target with argument P.
14. ReturnIfAbrupt(targetDesc).
15. If targetDesc is undefined, then return true.
16. If targetDesc.[[Configurable]] is false, then throw a TypeError exception.
17. Return true.

NOTE [[Delete]] for proxy objects enforces the following invariant:
- A property cannot be deleted, if it exists as a non-configurable own property of the target object.

9.5.11 [[Enumerate]] ()

When the [[Enumerate]] 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. Assert: Type(handler) is Object.
4. Let target be the value of the [[ProxyTarget]] internal slot of O.
5. Let trap be GetMethod(handler, "enumerate").
6. ReturnIfAbrupt(trap).
7. If trap is undefined, then
   a. Return the result of calling the [[Enumerate]] internal method of target.
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.
9. ReturnIfAbrupt(trapResult).
10. If Type(trapResult) is not Object, then throw a TypeError exception.
11. 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 \( \text{handler} \) be the value of the [[ProxyHandler]] internal slot of \( O \).
2. If \( \text{handler} \) is \( \text{null} \), then throw a TypeError exception.
3. Assert: Type(\( \text{handler} \)) is Object.
4. Let \( \text{target} \) be the value of the [[ProxyTarget]] internal slot of \( O \).
5. Let \( \text{trap} \) be GetMethod(\( \text{handler} \), \"ownKeys\")
6. ReturnIfAbrupt(\( \text{trap} \)).
7. If \( \text{trap} \) is undefined, then
   a. Return the result of calling the [[OwnPropertyKeys]] internal method of \( \text{target} \).
8. Let \( \text{trapResultArray} \) be the result of calling the [[Call]] internal method of \( \text{trap} \) with \( \text{handler} \) as the this value and a new List containing \( \text{target} \).
9. Let \( \text{extensibleTarget} \) be IsExtensible(\( \text{target} \)).
10. ReturnIfAbrupt(\( \text{extensibleTarget} \)).
11. Let \( \text{targetKeys} \) be the result of calling the [[OwnPropertyKeys]] internal method of \( \text{target} \).
12. ReturnIfAbrupt(\( \text{targetKeys} \)).
13. Assert: \( \text{targetKeys} \) is a List containing only String and Symbol values.
14. Let \( \text{targetLen} \) be the number of elements in \( \text{targetKeys} \).
15. Let \( \text{targetNonconfigurableKeys} \) be an empty List.
16. Let \( \text{targetConfigurableKeys} \) be an empty List.
17. Repeat, for each element \( \text{key} \) of \( \text{targetKeys} \),
   a. Let \( \text{desc} \) be the result of calling the [[GetOwnProperty]] internal method of \( \text{target} \) with argument \( \text{key} \).
   b. ReturnIfAbrupt(\( \text{desc} \)).
   c. If \( \text{desc} \) is not undefined and \( \text{desc}.[[\text{Configurable}]] \) is false, then
      i. Append \( \text{key} \) as an element of \( \text{targetNonconfigurableKeys} \).
   d. Else,
      i. Append \( \text{key} \) as an element of \( \text{targetConfigurableKeys} \).
20. If \( \text{extensibleTarget} \) is true and \( \text{targetNonconfigurableKeys} \) is empty, then
   a. Return \( \text{trapResult} \).
21. Let \( \text{uncheckedResultKeys} \) be a new List which is a copy of \( \text{trapResult} \).
22. Repeat, for each key that is an element of \( \text{targetNonconfigurableKeys} \),
   a. If \( \text{key} \) is not an element of \( \text{uncheckedResultKeys} \), then throw a TypeError exception.
   b. Remove key from \( \text{uncheckedResultKeys} \)
23. If \( \text{extensibleTarget} \) is true, then return \( \text{trapResult} \).
24. Repeat, for each key that is an element of \( \text{targetConfigurableKeys} \),
   a. If \( \text{key} \) is not an element of \( \text{uncheckedResultKeys} \), then throw a TypeError exception.
   b. Remove key from \( \text{uncheckedResultKeys} \)
25. If \( \text{uncheckedResultKeys} \) is not empty, then throw a TypeError exception.
26. Return \( \text{trapResult} \).

NOTE [[OwnPropertyKeys]] for proxy objects enforces the following invariants:
- The result of [[OwnPropertyKeys]] is a List.
- The Type of each result List element is either String or Symbol.
- The result List must contain the keys of all non-configurable own properties of the target object.
- If the target object is not extensible, then the result List must contain all the keys of the own properties of the target object and no other values.
9.5.13 [[Call]] (thisArgument, argumentsList)

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. Assert: `Type(handler)` is `Object`.
4. Let `target` be the value of the `[[ProxyTarget]]` internal slot of `O`.
5. Let `trap` be `GetMethod(handler, "apply")`.
6. `ReturnIfAbrupt(trap)`.
7. If `trap` is `undefined`, then
   a. `Return the result of calling the [[Call]] internal method of target with arguments thisArgument and argumentsList`.
8. `Let argArray be CreateArrayFromList(argumentsList)`.
9. `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 A 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. Assert: `Type(handler)` is `Object`.
4. Let `target` be the value of the `[[ProxyTarget]]` internal slot of `O`.
5. Let `trap` be `GetMethod(handler, "construct")`.
6. `ReturnIfAbrupt(trap)`.
7. 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`.
8. `Let argArray be CreateArrayFromList(argumentsList)`.
9. `Let newObj be the result of calling trap with handler as the this value and a new List containing target and argArray`.
10. `ReturnIfAbrupt(newObj)`.
11. If `Type(newObj)` is not `Object`, then throw a `TypeError` exception.
12. `Return newObj`.

NOTE 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 [[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 (except for `[[Call]]` and `[[Construct]])` 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 normalization of text, or behave as though they were performing normalization 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 code point U+000A is line feed) and therefore the next code point 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 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-16Encoding

The UTF-16Encoding of a numeric code point value, `cp`, is determined as follows:
1. Assert: \(0 \leq \text{cp} \leq 0x10FFFF\).
2. If \(\text{cp} \leq 65535\), then return \(\text{cp}\).
3. Let \(cu1\) be \(\text{floor}(\text{cp} – 65536) / 1024\) + 0xD800.
4. Let \(cu2\) be \(((\text{cp} – 65536) \mod 1024) + 0xDC00\).
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. Let \(cp\) be \((lead – 0xD800) \times 1024 + (trail – 0xDC00) + 0x10000\).
3. 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, GeneratorBody, 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 `[[ECMAScriptCode]]` internal slot (see 9.1.14) of function and generator objects. It includes the code that defines and initializes the formal parameters of the function. 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 initialized. 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 14.1.1).
Module code is always strict code.

All parts of a `ClassDeclaration` or a `ClassExpression` are 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`, `GeneratorDeclaration`, `GeneratorExpression`, `MethodDefinition`, or `ArrowFunction` is strict function code if its `GeneratorDeclaration`, `GeneratorExpression`, `MethodDefinition`, or `ArrowFunction` is contained in strict mode code or if its `FunctionBody` 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 `UseStrictDirective`.

### 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 code units 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:

```javascript
a = b

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

where the first non-whitespace, non-comment code point 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);
```

Commented [AWB921]: May need to also say something about TemplateSubstitution tail. Also need to consider with there are any ASI issues concerning it.
Syntax

InputElementDiv ::
   WhiteSpace
   LineTerminator
   Comment
   Token
   DivPunctuator
   RightBracePunctuator

InputElementRegExp ::
   WhiteSpace
   LineTerminator
   Comment
   Token
   RightBracePunctuator
   RegularExpressionLiteral

InputElementTemplateTail ::
   WhiteSpace
   LineTerminator
   Comment
   Token
   DivPunctuator
   TemplateSubstitutionTail

11.1 Unicode Format-Control Characters

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

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

U+200C (ZERO WIDTH NON-JOINER) and U+200D (ZERO WIDTH JOINER) are format-control characters that are used to make necessary distinctions when forming words or phrases in certain languages. In ECMAScript source text these code points may also be used in an IdentifierName (see 11.6.1) after the first character.

U+FEFF (BYTE ORDER MARK) is a format-control character used primarily at the start of a text to mark it as Unicode and to allow detection of the text's encoding and byte order. <BOM> characters intended for this purpose can sometimes also appear after the start of a text, for example as a result of concatenating files. In ECMAScript source text <BOM> code points are treated as white space characters (see 11.2).

The special treatment of certain format-control characters outside of comments, string literals, and regular expression literals is summarized in Table 30.
11.2 White Space

White space code points are used to improve source text readability and to separate tokens (indivisible lexical units) from each other, but are otherwise insignificant. White space code points may occur between any two tokens and at the start or end of input. White space code points may occur within a StringLiteral, a RegularExpressionLiteral, a Template, or a TemplateSubstitutionTail where they are considered significant code points 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 code points are listed in Table 31.

<table>
<thead>
<tr>
<th>Code Point</th>
<th>Name</th>
<th>Abbreviation</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+0009</td>
<td>CHARACTER TABULATION</td>
<td>&lt;TAB&gt;</td>
<td></td>
</tr>
<tr>
<td>U+000B</td>
<td>LINE TABULATION</td>
<td>&lt;VT&gt;</td>
<td></td>
</tr>
<tr>
<td>U+000C</td>
<td>FORM FEED</td>
<td>&lt;FF&gt;</td>
<td></td>
</tr>
<tr>
<td>U+0020</td>
<td>SPACE</td>
<td>&lt;SP&gt;</td>
<td></td>
</tr>
<tr>
<td>U+00A0</td>
<td>NO-BREAK SPACE</td>
<td>&lt;NBSP&gt;</td>
<td></td>
</tr>
<tr>
<td>U+FEFF</td>
<td>BYTE ORDER MARK</td>
<td>&lt;BOM&gt;</td>
<td></td>
</tr>
<tr>
<td>Other category “Zs”</td>
<td>Any other Unicode “Separator, space” code point</td>
<td>&lt;USP&gt;</td>
<td></td>
</tr>
</tbody>
</table>

ECMAScript implementations must recognize as White space code points listed in the “Separator, space” (Zs) category by Unicode 5.1. ECMAScript implementations may also recognize as White space additional category Zs code points from subsequent editions of the Unicode Standard.

NOTE Other than for the code points listed in Table 31, ECMAScript White space intentionally excludes all code points that have the Unicode “White_Space” property but which are not classified in category “Zs”.

Syntax

WhiteSpace ::

<TAB>
<VT>
<FF>
<SP>
<NBSP>
<BOM>
<USP>
11.3 Line Terminators

Like white space code points, line terminator code points are used to improve source text readability and to separate tokens (indivisible lexical units) from each other. However, unlike white space code points, 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 code points that are matched by the \s class in regular expressions.

The ECMAScript line terminator code points are listed in Table 32.

Table 32 — Line Terminator Code Points

<table>
<thead>
<tr>
<th>Code Point</th>
<th>Unicode Name</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+000A</td>
<td>LINE FEED</td>
<td>\n</td>
</tr>
<tr>
<td>U+000D</td>
<td>CARRIAGE RETURN</td>
<td>\r</td>
</tr>
<tr>
<td>U+2028</td>
<td>LINE SEPARATOR</td>
<td>\t</td>
</tr>
<tr>
<td>U+2029</td>
<td>PARAGRAPH SEPARATOR</td>
<td>\p</td>
</tr>
</tbody>
</table>

Only the Unicode code points in Table 32 are treated as line terminators. Other new line or line breaking Unicode code points are not treated as line terminators but are treated as white space if they meet the requirements listed in Table 31. The sequence \r\n is commonly used as a line terminator. It should be considered a single SourceCharacter for the purpose of reporting line numbers.

Syntax

LineTerminator ::
\n\r\t\p

LineTerminatorSequence ::
\n\r\t\p\n
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 code point, and because of the general rule that a token is always as long as possible, a single-line comment always consists of all code points 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 recognized 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 code point, then the entire comment is considered to be a LineTerminator for purposes of parsing by the syntactic grammar.

**Syntax**

Comment ::
  MultiLineComment
  SingleLineComment

MultiLineComment ::
  /* MultiLineCommentCharsopt */

MultiLineCommentChars ::
  MultiLineNotAsteriskChar 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

**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` 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`. The syntactic grammar defines `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 code points defined in later editions of the Unicode Standard.

NOTE 1 This standard specifies specific code point additions: U+0024 (DOLLAR SIGN) and U+005F (LOW LINE) are permitted anywhere in an `IdentifierName`, and the characters U+200D (ZERO-WIDTH NON-JOINER) and U+200D (ZERO-WIDTH JOINER) are permitted anywhere after the first code unit 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 u and { code units, 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 code point.

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

Syntax

```
IdentifierName ::
  IdentifierStart
  IdentifierName IdentifierPart

IdentifierStart ::
  UnicodeIDStart
  $  UnicodeEscapeSequence

IdentifierPart ::
  UnicodeIDContinue
  $  UnicodeEscapeSequence

<ZWNJ>
<ZWJ>
```
UnicodeIDStart ::
    any Unicode code point with the Unicode property “ID_Start” or “Other_ID_Start”

UnicodeIDContinue ::
    any Unicode code point with the Unicode property “ID_Continue”, “Other_ID_Continue”, or “Other_ID_Start”

The definitions of the nonterminal UnicodeEscapeSequence is given in 11.8.4.

11.6.1 Identifier Names

11.6.1.1 Static Semantics: Early Errors

IdentifierStart :: \ UnicodeEscapeSequence
    • It is a Syntax Error if SV(UnicodeEscapeSequence) is neither the UTF-16Encoding (10.1.1) of a single Unicode code point with the Unicode property "ID_Start" nor "$" or "_".

IdentifierPart :: \ UnicodeEscapeSequence
    • It is a Syntax Error if SV(UnicodeEscapeSequence) is neither the UTF-16Encoding (10.1.1) of a single Unicode code point with the Unicode property "ID_Continue" nor "$" or "_" nor the UTF-16Encoding of either <ZWNJ> or <ZWJ>.

11.6.1.2 Static Semantics: StringValue

See also: 11.8.4.2, 12.1.3.

IdentifierName ::
    IdentifierStart
    IdentifierName IdentifierPart

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

NOTE  The ReservedWord definitions are specified as literal sequences of specific SourceCharacter elements. A code point in a ReservedWord cannot be expressed by a \ UnicodeEscapeSequence.
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

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>do</td>
</tr>
<tr>
<td>case</td>
<td>else</td>
</tr>
<tr>
<td>catch</td>
<td>export</td>
</tr>
<tr>
<td>class</td>
<td>extends</td>
</tr>
<tr>
<td>const</td>
<td>finally</td>
</tr>
<tr>
<td>continue</td>
<td>for</td>
</tr>
<tr>
<td>debugger</td>
<td>function</td>
</tr>
<tr>
<td>default</td>
<td>if</td>
</tr>
<tr>
<td>delete</td>
<td>import</td>
</tr>
</tbody>
</table>

NOTE In some contexts yield is given the semantics of an Identifier. See 12.1.1. In strict mode code, let and static are treated as reserved keywords through static semantic restrictions (see 12.1.1, 1.1.1, 13.2.1.1, 13.6.4.1, and 14.5.1) rather than the lexical grammar.

11.6.2.2 Future Reserved Words

The following tokens are reserved for use as keywords in future language extensions.

Syntax

FutureReservedWord ::

<table>
<thead>
<tr>
<th>FutureReservedWord</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>enum</td>
<td></td>
</tr>
<tr>
<td>await</td>
<td></td>
</tr>
</tbody>
</table>

await is only treated as a FutureReservedWord when Module is the goal symbol of the syntactic grammar.

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 12.1.1) rather than the lexical grammar:

<table>
<thead>
<tr>
<th>FutureReservedWord</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>implements</td>
<td>package</td>
</tr>
<tr>
<td>interface</td>
<td>private</td>
</tr>
<tr>
<td>protected</td>
<td>public</td>
</tr>
</tbody>
</table>
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 ::

DecimalLiteral
  BinaryIntegerLiteral
  OctalIntegerLiteral
  HexIntegerLiteral

DecimalLiteral ::

DecimalIntegerLiteral . DecimalDigitsExponentPart
  DecimalDigits ExponentPart

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?
DecimalIntegerLiteral ::
    0
    NonZeroDigit DecimalDigits opt

DecimalDigits ::
    DecimalDigit
    DecimalDigits DecimalDigit

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

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

ExponentPart ::
    ExponentIndicator SignedInteger

ExponentIndicator :: one of
    e E

SignedInteger ::
    DecimalDigits
    + DecimalDigits
    - DecimalDigits

BinaryIntegerLiteral ::
    0b 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

Commented [AWB724]: The various Digit productions could be refactored to have less redundancy.
HexDigits ::
  HexDigit
  HexDigits HexDigit

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

The SourceCharacter immediately following a NumericLiteral must not be an IdentifierStart or DecimalDigit.

NOTE For example:
3in is an error and not the two input elements 3 and in.

A conforming implementation, when processing strict mode code (see 10.2.1), must not extend the syntax of NumericLiteral to include LegacyOctalIntegerLiteral as described in B.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 is the MV of DecimalIntegerLiteral.
- The MV of DecimalLiteral :: DecimalIntegerLiteral ExponentPart is the MV of DecimalIntegerLiteral \times 10^e, where e is the MV of ExponentPart.
- The MV of DecimalIntegerLiteral :: DecimalIntegerLiteral DecimalDigit is the MV of DecimalIntegerLiteral plus (the MV of DecimalDigits \times 10^n), where n is the number of code points in DecimalDigits and e is the MV of ExponentPart.
- The MV of DecimalDigits :: DecimalDigit is the MV of DecimalDigit.
- The MV of DecimalDigits :: DecimalDigits DecimalDigit is the MV of DecimalDigits plus (the MV of DecimalDigits \times 10^n), where n is the number of code points in DecimalDigits and e is the MV of ExponentPart.
- The MV of ExponentPart :: ExponentIndicator SignedInteger is the MV of SignedInteger.
• The MV of SignedInteger :: DecimalDigits is the MV of DecimalDigits.
• The MV of SignedInteger :: + DecimalDigits is the MV of DecimalDigits.
• The MV of SignedInteger :: - DecimalDigits is the negative of the MV of DecimalDigits.
• The MV of DecimalDigit :: 0 or of HexDigit :: 0 or of OctalDigit :: 0 or of BinaryDigit :: 0 is 0.
• The MV of DecimalDigit :: 1 or of NonZeroDigit :: 1 or of HexDigit :: 1 or of OctalDigit :: 1 or of BinaryDigit :: 1 is 1.
• The MV of DecimalDigit :: 2 or of NonZeroDigit :: 2 or of HexDigit :: 2 or of OctalDigit :: 2 is 2.
• The MV of DecimalDigit :: 3 or of NonZeroDigit :: 3 or of HexDigit :: 3 or of OctalDigit :: 3 is 3.
• The MV of DecimalDigit :: 4 or of NonZeroDigit :: 4 or of HexDigit :: 4 or of OctalDigit :: 4 is 4.
• The MV of DecimalDigit :: 5 or of NonZeroDigit :: 5 or of HexDigit :: 5 or of OctalDigit :: 5 is 5.
• The MV of DecimalDigit :: 6 or of NonZeroDigit :: 6 or of HexDigit :: 6 or of OctalDigit :: 6 is 6.
• The MV of DecimalDigit :: 7 or of NonZeroDigit :: 7 or of HexDigit :: 7 or of OctalDigit :: 7 is 7.
• The MV of DecimalDigit :: 8 or of NonZeroDigit :: 8 or of HexDigit :: 8 is 8.
• The MV of DecimalDigit :: 9 or of NonZeroDigit :: 9 or of HexDigit :: 9 is 9.
• The MV of HexDigit :: a or of HexDigit :: A is 10.
• The MV of HexDigit :: b or of HexDigit :: B is 11.
• The MV of HexDigit :: c or of HexDigit :: C is 12.
• The MV of HexDigit :: d or of HexDigit :: D is 13.
• The MV of HexDigit :: e or of HexDigit :: E is 14.
• The MV of HexDigit :: f or of HexDigit :: F is 15.
• The MV of BinaryIntegerLiteral :: 0b BinaryDigits is the MV of BinaryDigits.
• The MV of BinaryIntegerLiteral :: 0B BinaryDigits is the MV of BinaryDigits.
• The MV of BinaryDigits :: BinaryDigit is the MV of BinaryDigit.
• The MV of BinaryDigits :: BinaryDigits BinaryDigit is (the MV of BinaryDigits × 2) plus the MV of BinaryDigit.
• The MV of OctalIntegerLiteral :: 0o OctalDigits is the MV of OctalDigits.
• The MV of OctalIntegerLiteral :: 0O OctalDigits is the MV of OctalDigits.
• The MV of OctalDigits :: OctalDigit is the MV of OctalDigit.
• The MV of OctalDigits :: OctalDigits OctalDigit is (the MV of OctalDigits × 8) plus the MV of OctalDigit.
• The MV of HexIntegerLiteral :: 0x HexDigits is the MV of HexDigits.
• The MV of HexIntegerLiteral :: 0X HexDigits is the MV of HexDigits.
• The MV of HexDigits :: HexDigit is the MV of HexDigit.
• The MV of HexDigits :: HexDigits HexDigit is (the MV of HexDigits × 16) plus the MV of HexDigit.

Once the exact MV for a numeric literal has been determined, it is then rounded to a value of the Number type. If the MV is 0, then the rounded value is +0; otherwise, the rounded value must be the Number value for the MV (as specified in 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; or
• 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 code points may appear literally in a string literal except for the closing quote code points, backslash, carriage return, line separator, paragraph separator, and line feed. Any code points 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

```plaintext
StringLiteral ::
  " DoubleStringCharactersopt "
  ' SingleStringCharactersopt '

DoubleStringCharacters ::
  DoubleStringCharacter DoubleStringCharactersopt
SingleStringCharacters ::
  SingleStringCharacter SingleStringCharactersopt

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

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

LineContinuation ::
  \ LineTerminatorSequence

EscapeSequence ::
  CharacterEscapeSequence
  HexEscapeSequence
  UnicodeEscapeSequence

CharacterEscapeSequence::
  SingleEscapeCharacter
  NonEscapeCharacter

SingleEscapeCharacter ::
  one of " \ b f n r t v

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

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 B.1.2.
EscapeCharacter ::
    SingleEscapeCharacter
    DecimalDigit
    \u

HexEscapeSequence ::
    \ HexDigit HexDigit

UnicodeEscapeSequence ::
    u Hex4Digits
    u { HexDigits }

Hex4Digits ::
    HexDigit HexDigit HexDigit HexDigit

The definition of the nonterminal \textit{HexDigit} is given in 11.8.3. \textit{SourceCharacter} is defined in 10.1.

\textbf{NOTE} A line terminator code point cannot appear in a string literal, except as part of a \textit{LineContinuation} to produce the empty code points sequence. The correct way to cause a line terminator code points to be part of the String value of a string literal is to use an escape sequence such as \texttt{\textbackslash n} or \texttt{\textbackslash u000A}.

11.8.4.1 Static Semantics: Early Errors

UnicodeEscapeSequence :: u { HexDigits }

- It is a Syntax Error if the \textit{MV} of \textit{HexDigits} > 1114111.

11.8.4.2 Static Semantics: StringValue

See also: 11.6.1.2, 12.1.3.

StringValue ::
    " DoubleStringCharacters\texttt{\textbackslashq}" \\
    ' SingleStringCharacters\texttt{\textbackslashq}''

1. Return the String value whose elements are the \textit{SV} of this \textit{StringValue}.

11.8.4.3 Static Semantics: \textit{SV}'s and \textit{CV}'s

A string literal stands for a value of the String type. The String value (\textit{SV}) of the literal is described in terms of code unit values (\textit{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 (\textit{MV}), as described below or in 11.8.3.

- The \textit{SV} of \textit{StringLiteral} :: " " is the empty code unit sequence.
- The \textit{SV} of \textit{StringLiteral} :: " ' is the empty code unit sequence.
- The \textit{SV} of \textit{StringLiteral} :: " DoubleStringCharacters " is the \textit{SV} of \textit{DoubleStringCharacters}.
- The \textit{SV} of \textit{StringLiteral} :: ' SingleStringCharacters ' is the \textit{SV} of \textit{SingleStringCharacters}.
- The \textit{SV} of \textit{DoubleStringCharacters} :: DoubleStringCharacter is a sequence of one or two code units that is the \textit{CV} of \textit{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-16Encoding (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 code unit sequence.

The CV of SingleStringCharacter :: SourceCharacter but not one of ' or \ or LineTerminator is the UTF-16Encoding (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 code unit 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 code unit whose value is determined by the SingleEscapeCharacter according to Table 33.

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Code Unit Value</th>
<th>Unicode Character 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>CHARACTER TABULATION</td>
<td>&lt;HT&gt;</td>
</tr>
<tr>
<td>\n</td>
<td>0x000A</td>
<td>LINE FEED</td>
<td>&lt;LF&gt;</td>
</tr>
<tr>
<td>\v</td>
<td>0x000B</td>
<td>LINE TABULATION</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>QUOTATION MARK</td>
<td>&quot;</td>
</tr>
<tr>
<td>'</td>
<td>0x0027</td>
<td>APOSTROPHE</td>
<td>'</td>
</tr>
<tr>
<td>\</td>
<td>0x005C</td>
<td>REVERSE SOLIDUS</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-16Encoding (10.1.1) of the code point value of SourceCharacter.

The CV of HexEscapeSequence :: x HexDigit HexDigit is the code unit value that is (16 times the MV of the first HexDigit) plus the MV of the second HexDigit.

The CV of UnicodeEscapeSequence :: a HexDigit is the CV of HexDigit.

The CV of HexDigit :: HexDigit HexDigit HexDigit HexDigit is the code unit value that is (4096 times the MV of the first HexDigit) plus (256 times the MV of the second HexDigit) plus (16 times the MV of the third HexDigit) plus the MV of the fourth HexDigit.
• The CV of UnicodeEscapeSequence :: u{ HexDigits } is the UTF-16Encoding (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 === to each other even if the two literals’ contents are identical. A RegExp object may also be created at runtime by new RegExp (see 21.2.3.2) or calling the 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 RegularExpressionBody and the 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 RegularExpressionBody and RegularExpressionFlags productions defined below or the productions used by these productions.

Syntax

RegularExpressionLiteral ::
    / RegularExpressionBody / RegularExpressionFlags

RegularExpressionBody ::
    RegularExpressionFirstChar RegularExpressionChars

RegularExpressionChars ::
    [empty]
    RegularExpressionChars RegularExpressionChar

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

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

RegularExpressionBackslashSequence ::
    \ RegularExpressionNonTerminator

RegularExpressionNonTerminator ::
    SourceCharacter but not LineTerminator

RegularExpressionClass ::
    [ RegularExpressionClassChars ]

RegularExpressionClassChars ::
    [empty]
    RegularExpressionClassChars RegularExpressionClassChar
RegularExpressionClassChar ::
  RegularExpressionNonTerminator but not one of ] or \\
  RegularExpressionBackslashSequence

RegularExpressionFlags ::
  [empty]
  RegularExpressionFlags IdentifierPart

NOTE Regular expression literals may not be empty; instead of representing an empty regular expression literal, the code unit sequence // starts 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 recognized as RegularExpressionBody.

11.8.5.3 Static Semantics: FlagText

RegularExpressionLiteral :: / RegularExpressionBody / RegularExpressionFlags
  1. Return the source code that was recognized 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 TemplateCharacters opt

TemplateCharacter ::
    $ [lookahead # { ]
    \ EscapeSequence
    LineContinuation
    LineTerminatorSequence
    SourceCharacter  but not one of ` or \ or $ or LineTerminator

A conforming implementation must not use the extended definition of EscapeSequence described in B.1.2 when parsing a TemplateCharacter.

NOTE  TemplateSubstitutionTail is used by the InputElementTemplateTail alternative lexical goal.

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 TemplateCharacters :: TemplateCharacter TemplateCharacters is a sequence consisting of the code units in the TV of TemplateCharacter followed by all the code units in the TV of TemplateCharacters in order.
- The TV of TemplateCharacter :: SourceCharacter but not one of ` or \ or $ or LineTerminatorSequence is the UTF-16Encoding (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 CV of LineContinuation.
- The TV of TemplateCharacter :: LineTerminatorSequence is the TRV of LineTerminatorSequence.
- The TV of NoSubstitutionTemplate :: TemplateCharacters ` is the TRV of TemplateCharacters.
- The TV of TemplateHead :: TemplateCharacters $ { is the TRV of TemplateCharacters.
- The TV of TemplateMiddle :: TemplateCharacters $ { is the TRV of TemplateCharacters.
- The TV of TemplateTail :: TemplateCharacters $ is the TRV of TemplateCharacters.
• The TRV of TemplateCharacters :: TemplateCharacter is the TRV of TemplateCharacter.
• The TRV of TemplateCharacters :: TemplateCharacter TemplateCharacters is a sequence consisting of the code units in the TRV of TemplateCharacter followed by all the code units in the TRV of TemplateCharacters, in order.
• The TRV of TemplateCharacter :: SourceCharacter but not one of ` or \ or $ 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 0x003C 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 :: \ HexDigit HexDigit is the code unit value 0x003C followed by the code units of TRV of HexDigit.
• 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 code point.
• 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 :: \ HexDigits is the sequence consisting of code unit value 0x0075 followed by TRV of HexDigits.
• The TRV of UnicodeEscapeSequence :: u ( HexDigits ) \ is the sequence consisting of code unit value 0x0075 followed by code unit value 0x007B followed by TRV of HexDigits followed by code unit value 0x007D.
• The TRV of HexDigits :: HexDigit HexDigit HexDigit HexDigit is the sequence consisting of the TRV of the first HexDigit followed by the TRV of the second HexDigit followed by the TRV of the third HexDigit followed by the TRV of the fourth HexDigit.
• The TRV of HexDigits :: HexDigit is the TRV of HexDigit.
• The TRV of HexDigits :: HexDigits HexDigit is the sequence consisting of TRV of HexDigits followed by TRV of HexDigit.
• The TRV of a HexDigit is the CV of the SourceCharacter that is that HexDigit.
• The TRV of LineContinuation :: \ LineTerminatorSequence is the sequence consisting of the code unit value 0x001C 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 0x000A.
• The TRV of LineTerminatorSequence :: \ <LS> is the code unit value 0x2028.
• The TRV of LineTerminatorSequence :: \ <PS> is the code unit value 0x2029.
• The TRV of LineTerminatorSequence :: \ <CR><LF> is the sequence consisting of the code unit value 0x000A.

NOTE TV excludes the code units of LineContinuation while TRV includes them. \ <CR><LF> and \ <CR> LineTerminatorSequences 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`, `const`, `import`, `export`, and module 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, as the script is parsed from left to right, a token is encountered that is allowed by some production of the grammar, but the production is a restricted production and the token would be the first token for a terminal or nonterminal immediately following the annotation "[no LineTerminator here]" within the restricted production (and therefore such a token is called a restricted token), and the restricted token is separated from the previous token by at least one `LineTerminator`, then a semicolon is automatically inserted before the restricted token.

However, there is an additional overriding condition on the preceding rules: a semicolon is never inserted automatically if the semicolon would then be parsed as an empty statement or if that semicolon would become one of the two semicolons in the header of a `for` statement (see 13.6.3).

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

- `PrefixExpression{Yield}: LeftHandSideExpression{Yield} [no LineTerminator here] ++`  
- `PrefixExpression{Yield}: LeftHandSideExpression{Yield} [no LineTerminator here] --`

- `ContinueStatement{Yield}: continue;`  
  - `continue [no LineTerminator here] LabelIdentifier{Yield};`

- `BreakStatement{Yield}: break;`  
  - `break [no LineTerminator here] LabelIdentifier{Yield};`

- `ReturnStatement{Yield}: return;`  
  - `return [no LineTerminator here] Expression;`  
  - `return [no LineTerminator here] Expression{Yield};`
ThrowStatement\[\textit{Yield}\] :
  \textit{throw} \text{[no LineTerminator here]} \text{Expression} \text{[\textit{Yield}];}

ArrowFunction\[\textit{Yield}\] :
  \text{ArrowParameters} \[\textit{Yield}\] \text{=>} \text{ConciseBody} \[\textit{In}\]

YieldExpression\[\textit{In}\] :
  \textit{yield} \text{[no LineTerminator here]} * \text{Lexical goal InputElementRegExp AssignmentExpression} \text{[\textit{In, Yield}]}
  \textit{yield} \text{[no LineTerminator here]} * \text{Lexical goal InputElementRegExp AssignmentExpression} \text{[\textit{In, Yield}]}

ModuleImport :
  \text{module} \text{[no LineTerminator here]} \text{ImportedBinding 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
  \text{for} (a; b \\
  \text{)
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

```javascript
return
  a + b
```

is transformed by automatic semicolon insertion into the following:

```javascript
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

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

is transformed by automatic semicolon insertion into the following:

```javascript
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

```javascript
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

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

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

```javascript
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 Identifiers

Syntax

IdentifierReference[Yield] : Identifier
[Yield] yield

BindingIdentifier[Default, Yield] :
[+Default] default
[Yield] yield
Identifier

LabelIdentifier[Yield] :
Identifier
[Yield] yield

Identifier : IdentifierName but not ReservedWord

12.1.1 Static Semantics: Early Errors

BindingIdentifier : Identifier

• It is a Syntax Error if this production is contained in strict code and the StringValue of Identifier is "arguments" or "eval".

IdentifierReference[Yield] : yield
BindingIdentifier[Default, Yield] : yield
LabelIdentifier[Yield] : yield

• It is a Syntax Error if this production has a [Yield] parameter.
• It is a Syntax Error if this production is contained in strict code.
• It is a Syntax Error if this production is within the FunctionBody of a GeneratorMethod, GeneratorDeclaration, or GeneratorExpression.

Identifier : IdentifierName but not ReservedWord

• It is a Syntax Error if this phrase is contained in strict code and the StringValue of IdentifierName is: "implements", "interface", "let", "package", "private", "protected", "public", or "static".

NOTE StringValue of IdentifierName normalizes any Unicode escape sequences in IdentifierName hence such escapes cannot be used to write an Identifier whose code point sequence is the same as a ReservedWord.

12.1.2 Static Semantics: BoundNames

See also: 13.2.1.2, 13.2.2.1, 0, 14.1.3, 14.2.2, 14.4.2, 14.5.2, 15.2.1.2, 15.2.2.1.
158

BindingIdentifier : Identifier
  1. Return a new List containing the StringValue of Identifier.

BindingIdentifier : yield
  1. Return a new List containing "yield".

BindingIdentifier : default
  1. Return a new List containing "default".

12.1.3 Static Semantics: StringValue

See also: 11.6.1.2, 11.8.4.2.

IdentifierReference : yield
BindingIdentifier : yield
LabelIdentifier : yield
  1. Return "yield".

BindingIdentifier : default
  1. Return "default".

Identifier : IdentifierName but not ReservedWord
  1. Return the StringValue of IdentifierName.

12.1.4 Runtime Semantics: BindingInitialization

With arguments value and environment.

See also: 13.2.2.4, 13.2.3.5, 13.6.4.5, 13.14.4.

NOTE undefined is passed for environment to indicate that a PutValue operation should be used to assign the initialization value. This is the case for var statements and formal parameter lists of some non-strict functions (See 9.2.13). In those cases a lexical binding is hoisted and preinitialized prior to evaluation of its initializer.

BindingIdentifier : Identifier
  1. Let name be StringValue of Identifier.
  2. Return InitializeBoundName(name, value, environment).

BindingIdentifier : default
  1. Return InitializeBoundName("default", value, environment).

BindingIdentifier : yield
  1. Return InitializeBoundName("yield", value, environment).
12.1.4.1 Runtime Semantics: `InitializeBoundName(name, value, environment)`

1. Assert: Type(name) is String.
2. If `environment` is not `undefined`, then
   a. Let `env` be the environment record component of `environment`.
   b. Call the InitializeBinding concrete method of `env` passing `name` and `value` as the arguments.
   c. Return NormalCompletion(`undefined`).
3. Else
   a. Let `lhs` be ResolveBinding(`name`).
   b. Return PutValue(`lhs`, `value`).

12.1.5 Runtime Semantics: Evaluation

`IdentifierReference : Identifier`

1. Return ResolveBinding(StringValue(Identifier)).

`IdentifierReference : yield`

1. Return ResolveBinding("yield").

**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 resolves the binding of `yield` as if it was an `Identifier`. 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.

12.2 Primary Expression

**Syntax**

PrimaryExpression$_{\gamma\delta\alpha\beta\theta} :$

```
this
IdentifierReference$_{\gamma\delta\alpha\beta\theta}$
Literal
ArrayInitializer$_{\gamma\delta\alpha\beta\theta}$
ObjectLiteral$_{\gamma\delta\alpha\beta\theta}$
FunctionExpression
ClassExpression
GeneratorExpression
RegularExpressionLiteral
TemplateLiteral$_{\gamma\delta\alpha\beta\theta}$
CoverParenthesizedExpressionAndArrowParameterList$_{\gamma\delta\alpha\beta\theta}$
```

`CoverParenthesizedExpressionAndArrowParameterList$_{\gamma\delta\alpha\beta\theta} :$

```
( Expression$_{\gamma\delta\alpha\beta\theta}$ )
( )
( ... BindingIdentifier$_{\gamma\delta\alpha\beta\theta}$ )
( Expression$_{\gamma\delta\alpha\beta\theta}$ , ... BindingIdentifier$_{\gamma\delta\alpha\beta\theta}$ )
```

**Supplemental Syntax**

When processing the production
12.2.0 Semantics

12.2.0.1 Static Semantics: `CoveredParenthesizedExpression`

`CoverParenthesizedExpressionAndArrowParameterList[Yield] : { Expression, ?Yield }`

1. Return the result of parsing the lexical token stream matched by `CoverParenthesizedExpressionAndArrowParameterList[Yield]` using either `ParenthesizedExpression` or `ParenthesizedExpression[Yield]` as the goal symbol depending upon whether the `Yield` grammar parameter was present when `CoverParenthesizedExpressionAndArrowParameterList` was matched.

12.2.0.2 Static Semantics: `IsFunctionDefinition`

See also: 12.2.9.2, 12.3.1.2, 12.4.2, 12.5.2, 12.6.1, 12.7.1, 12.8.1, 12.9.1, 12.10.1, 12.11.1, 12.12.1, 12.13.1, 12.14.2, 12.15.1, 14.1.11, 14.4.8, 14.5.8.

PrimaryExpression:

`this`
- IdentifierReference
- Literal
- ArrayInitializer
- ObjectLiteral
- RegularExpressionLiteral
- TemplateLiteral

1. Return false.

PrimaryExpression : `CoveredParenthesizedExpressionAndArrowParameterList`

1. Let `expr` be `CoveredParenthesizedExpression` of `CoverParenthesizedExpressionAndArrowParameterList`
2. Return `IsFunctionDefinition` of `expr`.

12.2.0.3 Static Semantics: `IsIdentifierRef`

See also: 12.3.1.3.

PrimaryExpression:

`IdentifierReference`

1. Return true.
PrimaryExpression:
this
Literal
ArrayInitializer
ObjectLiteral
FunctionExpression
ClassExpression
GeneratorExpression
RegularExpressionLiteral
TemplateLiteral
CoverParenthesizedExpressionAndArrowParameterList

1. Return false.

12.2.0.4 Static Semantics: IsValidSimpleAssignmentTarget


PrimaryExpression:
this
Literal
ArrayInitializer
ObjectLiteral
FunctionExpression
ClassExpression
GeneratorExpression
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: CoverParenthesizedExpressionAndArrowParameterList

1. Let expr be CoverParenthesizedExpression of CoverParenthesizedExpressionAndArrowParameterList.
2. Return IsValidSimpleAssignmentTarget of expr.

12.2.1 The this Keyword

12.2.1.1 Runtime Semantics: Evaluation

PrimaryExpression: this

1. Return ResolveThisBinding().
12.2.2 Identifier Reference

See 12.1 for IdentifierReference.

12.2.3 Literals

Syntax

Literal: NullLiteral
(BooleanLiteral
(NumericLiteral
(StringLiteral

12.2.3.1 Runtime Semantics: Evaluation

Literal : NullLiteral
1. Return null.

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

Literal : NumericLiteral
1. Return the number whose value is MV of NumericLiteral as defined in 11.8.3.

Literal : StringLiteral
1. Return the StringValue of StringLiteral as defined in 11.8.4.2.

12.2.4 Array Initializer

Syntax

ArrayInitializer: ArrayLiteral

12.2.4.1 Array Literal

NOTE: An ArrayLiteral is an expression describing the initialization 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 initializer 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(\[\]\) : 
  [ Elisionopt ]
  [ ElementList(\[\]\) ]
  [ ElementList(\[\]\), Elisionopt ]

ElementList(\[\]\) : 
  Elisionopt AssignmentExpression(\[\]\),? Elisionopt
  Elisionopt SpreadElement(\[\]\)
  ElementList(\[\]\), Elisionopt AssignmentExpression(\[\]\),? Elisionopt
  ElementList(\[\]\), Elisionopt SpreadElement(\[\]\))

Elision : ,
  Elision ,

SpreadElement(\[\]\) : 
  . . . AssignmentExpression(\[\]\),? Elisionopt

12.2.4.1.1 Static Semantics: ElisionWidth

Elision : ,
  1. Return the numeric value 1.

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

12.2.4.1.2 Runtime Semantics: Array Accumulation

With parameters array and nextIndex.

ElementList \* Elisionopt 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 initialValue be GetValue(initResult).
  4. ReturnIfAbrupt(initialValue).
  5. Let created be the result of calling the [[DefineOwnProperty]] internal method of array with arguments ToString(ToUint32(nextIndex+padding)) and the PropertyDescriptor{ [[Value]]: initialValue, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}.
  6. Assert: created is true.
  7. Return nextIndex+padding+1.

ElementList \* Elisionopt 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 , Elisionopt AssignmentExpression

1. Let postIndex be the result of performing ArrayAccumulation for ElementList with arguments array and nextIndex.
2. ReturnIfAbrupt(postIndex).
3. Let padding be the ElisionWidth 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 , Elisionopt SpreadElement

1. Let postIndex be the result of performing ArrayAccumulation for ElementList with arguments array and nextIndex.
2. ReturnIfAbrupt(postIndex).
3. Let padding be the ElisionWidth of Elision; if Elision is not present, use the numeric value zero.
4. Return the result of performing ArrayAccumulation for SpreadElement with arguments array and postIndex+padding.

SpreadElement : ... AssignmentExpression

1. Let spreadRef be the result of evaluating AssignmentExpression.
2. Let spreadObj be ToObject(GetValue(spreadRef)).
3. Let iterator be GetIterator(spreadObj).
4. ReturnIfAbrupt(iterator).
5. Repeat
   a. Let next be 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 CreateDataPropertyOrThrow(array, ToString(ToUint32(nextIndex)), nextValue).
   g. ReturnIfAbrupt(defineStatus).
   h. Let nextIndex be 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]].

12.2.4.1.3 Runtime Semantics: Evaluation

ArrayLiteral : [ Elisionopt ]

1. Let array be ArrayCreate(0).
2. Let pad be the ElisionWidth of Elision; if Elision is not present, use the numeric value zero.
4. NOTE: the above Put can not fail because of the nature of the object returned by ArrayCreate.
5. Return array.
ArrayLiteral: [ ElementList ]
1. Let array be ArrayCreate(0).
2. Let len be the result of performing ArrayAccumulation for ElementList with arguments array and 0.
3. ReturnIfAbrupt(len).
5. NOTE: the above Put can not fail because of the nature of the object returned by ArrayCreate.
6. Return array.

ArrayLiteral: [ ElementList , Elisionopt ]
1. Let array be ArrayCreate(0).
2. Let len be the result of performing ArrayAccumulation for ElementList with arguments array and 0.
3. ReturnIfAbrupt(len).
4. Let padding be the ElisionWidth of Elision; if Elision is not present, use the numeric value zero.
5. Perform Put(array, "length", ToUint32(padding+len), false).
6. NOTE: the above Put can not fail because of the nature of the object returned by ArrayCreate.
7. Return array.

12.2.5 Object Initializer

NOTE 1 An object initializer is an expression describing the initialization 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 initializer is evaluated.

Syntax
ObjectLiteralτvoid :
{ }
{ PropertyDefinitionList[τvoid] }
{ PropertyDefinitionList[τvoid] , }  
PropertyDefinitionList[τvoid] :
PropertyDefinition[τvoid]
PropertyDefinitionList[τvoid] , PropertyDefinition[τvoid]

PropertyDefinition[τvoid] :
IdentifierReference[τvoid]
CoverInitializedName[τvoid]
PropertyName[τvoid] : AssignmentExpression[τ, τvoid]
MethodDefinition[τvoid]

PropertyName[τvoid,GeneratorParameter] :
LiteralPropertyName
[+GeneratorParameter] ComputedPropertyName
[-GeneratorParameter] ComputedPropertyName[τvoid]

LiteralPropertyName :
IdentifierName
StringLiteral
NumericLiteral
ComputedPropertyName: 
  [ AssignmentExpression, Yield ]

CoverInitializedName: 
  IdentifierReference, Initializer

Initializer: 
  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 CoverInitializedName 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.

12.2.5.1 Static Semantics: Early Errors

In addition to describing an actual object initializer the ObjectLiteral productions are also used as a cover grammar for ObjectAssignmentPattern (12.14.5), and may be recognized as part of a CoverParenthesizedExpressionAndArrowParameterList. When ObjectLiteral appears in a context where ObjectAssignmentPattern is required the following Early Error rules are not applied. In addition, they are not applied when initially parsing a CoverParenthesizedExpressionAndArrowParameterList.

PropertyDefinition: CoverInitializedName

  - 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 (12.14.5). It cannot occur in an actual object initializer.

12.2.5.2 Static Semantics: ComputedPropertyContains

With parameter symbol.

See also: 14.3.2, 14.4.3, 14.5.5.

PropertyName: LiteralPropertyName

  1. Return false.

PropertyName: ComputedPropertyName

  1. Return the result of ComputedPropertyName Contains symbol.

12.2.5.3 Static Semantics: Contains

With parameter symbol.

See also: 5.3, 12.3.1.1, 14.1.4, 14.2.3, 14.4.3, 14.5.4

PropertyDefinition: MethodDefinition

  1. If symbol is MethodDefinition, return true.
  2. Return the result of ComputedPropertyContains for MethodDefinition with argument symbol.
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.

12.2.5.4  Static Semantics: HasComputedPropertyKey

See also: 14.3.4, 14.4.5

PropertyDefinitionList : PropertyDefinitionList , PropertyDefinition
1. If HasComputedPropertyKey of PropertyDefinitionList is true, then return true.
2. Return HasComputedPropertyKey of PropertyDefinition.

PropertyDefinition : IdentifierReference
1. Return false.

PropertyDefinition : PropertyName : AssignmentExpression
1. Return IsComputedPropertyKey of PropertyName.

12.2.5.5  Static Semantics: IsComputedPropertyKey

PropertyName : LiteralPropertyName
1. Return false.

PropertyName : ComputedPropertyName
1. Return true.

12.2.5.6  Static Semantics: PropName

See also: 14.3.5, 14.4.9, 14.5.12

PropertyDefinition : IdentifierReference
1. Return StringValue of IdentifierReference.

PropertyDefinition : PropertyName : AssignmentExpression
1. Return PropName of PropertyName.

LiteralPropertyName : IdentifierName
1. Return StringValue of IdentifierName.

LiteralPropertyName : StringLiteral
1. Return a String value whose code units 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.

12.2.5.7 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.

12.2.5.8 Runtime Semantics: Evaluation

ObjectLiteral : { }
  1. Return ObjectCreate(%ObjectPrototype%).

ObjectLiteral :
  { PropertyDefinitionList
    { PropertyDefinitionList , } }
  1. Let obj be ObjectCreate(%ObjectPrototype%).
  2. Let status be the result of performing PropertyDefinitionEvaluation of PropertyDefinitionList with argument obj.
  3. ReturnIfAbrupt(status).
  4. Return obj.

LiteralPropertyName : IdentifierName
  1. Return StringValue of IdentifierName.

LiteralPropertyName : StringLiteral
  1. Return a String value whose code units 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 exprValue be the result of evaluating AssignmentExpression.
  2. Let propName be GetValue(exprValue).
3. ReturnIfAbrupt(propName).
4. Return ToPropertyKey(propName).

12.2.5.9 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. Return the result of performing PropertyDefinitionEvaluation of PropertyDefinition with argument object.

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. Let desc be the Property Descriptor({[[Value]]: propValue, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}.
7. Return DefinePropertyOrThrow(object, propName, desc).

PropertyDefinition : PropertyName : AssignmentExpression
1. Let propName be the result of evaluating PropertyName.
2. ReturnIfAbrupt(propName).
3. Let propKey be the result of evaluating AssignmentExpression.
4. Let propValue be GetValue(exprValueRef).
5. ReturnIfAbrupt(propValue).
6. If IsFunctionDefinition of AssignmentExpression is true, then
   a. Assert: propValue is an ECMAScript function object.
   b. Let referencesSuper be the value of propValue’s [[NeedsSuper]] internal slot.
   c. Let thisMode be the value of propValue’s [[ThisMode]] internal slot.
   d. If thisMode is not lexical and referencesSuper is true, then
      i. If propValue’s [[HomeObject]] internal slot is undefined, then
         1. Assert: AssignmentExpression is not a class definition whose constructor references super.
      2. Set propValue’s [[HomeObject]] internal slot to object.
      3. Set propValue’s [[MethodName]] internal slot to propKey.
   e. If IsAnonymousFunctionDefinition(AssignmentExpression) is true, then
      i. SetFunctionName(propValue, propKey).
      ii. Assert: SetFunctionName will not return an abrupt completion.
7. Let desc be the Property Descriptor({[[Value]]: propValue, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}.
8. Return DefinePropertyOrThrow(object, propKey, desc).

NOTE An alternative semantics for this production is given in B.3.1.
12.2.6 Function Defining Expressions

See 14.1 for PrimaryExpression : FunctionExpression.
See 14.4 for PrimaryExpression : GeneratorExpression.
See 14.5 for PrimaryExpression : ClassExpression.

12.2.7 Regular Expression Literals

Syntax
See 11.8.4.

12.2.7.1 Static Semantics: Early Errors

PrimaryExpression : RegularExpressionLiteral

- It is a Syntax Error if BodyText of RegularExpressionLiteral cannot be recognized using the goal symbol Pattern of the ECMAScript RegExp grammar specified in 21.2.1.
- It is a Syntax Error if FlagText of RegularExpressionLiteral contains any code points other than “g”, “i”, “m”, “u”, or “y”, or if it contains the same code point more than once.

12.2.7.2 Runtime Semantics: Evaluation

PrimaryExpression : RegularExpressionLiteral

1. Let pattern be the string value consisting of the UTF-16Encoding of each code point of BodyText of RegularExpressionLiteral.
2. Let flags be the string value consisting of the UTF-16Encoding of each code point of FlagText of RegularExpressionLiteral.
3. Return RegExpCreate(pattern, flags).

12.2.8 Template Literals

Syntax

TemplateLiteralYield : NoSubstitutionTemplate TemplateHead Expression[In, ?, Yield] [Lexical goal InputElementTemplateTail] TemplateSpans[In, ?, Yield]

TemplateSpansYield : TemplateTail
TemplateMiddleListYield : TemplateMiddle Expression[In, ?, Yield] [Lexical goal InputElementTemplateTail] TemplateTail
TemplateMiddleListYieldYield : TemplateMiddleExpression[In, ?, Yield] [Lexical goal InputElementTemplateTail] TemplateMiddleExpression[In, ?, Yield]
12.2.8.1 Static Semantics

12.2.8.1.1 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
   a. Let last be the TV of TemplateMiddle.
3. Else,
   a. Let last be the TRV of TemplateMiddle.
4. Append \textit{last} as the last element of the List \textit{front}.
5. Return \textit{front}.

12.2.8.2 Runtime Semantics

12.2.8.2.1 Runtime Semantics: \texttt{ArgumentListEvaluation}

See also: 12.3.6.1

\texttt{TemplateLiteral : NoSubstitutionTemplate}

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

\texttt{TemplateLiteral : TemplateHead Expression TemplateSpans}

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

12.2.8.2.2 Runtime Semantics: \texttt{GetTemplateCallSite}

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

1. If a call site object for the source code corresponding to \texttt{templateLiteral} has already been created (see step 12 below) by a previous call to this abstract operation, then
   a. Return that call site object.
2. Let \textit{cookedStrings} be \texttt{TemplateStrings} of \texttt{templateLiteral} with argument \textit{false}.
3. Let \textit{rawStrings} be \texttt{TemplateStrings} of \texttt{templateLiteral} with argument \textit{true}.
4. Let \textit{count} be the number of elements in the List \textit{cookedStrings}.
5. Let \textit{siteObj} be \texttt{ArrayCreate}(\textit{count}).
6. Let \textit{rawObj} be \texttt{ArrayCreate}(\textit{count}).
7. Let \textit{index} be 0.
8. Repeat while \textit{index} \textlt; \textit{count}
   a. Let \textit{prop} be \texttt{ToString}(\textit{index}).
   b. Let \textit{cookedValue} be the string value at 0-based position \textit{index} of the List \textit{cookedStrings}.
   c. Call the \texttt{[[DefineOwnProperty]]} internal method of \textit{siteObj} with arguments \textit{prop} and \texttt{PropertyDescriptor\{[[Value]]: \textit{cookedValue}, [[Enumerable]]: \textit{true}, [[Writable]]: \textit{false}, [[Configurable]]: \textit{false}\}}.
   d. Let \textit{rawValue} be the string value at 0-based position \textit{index} of the List \textit{rawStrings}.
   e. Call the \texttt{[[DefineOwnProperty]]} internal method of \textit{rawObj} with arguments \textit{prop} and \texttt{PropertyDescriptor\{[[Value]]: \textit{rawValue}, [[Enumerable]]: \textit{true}, [[Writable]]: \textit{false}, [[Configurable]]: \textit{false}\}}.
   f. Let \textit{index} be \textit{index}+1.
9. Perform SetIntegrityLevel(\textit{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").
12. Remember an association between the source code corresponding to templateLiteral and siteObj such that siteObj can be retrieved in subsequent calls to this abstract operation.
13. Return siteObj.

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

NOTE 2  Each TemplateLiteral in the program code is associated with a unique Template call site object that is used in the evaluation of tagged Templates (12.2.8.2.4). The call site objects are frozen and 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.

NOTE 3  Future editions of this specification may define additional non-enumerable properties of call site objects.

12.2.8.2.3 Runtime Semantics: SubstitutionEvaluation

TemplateSpans : TemplateTail
   1. Return an empty List.

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

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

TemplateMiddleList : TemplateMiddleList TemplateMiddle Expression
   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.

12.2.8.2.4 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 \( \text{tail} \) be the result of evaluating TemplateSpans.
6. ReturnIfAbrupt(\( \text{tail} \)).
7. Return the string value whose elements are the code units of \( \text{head} \) followed by the elements of \( \text{middle} \) followed by the elements of \( \text{tail} \).

NOTE The string conversion semantics applied to the Expression value are like `String.prototype.concat` rather than the `+` operator.

TemplateSpans : TemplateTail
1. Let \( \text{tail} \) be the TV of TemplateTail as defined in 11.8.6.
2. Return the string whose elements are the code units of \( \text{tail} \).

TemplateSpans : TemplateMiddleList TemplateTail
1. Let \( \text{head} \) be the result of evaluating TemplateMiddleList.
2. ReturnIfAbrupt(\( \text{head} \)).
3. Let \( \text{tail} \) be the TV of TemplateTail as defined in 11.8.6.
4. Return the string whose elements are the elements of \( \text{head} \) followed by the elements of \( \text{tail} \).

TemplateMiddleList : TemplateMiddle Expression
1. Let \( \text{head} \) be the TV of TemplateMiddle as defined in 11.8.6.
2. Let \( \text{sub} \) be the result of evaluating Expression.
3. Let \( \text{middle} \) be ToString(\( \text{sub} \)).
4. ReturnIfAbrupt(\( \text{middle} \)).
5. Return the sequence of code units consisting of the code units of \( \text{head} \) followed by the elements of \( \text{middle} \).

NOTE The string conversion semantics applied to the Expression value are like `String.prototype.concat` rather than the `+` operator.

TemplateMiddleList : TemplateMiddleList TemplateMiddle Expression
1. Let \( \text{rest} \) be the result of evaluating TemplateMiddleList.
2. ReturnIfAbrupt(\( \text{rest} \)).
3. Let \( \text{middle} \) be the TV of TemplateMiddle as defined in 11.8.6.
4. Let \( \text{sub} \) be the result of evaluating Expression.
5. Let \( \text{last} \) be ToString(\( \text{sub} \)).
6. ReturnIfAbrupt(\( \text{last} \)).
7. Return the sequence of code units consisting of the elements of \( \text{rest} \) followed by the code units of \( \text{middle} \) followed by the elements of \( \text{last} \).

NOTE The string conversion semantics applied to the Expression value are like `String.prototype.concat` rather than the `+` operator.

12.2.9 The Grouping Operator

12.2.9.1 Static Semantics: Early Errors

PrimaryExpression : CoverParenthesizedExpressionAndArrowParameterList
- It is a Syntax Error if the lexical token sequence matched by CoverParenthesizedExpressionAndArrowParameterList cannot be parsed with no tokens left over using ParenthesizedExpression as the goal symbol.
12.2.9.2 Static Semantics: `IsFunctionDefinition`

See also: 12.2.0.2, 12.3.1.2, 12.4.2, 12.5.2, 12.6.1, 12.7.1, 12.8.1, 12.9.1, 12.10.1, 12.11.1, 12.12.1, 12.13.1, 12.14.2, 12.15.1, 14.1.11, 14.4.8, 14.5.8.

`ParenthesizedExpression` : ( `Expression` )

1. Return `IsFunctionDefinition` of `Expression`.

12.2.9.3 Static Semantics: `IsValidSimpleAssignmentTarget`


`ParenthesizedExpression` : ( `Expression` )

1. Return `IsValidSimpleAssignmentTarget` of `Expression`.

12.2.9.4 Runtime Semantics: Evaluation

`PrimaryExpression` : `CoverParenthesizedExpressionAndArrowParameterList`

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

`ParenthesizedExpression` : ( `Expression` )

1. Return the result of evaluating `Expression`. This may be of type `Reference`.

NOTE This algorithm does not apply `GetValue` to the result of evaluating `Expression`. The principal motivation for this is so that operators such as `delete` and `typeof` may be applied to parenthesized expressions.

12.3 Left-Hand-Side Expressions

Syntax

`MemberExpression`:  
- `[lexical goal InputElementRegExp ] PrimaryExpression[ γvad ]`
- `MemberExpression[ γvad ] [ Expression[ γvad ] ]`
- `MemberExpression[ γvad ] . IdentifierName`
- `MemberExpression[ γvad ] TemplateLiteral[ γvad ]`
- `super [ Expression[ γvad ] ]`
- `super . IdentifierName`
- `new super Arguments[ γvad ]`
- `new MemberExpression[ γvad ] . Arguments[ γvad ]`
NewExpression : 
  MemberExpression

  new NewExpression

  new super

CallExpression : 
  MemberExpression Arguments

  super Arguments

  CallExpression [ Expression, ... ]

  CallExpression , IdentifierName

  CallExpression TemplateLiteral

Arguments : ()

  ( ArgumentList )

ArgumentList : AssignmentExpression

  . . . AssignmentExpression

  ArgumentList , AssignmentExpression

  ArgumentList , . . . AssignmentExpression

LeftHandSideExpression :

  NewExpression

  CallExpression

12.3.1 Static Semantics

12.3.1.1 Static Semantics: Contains

With parameter symbol.

See also: 5.3, 12.2.5.2, 14.1.4, 14.2.3, 14.4.3, 14.5.4

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.

12.3.1.2 Static Semantics: IsFunctionDefinition

See also: 12.2.0.2, 12.2.9.2, 12.4.2, 12.5.2, 12.6.1, 12.7.1, 12.8.1, 12.9.1, 12.10.1, 12.11.1, 12.12.1,

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

NewExpression :
    new NewExpression
    new super

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

1. Return false.

12.3.1.3 Static Semantics: IsIdentifierRef

See also: 12.2.0.3.

LeftHandSideExpression :
    CallExpression

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

1. Return false.

12.3.1.4 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
    new super

MemberExpression:
    MemberExpression TemplateLiteral
    new super Arguments
    new MemberExpression Arguments

1. Return false.

12.3.2 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

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 the result of evaluating StringValue of IdentifierName.

12.3.2.1 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 RequireObjectCoercible(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.

MemberExpression : MemberExpression . IdentifierName

1. Let baseReference be the result of evaluating MemberExpression.
2. Let baseValue be GetValue(baseReference).
3. ReturnIfAbrupt(baseValue).
4. Let bv be RequireObjectCoercible(baseValue).
5. ReturnIfAbrupt(bv).
6. Let propertyNameString be StringValue of IdentifierName.
7. If the code matched by the syntactic production that is being evaluated is strict mode code, let strict be true, else let strict be false.
8. Return a value of type Reference whose base value is bv, and whose referenced name is propertyNameString, and whose strict reference flag is strict.

CallExpression : CallExpression [ Expression ]

Is evaluated in exactly the same manner as MemberExpression : MemberExpression [ Expression ] except that the contained CallExpression is evaluated in step 1.

CallExpression : CallExpression . IdentifierName

Is evaluated in exactly the same manner as MemberExpression : MemberExpression . IdentifierName except that the contained CallExpression is evaluated in step 1.
12.3.3 The new Operator

12.3.3.1 Runtime Semantics: Evaluation

NewExpression : `new` NewExpression

1. Let `thisNewExpression` be this `NewExpression`.
2. Return `EvaluateNew(thisNewExpression, NewExpression, empty)`.

MemberExpression : `new` MemberExpression Arguments

1. Let `thisMemberExpression` be this `MemberExpression`.
2. Return `EvaluateNew(thisMemberExpression, MemberExpression, Arguments)`.

12.3.3.1.1 Runtime Semantics: EvaluateNew(thisCall, constructProduction, arguments)

The abstract operation `EvaluateNew` with arguments `production` and `arguments` performs the following steps:

1. Assert: `thisCall` is either a `NewExpression` or a `MemberExpression`.
2. Assert: `constructProduction` is either a `NewExpression` or a `MemberExpression`.
3. Assert: `arguments` is either `empty` or an `Arguments` production.
4. Let `ref` be the result of evaluating `constructProduction`.
5. Let `constructor` be `GetValue(ref)`.
6. ReturnIfAbrupt(`constructor`).
7. If `arguments` is `empty`, then let `argList` be an empty List.
8. Else,
   a. Let `argList` be the result of evaluating `Arguments`, producing a List of argument values (12.3.6).
   b. ReturnIfAbrupt(`argList`).
9. If `IsConstructor`(`constructor`) is `false`, throw a `TypeError` exception.
10. Let `tailCall` be `IsInTailPosition(thisCall)`. (See 14.6.1)
11. If `tailCall` is `true`, then perform the `PrepareForTailCall` abstract operation.
12. Let `result` be the result of calling the `[[Construct]]` internal method on `constructor`, passing `argList` as the argument.
13. Assert: If `tailCall` is `true`, the above call of `[[Construct]]` will not return here, but instead evaluation will continue as if the following return has already occurred.
14. Return `result`.

12.3.4 Function Calls

12.3.4.1 Runtime Semantics: Evaluation

CallExpression : MemberExpression Arguments

1. Let `ref` be the result of evaluating `MemberExpression`,
2. If `MemberExpression` consists solely of the IdentifierName `eval`, then
   a. [check if direct eval]
   b. Return `EvaluateCall(ref, Arguments, false)`.
3. Let `thisCall` be this `CallExpression`.
4. Let `tailCall` be `IsInTailPosition(thisCall)`. (See 14.6.1)
5. Return `EvaluateCall(ref, Arguments, tailCall)`.
CallExpression : CallExpression Arguments

1. Let ref be the result of evaluating CallExpression.
2. Let thisCall be this CallExpression.
3. Let tailCall be IsInTailPosition(thisCall). (See 14.6.1)
4. Return EvaluateCall(ref, Arguments, tailCall).

12.3.4.2 Runtime Semantics: EvaluateCall

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

1. Let func be GetValue(ref).
2. ReturnIfAbrupt(func).
3. Let argList be ArgumentListEvaluation(arguments).
4. ReturnIfAbrupt(argList).
5. If IsCallable(func) is false, throw a TypeError exception.
6. 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).
7. Else Type(ref) is not Reference,
   a. Let thisValue be undefined.
8. If tailPosition is true, then perform the PrepareForTailCall abstract operation.
9. Let result be the result of calling the [[Call]] internal method on func, passing thisValue as the thisArgument and argList as the argumentsList.
10. Assert: If tailPosition is true, the above call will not return here, but instead evaluation will continue as if the following return has already occurred.
11. Assert: If result is not an abrupt completion then Type(result) is an ECMAScript language type
12. Return result.

12.3.5 The super Keyword

12.3.5.1 Static Semantics: Early Errors

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

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.

12.3.5.2 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 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 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 MakeSuperReference(`undefined`, `strict`).
3. Let `constructor` be GetValue(`ref`).
4. ReturnIfAbrupt(`constructor`).
5. Let `argList` be the result of evaluating `Arguments`, producing a List of argument values (12.3.6).
6. ReturnIfAbrupt(`argList`).
7. If IsConstructor(`constructor`) is `false`, throw a `TypeError` exception.
8. Let `thisCall` be this `MemberExpression`.
9. Let `tailCall` be IsInTailPosition(`thisCall`). (See 14.6.1)
10. If `tailCall` is `true`, then perform the PrepareForTailCall abstract operation.
11. Let `result` be the result of calling the `[[Construct]]` internal method on `constructor`, passing `argList` as the argument.
12. Assert: If `tailCall` is `true`, the above call of `[[Construct]]` will not return here, but instead evaluation will continue as if the following return has already occurred.
13. Return `result`.

**NewExpression**: `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 MakeSuperReference(`undefined`, `strict`).
3. Let `constructor` be GetValue(`ref`).
4. ReturnIfAbrupt(`constructor`).
5. Let `argList` be a new empty List.
6. ReturnIfAbrupt(`argList`).
7. If IsConstructor(`constructor`) is `false`, throw a `TypeError` exception.
8. Let `thisCall` be this `NewExpression`.
9. Let `tailCall` be IsInTailPosition(`thisCall`). (See 14.6.1)
10. If `tailCall` is `true`, then perform the PrepareForTailCall abstract operation.
11. Let `result` be the result of calling the `[[Construct]]` internal method on `constructor`, passing `argList` as the argument.
12. Assert: If `tailCall` is `true`, the above call of `[[Construct]]` will not return here, but instead evaluation will continue as if the following return has already occurred.
13. 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 MakeSuperReference(`undefined`, `strict`).
3. ReturnIfAbrupt(`ref`).
4. Let `thisCall` be this `CallExpression`.
5. Let `tailCall` be IsInTailPosition(`thisCall`). (See 14.6.1)
6. Return EvaluateCall(`ref`, `Arguments`, `tailCall`).

12.3.5.3 Runtime Semantics: MakeSuperReference(`propertyKey`, `strict`)

The abstract operation MakeSuperReference with arguments `propertyKey` and `strict` performs the following steps:

1. Let `env` be GetThisEnvironment( ).
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 RequireObjectCoercible(`baseValue`).
6. ReturnIfAbrupt(`bv`).
7. If `propertyKey` is `undefined`, then
   a. Let `propertyKey` be the result of calling the GetMethodName concrete method of `env`.
   b. If `propertyKey` is `undefined`, then throw a `ReferenceError` exception.
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`.

12.3.6 Argument Lists

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

12.3.6.1 Runtime Semantics: ArgumentListEvaluation

See also: 12.2.8.2.1

**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 ToObject(GetValue(`spreadRef`)).
4. Let `iterator` be GetIterator(`spreadObj`).
5. ReturnIfAbrupt(iterator).
6. Repeat
   a. Let next be 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. Append arg to the end of precedingArgs.
7. Return precedingArgs.

**ArgumentList** : ArgumentList, ... AssignmentExpression
1. Let precedingArgs be the result of evaluating ArgumentList.
2. Let spreadRef be the result of evaluating AssignmentExpression.
3. Let iterator be GetIterator(ToObject(GetValue(spreadRef))).
4. ReturnIfAbrupt(iterator).
5. Repeat
   a. Let next be 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.

### 12.3.7 Tagged Templates

**NOTE** A tagged template is a function call where the arguments of the call are derived from a `TemplateLiteral` (12.2.8). The actual arguments include a call site object (12.2.8.2.2) and the values produced by evaluating the expressions embedded within the `TemplateLiteral`.

#### 12.3.7.1 Runtime Semantics: Evaluation

**MemberExpression** : MemberExpression TemplateLiteral
1. Let tagRef be the result of evaluating MemberExpression.
2. Let thisCall be thisMemberExpression.
3. Let tailCall be IsInTailPosition(thisCall). (See 14.6.1)
4. Return EvaluateCall(tagRef, TemplateLiteral, tailCall).

**CallExpression** : CallExpression TemplateLiteral
1. Let tagRef be the result of evaluating CallExpression.
2. Let thisCall be thisCallExpression.
3. Let tailCall be IsInTailPosition(thisCall). (See 14.6.1)
4. Return EvaluateCall(tagRef, TemplateLiteral, tailCall).
12.4 Postfix Expressions

Syntax
PostfixExpression\[Yield\] :
  LeftHandSideExpression[?Yield] [no LineTerminator here] ++
  LeftHandSideExpression[?Yield] [no LineTerminator here] --

12.4.1 Static Semantics: Early Errors

PostfixExpression :
  LeftHandSideExpression  ++
  LeftHandSideExpression  --

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

12.4.2 Static Semantics: IsFunctionDefinition

See also: 12.2.0.2, 12.2.9.2, 12.3.1.2, 12.5.2, 12.6.1, 12.7.1, 12.8.1, 12.9.1, 12.10.1, 12.11.1, 12.12.1, 12.13.1, 12.14.2, 12.15.1, 14.1.11, 14.4.8, 14.5.8

PostfixExpression :
  LeftHandSideExpression  ++
  LeftHandSideExpression  --

  1. Return false.

12.4.3 Static Semantics: IsValidSimpleAssignmentTarget


PostfixExpression :
  LeftHandSideExpression  ++
  LeftHandSideExpression  --

  1. Return false.

12.4.4 Postfix Increment Operator

12.4.4.1 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 12.7.5).
  5. Let status be PutValue(lhs, newValue).
  6. ReturnIfAbrupt(status).
  7. Return oldValue.
12.4.5 Postfix Decrement Operator

12.4.5.1 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 subtracting the value 1 from oldValue, using the same rules as for the
   - operator (12.7.5).
5. Let status be PutValue(lhs, newValue).
6. ReturnIfAbrupt(status).
7. Return oldValue.

12.5 Unary Operators

Syntax

UnaryExpressionγ∀a,γ0,γ0,γ0,γ0,γ0 :
   PostfixExpressionγ∀a,γ0,γ0,γ0,γ0,γ0
   delete UnaryExpressionγ∀a,γ0,γ0,γ0,γ0,γ0
   void UnaryExpressionγ∀a,γ0,γ0,γ0,γ0,γ0
   typeof UnaryExpressionγ∀a,γ0,γ0,γ0,γ0,γ0
   ++ UnaryExpressionγ∀a,γ0,γ0,γ0,γ0,γ0
   -- UnaryExpressionγ∀a,γ0,γ0,γ0,γ0,γ0
   + UnaryExpressionγ∀a,γ0,γ0,γ0,γ0,γ0
   = UnaryExpressionγ∀a,γ0,γ0,γ0,γ0,γ0
   UnaryExpressionγ∀a,γ0,γ0,γ0,γ0,γ0
   ! UnaryExpressionγ∀a,γ0,γ0,γ0,γ0,γ0

12.5.1 Static Semantics: Early Errors

UnaryExpression γ :
   ++ UnaryExpression
   -- UnaryExpression

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

12.5.2 Static Semantics: IsFunctionDefinition

See also: 12.2.0.2, 12.2.9.2, 12.3.1.2, 12.4.2, 12.6.1, 12.7.1, 12.8.1, 12.9.1, 12.10.1, 12.11.1, 12.12.1,
UnaryExpression :
  delete UnaryExpression
  void UnaryExpression
  typeof UnaryExpression
  ++ UnaryExpression
  -- UnaryExpression
  + UnaryExpression
  - UnaryExpression
  ~ UnaryExpression
  ! UnaryExpression

1. Return false.

12.5.3 Static Semantics: IsValidSimpleAssignmentTarget


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

1. Return false.

12.5.4 The delete Operator

12.5.4.1 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 : CoverParenthesizedExpressionAndArrowParameterList
  and CoverParenthesizedExpressionAndArrowParameterList ultimately derives a phrase 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.

12.5.4.2 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. Assert: IsStrictReference(ref) is false.
   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.

12.5.5 The void Operator

12.5.5.1 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.

12.5.6 The typeof Operator

12.5.6.1 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".
  3. Let val be GetValue(val).
  4. ReturnIfAbrupt(val).
  5. Return a String according to Table 34.
### Table 34 — typeof Operator Results

<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 (implements [[Call]])</td>
<td>&quot;function&quot;</td>
</tr>
<tr>
<td>Object (non-standard exotic and does not implement [[Call]])</td>
<td>Implementation-defined. Must 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.

#### 12.5.7 Prefix Increment Operator

##### 12.5.7.1 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 12.7.5).
5. Let `status` be `PutValue(expr, newValue)`.
6. ReturnIfAbrupt(`status`).
7. Return `newValue`.

#### 12.5.8 Prefix Decrement Operator

##### 12.5.8.1 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 12.7.5).
5. Let `status` be `PutValue(expr, newValue)`.
6. ReturnIfAbrupt(status).
7. Return newValue.

12.5.9 Unary + Operator

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

12.5.9.1 Runtime Semantics: Evaluation

UnaryExpression : + UnaryExpression

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

12.5.10 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.

12.5.10.1 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.

12.5.11 Bitwise NOT Operator (~)

12.5.11.1 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.

12.5.12 Logical NOT Operator (!)

12.5.12.1 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.
12.6 Multiplicative Operators

Syntax

\[
\text{MultiplicativeExpression} @: \\
\text{UnaryExpression} \text{ UnaryExpression} \text{ UnaryExpression} \text{ UnaryExpression} \text{ UnaryExpression} \text{ UnaryExpression} \text{ UnaryExpression} \text{ UnaryExpression}
\]

12.6.1 Static Semantics: IsFunctionDefinition

See also: 12.2.0.2, 12.2.9.2, 12.4.2, 12.5.2, 12.7.1, 12.8.1, 12.9.1, 12.10.1, 12.11.1, 12.12.1, 12.13.1, 12.14.2, 12.15.1, 14.1.11, 14.4.8, 14.5.8.

12.6.2 Static Semantics: IsValidSimpleAssignmentTarget


12.6.3 Runtime Semantics: Evaluation

The production \( \text{MultiplicativeExpression} @: \text{UnaryExpression} \), where @ stands for one of the operators in the above definitions, is evaluated as follows:

1. Let \( \text{left} \) be the result of evaluating \( \text{MultiplicativeExpression} \).
2. Let \( \text{leftValue} \) be \( \text{Get} \text{Value}(\text{left}) \).
3. ReturnIfAbrupt(\( \text{leftValue} \)).
4. Let \( \text{right} \) be the result of evaluating \( \text{UnaryExpression} \).
5. Let \( \text{rightValue} \) be \( \text{Get} \text{Value}(\text{right}) \).
6. Let \( \text{lnum} \) be \( \text{ToNumber}(\text{leftValue}) \).
7. ReturnIfAbrupt(\( \text{lnum} \)).
8. Let \( \text{rnum} \) be \( \text{ToNumber}(\text{rightValue}) \).
9. ReturnIfAbrupt(\( \text{rnum} \)).
10. Return the result of applying the specified operation \( (\ast, \div, \% \) to \( \text{lnum} \) and \( \text{rnum} \). See the Notes below 12.6.3.1, 12.6.3.2, 12.6.3.3.
12.6.3.1 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.

12.6.3.2 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 a finite value by an infinity results in zero. 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.
12.6.3.3 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.

12.7 Additive Operators

Syntax

AdditiveExpression : 
  MultiplicativeExpression
AdditiveExpression + MultiplicativeExpression
AdditiveExpression - MultiplicativeExpression

12.7.1 Static Semantics: IsFunctionDefinition

See also: 12.2.0.2, 12.2.9.2, 12.3.1.2, 12.4.2, 12.5.2, 12.6.1, 12.8.1, 12.9.1, 12.10.1, 12.11.1, 12.12.1, 12.13.1, 12.14.2, 12.15.1, 14.1.11, 14.4.8, 14.5.8.

AdditiveExpression :
  AdditiveExpression + MultiplicativeExpression
  AdditiveExpression - MultiplicativeExpression

  1. Return false.
12.7.2 Static Semantics: IsValidSimpleAssignmentTarget


AdditiveExpression :
AdditiveExpression + MultiplicativeExpression
AdditiveExpression − MultiplicativeExpression

1. Return false.

12.7.3 The Addition operator ( + )

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

12.7.3.1 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. If Type(lprim) is Symbol or Type(rprim) is Symbol, then throw a TypeError exception.
   b. Return the String that is the result of concatenating ToString(lprim) followed by
      ToString(rprim)
12. Let lnum be ToNumber(lprim).
13. ReturnIfAbrupt(lnum).
14. Let rnum be ToNumber(rprim).
15. ReturnIfAbrupt(rnum).
16. Return the result of applying the addition operation to lnum and rnum. See the Note below 12.7.5.

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 (7.2.9), by using the logical
or operation instead of the logical and operation.

12.7.4 The Subtraction Operator ( − )

12.7.4.1 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. 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 12.7.5.

12.7.5 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 $a + (-b)$.

12.8 Bitwise Shift Operators

Syntax

```
ShiftExpression\[?\]Yield 
AdditiveExpression\[?\]Yield 
AdditiveExpression\[?\]Yield 
AdditiveExpression\[?\]Yield 
```

© Ecma International 2014
12.8.1 Static Semantics: IsFunctionDefinition

See also: 12.2.0.2, 12.2.9.2, 12.3.1.2, 12.4.2, 12.5.2, 12.6.1, 12.7.1, 12.9.1, 12.10.1, 12.11.1, 12.12.1, 12.13.1, 12.14.2, 12.15.1, 14.1.11, 14.4.8, 14.5.8.

ShiftExpression:
- ShiftExpression << AdditiveExpression
- ShiftExpression >> AdditiveExpression
- ShiftExpression >>> AdditiveExpression
1. Return false.

12.8.2 Static Semantics: IsValidSimpleAssignmentTarget


ShiftExpression:
- ShiftExpression << AdditiveExpression
- ShiftExpression >> AdditiveExpression
- ShiftExpression >>> AdditiveExpression
1. Return false.

12.8.3 The Left Shift Operator (<<)

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

12.8.3.1 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.

12.8.4 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.
12.8.4.1 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.

12.8.5 The Unsigned Right Shift Operator ( >>> )

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

12.8.5.1 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 zero-filling right shift of lnum by shiftCount bits. Vacated bits are filled with zero. The result is an unsigned 32-bit integer.

12.9 Relational Operators

NOTE The result of evaluating a relational operator is always of type Boolean, reflecting whether the relationship named by the operator holds between its two operands.
### Syntax

*RelationalExpression*: 

\[ \text{ShiftExpression} \ |
\text{RelationalExpression} < \text{ShiftExpression} |
\text{RelationalExpression} > \text{ShiftExpression} |
\text{RelationalExpression} <= \text{ShiftExpression} |
\text{RelationalExpression} >= \text{ShiftExpression} |
\text{RelationalExpression} \text{ instanceof} \text{ShiftExpression} |
\text{RelationalExpression} \text{ in} \text{ShiftExpression} \]

**NOTE** The `in` grammar parameter is needed to avoid confusing the `in` operator in a relational expression with the `in` operator in a for statement.

#### 12.9.1 Static Semantics: `IsFunctionDefinition`

See also: 12.2.0.2, 12.2.9.2, 12.3.1.2, 12.4.2, 12.5.2, 12.6.1, 12.7.1, 12.8.1, 12.10.1, 12.11.1, 12.12.1, 12.13.1, 12.14.2, 12.15.1, 14.1.11, 14.4.8, 14.5.8.

*RelationalExpression*:

- `RelationalExpression < ShiftExpression`
- `RelationalExpression > ShiftExpression`
- `RelationalExpression <= ShiftExpression`
- `RelationalExpression >= ShiftExpression`
- `RelationalExpression instanceof ShiftExpression`
- `RelationalExpression in ShiftExpression`

1. Return `false`.

#### 12.9.2 Static Semantics: `IsValidSimpleAssignmentTarget`


*RelationalExpression*:

- `RelationalExpression < ShiftExpression`
- `RelationalExpression > ShiftExpression`
- `RelationalExpression <= ShiftExpression`
- `RelationalExpression >= ShiftExpression`
- `RelationalExpression instanceof ShiftExpression`
- `RelationalExpression in ShiftExpression`

1. Return `false`.

#### 12.9.3 Runtime Semantics: `Evaluation`

*RelationalExpression*:

1. Let `lref` be the result of evaluating `RelationalExpression`.
2. Let `lval` be `GetValue(lref)`.
3. Return `IsAbrupt(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 true or undefined, return false. Otherwise, return true.

RelationalExpression : RelationalExpression instanceof ShiftExpression

1. Let lref be the result of evaluating RelationalExpression.
2. Let lval be GetValue(lref).
3. ReturnIfAbrupt(lval).
4. Let rref be the result of evaluating ShiftExpression.
5. Let rval be GetValue(rref).
6. ReturnIfAbrupt(rval).
7. Return InstanceofOperator(lval, rval).

RelationalExpression : RelationalExpression in ShiftExpression

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

12.9.4 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 TypeError exception.
2. Let instOfHandler be GetMethod(C, @@hasInstance).
3. ReturnIfAbrupt(instOfHandler).
4. If instOfHandler is not undefined, then
   a. Let result be the result of calling the [[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 false, then throw a TypeError exception.
6. Return OrdinaryHasInstance(C, O).

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

12.10 Equality Operators

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

Syntax

EqualityExpression \(\equiv\) RelationalExpression

EqualityExpression \(\neq\) RelationalExpression

EqualityExpression \(===\) RelationalExpression

EqualityExpression \(!=\) RelationalExpression

12.10.1 Static Semantics: IsFunctionDefinition

See also: 12.2.0.2, 12.2.9.2, 12.3.1.2, 12.4.2, 12.5.2, 12.6.1, 12.7.1, 12.8.1, 12.9.1, 12.11.1, 12.12.1, 12.13.1, 12.14.2, 12.15.1, 14.1.11, 14.4.8, 14.5.8.

EqualityExpression :

EqualityExpression \(\equiv\) RelationalExpression
EqualityExpression \(!=\) RelationalExpression
EqualityExpression \(===\) RelationalExpression
EqualityExpression \(!==\) RelationalExpression
1. Return false.
12.10.2 Static Semantics: IsValidSimpleAssignmentTarget


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

1. Return false.

12.10.3 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

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 Strings values that are canonically equal according to the Unicode standard could test as unequal. In effect this algorithm assumes that both Strings are already in normalized form.

12.11 Binary Bitwise Operators

Syntax

```
BitwiseANDExpression\[\tau_{\text{valid}}\] :
  EqualityExpression\[\tau_{\text{valid}}\] & EqualityExpression\[\tau_{\text{valid}}\]

BitwiseXORExpression\[\tau_{\text{valid}}\] :
  BitwiseANDExpression\[\tau_{\text{valid}}\] & EqualityExpression\[\tau_{\text{valid}}\]
  BitwiseXORExpression\[\tau_{\text{valid}}\] ^ BitwiseANDExpression\[\tau_{\text{valid}}\]

BitwiseORExpression\[\tau_{\text{valid}}\] :
  BitwiseXORExpression\[\tau_{\text{valid}}\] | BitwiseORExpression\[\tau_{\text{valid}}\]
  BitwiseXORExpression\[\tau_{\text{valid}}\] ^ BitwiseXORExpression\[\tau_{\text{valid}}\]
```

12.11.1 Static Semantics: IsFunctionDefinition

See also: 12.2.0.2, 12.2.9.2, 12.3.1.2, 12.4.2, 12.5.2, 12.6.1, 12.7.1, 12.8.1, 12.9.1, 12.10.1, 12.12.1, 12.13.1, 12.14.2, 12.15.1, 14.1.11, 14.4.8, 14.5.8.

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

1. Return false.
12.11.2 Static Semantics: IsValidSimpleAssignmentTarget


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

1. Return false.

12.11.3 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.

12.12 Binary Logical Operators

Syntax

LogicalANDExpression::

BitwiseORExpression
   LogicalANDExpression && BitwiseORExpression

LogicalORExpression::

LogicalANDExpression
   LogicalORExpression || LogicalANDExpression

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.

12.12.1 Static Semantics: IsFunctionDefinition

See also: 12.2.0.2, 12.2.9.2, 12.3.1.2, 12.4.2, 12.5.2, 12.6.1, 12.7.1, 12.8.1, 12.9.1, 12.10.1, 12.11.1, 12.13.1, 12.14.2, 12.15.1, 14.1.11, 14.4.8, 14.5.8.

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

1. Return false.
12.12.2 Static Semantics: IsValidSimpleAssignmentTarget


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

1. Return false.

12.12.3 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).

12.13 Conditional Operator (?:)

Syntax


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.

12.13.1 Static Semantics: IsFunctionDefinition

See also: 12.2.0.2, 12.2.9.2, 12.3.1.2, 12.4.2, 12.5.2, 12.6.1, 12.7.1, 12.8.1, 12.9.1, 12.10.1, 12.11.1, 12.12.1, 12.12.2, 12.14.2, 12.15.1, 14.1.11, 14.4.8, 14.5.8.

ConditionalExpression : LogicalORExpression ? AssignmentExpression : AssignmentExpression

1. Return false.
12.13.2 Static Semantics: IsValidSimpleAssignmentTarget

See also: 12.2.0.3, 12.2.9.3, 12.3.1.3, 12.4.3, 12.5.3, 12.6.2, 12.7.2, 12.8.2, 12.9.2, 12.10.2, 12.11.2, 12.12.2, 12.13.4, 12.15.2.

ConditionalExpression : LogicalORExpression ? AssignmentExpression : AssignmentExpression
  1. Return false.

12.13.3 Runtime Semantics: Evaluation

ConditionalExpression : LogicalORExpression ? AssignmentExpression : AssignmentExpression
  1. Let lref be the result of evaluating LogicalORExpression.
  2. Let lval be ToBooleanGetValue(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).

12.14 Assignment Operators

Syntax

AssignmentExpression[Yield] : 
  ConditionalExpression[Yield]
  [Yield] YieldExpression
  ArrowFunction[Yield]
  LeftHandSideExpression[Yield] = AssignmentExpression[
  AssignmentOperator AssignmentExpression]

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

12.14.1 Static Semantics: Early Errors

AssignmentExpression : LeftHandSideExpression = AssignmentExpression

- It is a Syntax Error if LeftHandSideExpression is either an ObjectLiteral or an ArrayLiteral and 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 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 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 of LeftHandSideExpression is false.

12.14.2 Static Semantics: IsFunctionDefinition

See also: 12.2.0.2, 12.2.9.2, 12.3.1.2, 12.4.2, 12.5.2, 12.6.1, 12.7.1, 12.8.1, 12.9.1, 12.10.1, 12.11.1, 12.12.1, 12.13.1, 12.15.1, 14.1.11, 14.4.8, 14.5.8.

AssignmentExpression : ArrowFunction

1. Return true.

AssignmentExpression : YieldExpression

LeftHandSideExpression = AssignmentExpression

LeftHandSideExpression AssignmentOperator AssignmentExpression

1. Return false.

12.14.3 Static Semantics: IsValidSimpleAssignmentTarget

See also: 12.2.0.3, 12.2.9.3, 12.3.1.3, 12.4.3, 12.5.3, 12.6.2, 12.7.2, 12.8.2, 12.9.2, 12.10.2, 12.11.2, 12.12.2, 12.13.2, 12.15.2.

AssignmentExpression :

YieldExpression

ArrowFunction

LeftHandSideExpression AssignmentOperator AssignmentExpression

1. Return false.

12.14.4 Runtime Semantics: Evaluation

AssignmentExpression\[, Yield\] : LeftHandSideExpression\[Yield\] = AssignmentExpression\[, Yield\]

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 rval be GetValue(rref).
   d. If IsAnonymousFunctionDefinition(AssignmentExpression) and IsIdentifierRef of LeftHandSideExpression are both true, then
      i. Let hasNameProperty be HasOwnProperty(rval, "name").
      ii. ReturnIfAbrupt(hasNameProperty).
      iii. If hasNameProperty is false, then
         1. SetFunctionName(rval, GetReferencedName(lref)).
         2. Assert: SetFunctionName will not return an abrupt completion.
   e. Let status be PutValue(lref, rval).
   f. 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 ref be the result of evaluating AssignmentExpression.
4. Let rval be ToObject(GetValue(ref)).
5. ReturnIfAbrupt(rval).
6. Let status be the result of performing DestructuringAssignmentEvaluation of AssignmentPattern using rval as the argument.
7. ReturnIfAbrupt(status).
8. Return rval.

AssignmentExpression : LeftHandSideExpression AssignmentOperator AssignmentExpression
1. Let lref be the result of evaluating LeftHandSideExpression.
2. Let lval be GetValue(lref).
3. ReturnIfAbrupt(lval).
4. Let rref be the result of evaluating AssignmentExpression.
5. Let rval be GetValue(rref).
6. ReturnIfAbrupt(rval).
7. Let operator be the @ where AssignmentOperator is @=
8. Let r be the result of applying operator @ to lval and rval.
9. Let status be PutValue(lref, r).
10. ReturnIfAbrupt(status).
11. Return r.

NOTE When an assignment occurs within strict mode code, it is a runtime error if lref in step 1 of the first algorithm or step 9 of the second algorithm is an unresolved reference. If it is, a ReferenceError exception is thrown. The LeftHandSide also may not be a reference to a data property with the attribute value {
Writable]: false}, to an accessor property with the attribute value {
Set]: undefined}, or 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.

12.14.5 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] : 
  [ Elisionopt AssignmentRestElement[Yield]opt ] 
  [ AssignmentElementList[Yield], Elisionopt AssignmentRestElement[Yield]opt ] 
  [ AssignmentElementList[Yield], Elisionopt AssignmentRestElement[Yield]opt, Elisionopt AssignmentRestElement[Yield]opt ]
AssignmentPropertyList[yield]:
  AssignmentPropertyList[yield] , AssignmentProperty[yield]

AssignmentElementList[yield]:
  AssignmentElement[yield] , AssignmentElementList[yield] , AssignmentElement[yield]

AssignmentElement[yield]:
  Elision[yield] AssignmentElement[yield]

AssignmentProperty[yield]:
  IdentifierReference[yield] Initializer[yield], PropertyName : AssignmentElement[yield]

AssignmentElement[yield]:
  DestructuringAssignmentTarget[yield] Initializer[yield], PropertyName : AssignmentElement[yield]

AssignmentRestElement[yield]:
  . . . DestructuringAssignmentTarget[yield]

DestructuringAssignmentTarget[yield]:
  LeftHandSideExpression[yield]

12.14.5.1 Static Semantics: Early Errors

AssignmentProperty : IdentifierReference , Initializer[yield]
  • It is a Syntax Error if IsValidSimpleAssignment of IdentifierReference is false.
  • It is a Syntax Error if IdentifierReference statically resolves to an immutable binding.

AssignmentRestElement : . . . 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 CoverParenthesizedExpressionAndArrowParameterList : { 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.

Commented [AWB2028]: TODO: should define an abstract operation for this and use if here and several other places
NOTE The last rule means that the other rules are applied even if multiple levels of nested parentheses surround Expression.

12.14.5.2 Runtime Semantics: DestructuringAssignmentEvaluation

with parameter `obj`

**ObjectAssignmentPattern : { }**

1. Return NormalCompletion("").

**ArrayAssignmentPattern : [ ]**

1. Let `iterator` be GetIterator(`obj`).
2. ReturnIfAbrupt(`iterator`).
3. Return NormalCompletion("").

**ArrayAssignmentPattern : [ Elision ]**

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

**ArrayAssignmentPattern : [ Elision opt AssignmentRestElement ]**

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

**ArrayAssignmentPattern : [ AssignmentElementList , Elision opt AssignmentRestElement opt ]**

1. Let `iterator` be 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 not present, then return `status`. 
7. Return the result of performing IteratorDestructuringAssignmentEvaluation of AssignmentRestElement with iterator as the argument.

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 Initializer

1. Let P be StringValue of IdentifierReference.
2. Let v be Get(obj, P).
3. ReturnIfAbrupt(v).
4. If Initializer is present and v is undefined, then
   a. Let defaultValue be the result of evaluating Initializer.
   b. Let v be GetValue(defaultValue).
   c. ReturnIfAbrupt(v).
5. Let lref be 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.

12.14.5.3 Runtime Semantics: IteratorDestructuringAssignmentEvaluation

with parameters iterator

AssignmentElementList : AssignmentElisionElement

1. Return the result of performing IteratorDestructuringAssignmentEvaluation of AssignmentElisionElement 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 AssignmentElisionElement 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 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 IteratorStep(iterator).

AssignmentElement[Yield] ; DestructuringAssignmentTarget Initializer[Yield]

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 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 Initializer is present and v is undefined, then
   a. Let defaultValue be the result of evaluating Initializer.
   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 either AssignmentPattern or AssignmentPattern[Yield] as the goal symbol depending upon whether this AssignmentElement has the Yield parameter.
   b. Let v be ToObject(v).
   c. ReturnIfAbrupt(v).
   d. Return the result of performing DestructuringAssignmentEvaluation of nestedAssignmentPattern with v as the argument.

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 Initializer.

AssignmentRestElement[Yield] ; . . . DestructuringAssignmentTarget

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 A be ArrayCreate(0).
3. Let n=0;
4. Repeat
a. Let next be 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 CreateDataPropertyOrThrow(A, ToString(ToUint32(n)), nextValue).
g. ReturnIfAbrupt(defineStatus).
h. Increment n by 1.

5. If DestructuringAssignmentTarget is an ObjectLiteral or an ArrayLiteral then
   a. Let AssignmentPattern be the parse of the source code corresponding to
      DestructuringAssignmentTarget using either AssignmentPattern or AssignmentPattern[ Yield ]
      as the goal symbol depending upon whether this AssignmentRestElement has the Yield parameter.
   b. Return the result of performing DestructuringAssignmentEvaluation of
      AssignmentPattern with A as the argument.

12.14.5.4 Runtime Semantics: KeyedDestructuringAssignmentEvaluation

with parameters obj and propertyName

AssignmentElement[ Yield ]: DestructuringAssignmentTarget  Initialization
1. Let v be Get(obj, propertyName).
2. ReturnIfAbrupt(v).
3. If Initialization is present and v is undefined, then
   a. Let defaultValue be the result of evaluating Initialization.
   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 either AssignmentPattern or AssignmentPattern[ Yield ]
      as the goal symbol depending upon whether this AssignmentElement has the Yield parameter.
   b. Let v be ToObject(v).
   c. ReturnIfAbrupt(v).
   d. Return the result of performing DestructuringAssignmentEvaluation of AssignmentPattern with
      v as the argument.
5. Let lref be the result of evaluating DestructuringAssignmentTarget.

12.15 Comma Operator ( , )

Syntax

Expression[ Yield ] , Expression

Expression[ Yield ] , AssignmentExpression

12.15.1 Static Semantics: IsFunctionDefinition

See also: 12.2.0.2, 12.2.9.2, 12.3.1.2, 12.4.2, 12.5.2, 12.6.1, 12.7.1, 12.8.1, 12.9.1, 12.10.1, 12.11.1,

Commented [AW29]: Note that indices wrap. For example consider:
[ , , , , , , , , { 4294967293: “x”, length: Math.pow(2,32)-2 } ]
Expression : Expression , AssignmentExpression
  1. Return false.

12.15.2 Static Semantics: IsValidSimpleAssignmentTarget


Expression : Expression , AssignmentExpression
  1. Return false.

12.15.3 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[Yield, Return] :
  BlockStatement[Yield, Return]
  VariableStatement[Yield]
  EmptyStatement
  ExpressionStatement[Yield]
  IfStatement[Yield, Return]
  BreakableStatement[Yield, Return]
  ContinueStatement[Yield]
  BreakStatement[Yield]
  (1+Return) ReturnStatement[Yield]
  WithStatement[Yield, Return]
  LabelledStatement[Yield, Return]
  ThrowStatement[Yield]
  TryStatement[Yield, Return]
  DebuggerStatement

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

BreakableStatement[Yield, Return] :
  IterationStatement[Yield, Return]
  SwitchStatement[Yield, Return]
13.0 Statement Semantics

13.0.1 Static Semantics: DeclarationPart

Declaration : FunctionDeclaration
  1. Return FunctionDeclaration.

Declaration : GeneratorDeclaration
  1. Return GeneratorDeclaration.

Declaration : ClassDeclaration
  1. Return ClassDeclaration.

Declaration : LexicalDeclaration
  1. Return LexicalDeclaration.

13.0.2 Static Semantics: VarDeclaredNames


Statement :
  EmptyStatement
  ExpressionStatement
  ContinueStatement
  BreakStatement
  ReturnStatement
  ThrowStatement
  DebuggerStatement
  1. Return a new empty List.

13.0.3 Static Semantics: VarScopedDeclarations


Statement :
  EmptyStatement
  ExpressionStatement
  ContinueStatement
  BreakStatement
  ReturnStatement
  ThrowStatement
  DebuggerStatement
  1. Return a new empty List.

13.0.4 Runtime Semantics: LabelledEvaluation

  With argument labelSet.
See also: 13.6.1.2, 13.6.2.2, 13.6.3.3, 13.6.4.7, 13.12.12.

**BreakableStatement : IterationStatement**

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

**BreakableStatement : SwitchStatement**

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

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

### 13.0.5 Runtime Semantics: Evaluation

**BreakableStatement : IterationStatement**

SwitchStatement

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

### 13.1 Block

**Syntax**

BlockStatement[Yield Return] :

Block[Yield Return]

Block[Yield Return] :

{} StatementList[Yield Return]

StatementList[Yield Return] :

StatementListItem[Yield Return]

StatementList[Yield Return] StatementListItem[Yield Return]

StatementListItem[Yield Return] :

Statement[Yield Return]

Declaration[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: LexicallyDeclaredNames

See also: 13.11.2, 13.11.4, 14.1.14, 14.2.10, 15.1.3, 15.2.0.10.

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. If Statement is Statement : LabelledStatement, then return LexicallyDeclaredNames of Statement.
  2. Return a new empty List.

StatementListItem : Declaration
  1. Return the BoundNames of Declaration.

13.1.3 Static Semantics: LexicalScopedDeclarations

See also: 13.11.3, 13.12.5, 14.1.15, 14.2.11, 15.1.4, 15.2.0.11.

StatementList : StatementList StatementListItem
  1. Let declarations be LexicallyScopedDeclarations of StatementList.
  2. Append to declarations the elements of the LexicallyScopedDeclarations of StatementListItem.
  3. Return declarations.

StatementListItem : Statement
  1. If Statement is Statement : LabelledStatement, then return LexicalScopedDeclarations of Statement.
  2. Return a new empty List.

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

13.1.4 Static Semantics: TopLevelLexicallyDeclaredNames

See also: 13.12.6.

StatementList : StatementList StatementListItem
  1. Let names be TopLevelLexicallyDeclaredNames of StatementList.
  2. Append to names the elements of the TopLevelLexicallyDeclaredNames of StatementListItem.
  3. Return names.
StatementListItem : Statement
  1. Return a new empty List.

StatementListItem : Declaration
  1. If Declaration is Declaration : FunctionDeclaration, then return a new empty List.
  2. If Declaration is Declaration : GeneratorDeclaration, then return a new empty List.
  3. Return the BoundNames of Declaration.

NOTE At the top level of a function, or script, function declarations are treated like var declarations rather than like lexical declarations.

13.1.5 Static Semantics: TopLevelLexicallyScopedDeclarations

See also: 13.12.7.

Block : { }
  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.

StatementListItem : Statement
  1. Return a new empty List.

StatementListItem : Declaration
  1. If Declaration is Declaration : FunctionDeclaration, then return new empty List.
  2. If Declaration is Declaration : GeneratorDeclaration, then return new empty List.
  3. Return a new List containing Declaration.

13.1.6 Static Semantics: TopLevelVarDeclaredNames

See also: 13.12.8.

Block : { }
  1. Return a new empty List.

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 BoundNames of Declaration.
  2. If Declaration is Declaration : GeneratorDeclaration, then return the BoundNames of Declaration.
  3. Return a new empty List.
StatementListItem : Statement
    1. If Statement is Statement : LabelledStatement, then return TopLevelVarDeclaredNames of Statement.
    2. Return VarDeclaredNames of Statement.

NOTE At the top level of a function or script, inner function declarations are treated like var declarations.

13.1.7 Static Semantics: TopLevelVarScopedDeclarations

See also: 13.12.9.

Block : { }
    1. Return a new empty List.

StatementList : StatementList StatementListItem
    1. Let declarations be TopLevelVarScopedDeclarations of StatementList.
    2. Append to declarations the elements of the TopLevelVarScopedDeclarations of StatementListItem.
    3. Return declarations.

StatementListItem : Statement
    1. If Statement is Statement : LabelledStatement, then TopLevelVarScopedDeclarations of Statement.
    2. Return VarScopedDeclarations of Statement.

StatementListItem : Declaration
    1. If Declaration is Declaration : FunctionDeclaration, then return a new List containing FunctionDeclaration.
    2. If Declaration is Declaration : GeneratorDeclaration, then return a new List containing GeneratorDeclaration.
    3. Return a new empty List.

13.1.8 Static Semantics: VarDeclaredNames


Block : { }
    1. Return a new empty List.

StatementList : StatementList StatementListItem
    1. Let names be VarDeclaredNames of StatementList.
    2. Append to names the elements of the VarDeclaredNames of StatementListItem.
    3. Return names.

StatementListItem : Declaration
    1. Return a new empty List.
13.1.9 Static Semantics: VarScopedDeclarations


**Block**: {}
1. Return a new empty List.

**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**: Declaration
1. 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 NewDeclarativeEnvironment(oldEnv).
3. Perform BlockDeclarationInstantiation(StatementList, blockEnv).
4. Let blockValue be the result of evaluating StatementList.
5. Set the running execution context’s LexicalEnvironment to blockEnv.
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:

```javascript
eval("1; ; ; ; ;")
eval("1; {}")
eval("1; var a;")
```
13.1.11 Runtime Semantics: BlockDeclarationInstantiation( code, env )

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

BlockDeclarationInstantiation 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 LexicallyScopedDeclarations of code.
2. For each element d in declarations do
   a. For each element dn of the BoundNames of d do
      i. If IsConstantDeclaration of d is true, then
         1. Call env’s CreateImmutableBinding concrete method passing dn as the argument.
      ii. Else.
         1. 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 production or a FunctionDeclaration 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. Call env’s InitializeBinding concrete method passing fn, and fo as the arguments.

13.2 Declarations and the Variable Statement

13.2.1 Let and Const Declarations

NOTE 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 Initializer is assigned the value of its Initializer’s AssignmentExpression when the LexicalBinding is evaluated, not when the variable is created. If a LexicalBinding in a let declaration does not have an Initializer the variable is assigned the value undefined when the LexicalBinding is evaluated.

Syntax

LexicalDeclaration[In, Yield]:
  LetOrConst BindingList[?In, ?Yield];

LetOrConst:
  let
  const

BindingList[?In, ?Yield]:
  LexicalBinding[?In, ?Yield]
  BindingList[?In, ?Yield], LexicalBinding[?In, ?Yield]

LexicalBinding[?In, ?Yield]:
  BindingIdentifier[?Yield], Initializer[?In, ?Yield][opt]
  BindingPattern[?Yield], Initializer[?In, ?Yield][opt]
13.2.1.1 Static Semantics: Early Errors

LexicalDeclaration : LetOrConst BindingList ;
  - It is a Syntax Error if the BoundNames of BindingList contains "let".
  - It is a Syntax Error if the BoundNames of BindingList contains any duplicate entries.

LexicalBinding : BindingIdentifier Initializeropt
  - It is a Syntax Error if Initializer is not present and IsConstantDeclaration of the LexicalDeclaration containing this production is true.

13.2.1.2 Static Semantics: BoundNames

See also: 12.1.2, 13.6.4.2, 14.1.3, 14.2.2, 14.4.2, 14.5.2, 15.2.1.2, 15.2.2.1.

LexicalDeclaration : LetOrConst BindingList ;
  1. Return the BoundNames of BindingList.

BindingList : BindingList , LexicalBinding
  1. Let names be the BoundNames of BindingList.
  2. Append to names the elements of the BoundNames of LexicalBinding.
  3. Return names.

LexicalBinding : BindingIdentifier Initializeropt
  1. Return the BoundNames of BindingIdentifier.

LexicalBinding : BindingPattern Initializer
  1. Return the BoundNames of BindingPattern.

13.2.1.3 Static Semantics: IsConstantDeclaration

See also: 14.1.8, 14.4.5, 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 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 BindingInitialization for BindingIdentifier passing undefined and env as the arguments.

**NOTE** A static semantics rule ensures that this form of LexicalBinding never occurs in a const declaration.

**LexicalBinding : BindingIdentifier Initialize**

1. Let rhs be the result of evaluating Initialize.
2. Let value be GetValue(rhs).
3. ReturnIfAbrupt(value).
4. If IsAnonymousFunctionDefinition(Initializer) is true, then
   a. Let hasNameProperty be HasOwnProperty(value, “name”).
   b. ReturnIfAbrupt(hasNameProperty).
   c. If hasNameProperty is false, then
      i. SetFunctionName(value, StringValue(BindingIdentifier)).
     ii. Assert: SetFunctionName will not return an abrupt completion.
5. Let env be the running execution context’s LexicalEnvironment.
6. Return the result of performing BindingInitialization for BindingIdentifier passing value and env as the arguments.

**LexicalBinding : BindingPattern Initialize**

1. Let rhs be the result of evaluating Initialize.
2. Let value be ToObject(GetValue(rhs)).
3. ReturnIfAbrupt(value).
4. Let env be the running execution context’s LexicalEnvironment.
5. Return the result of performing BindingInitialization 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 initialized to undefined when created. Within the scope of any VariableEnvironment a common bindingIdentifier may appear in more than one VariableDeclaration but those declarations collective define only one variable. A variable defined by a VariableDeclaration with an Initializer is assigned the value of its Initializer’s AssignmentExpression when the VariableDeclaration is executed, not when the variable is created.

**Syntax**

```
var VariableDeclarationList[?, ?];
```
VariableDeclarationList(\text{In} \{\text{Yield}\}) : 
\text{VariableDeclarationList(\text{In} \{\text{Yield}\})} \text{, VariableDeclaration(\text{In} \{\text{Yield}\})}

VariableDeclaration(\text{In} \{\text{Yield}\}) : 
\text{BindingIdentifier(\text{Yield}) \text{Initializer(\text{Yield})}} 
\text{BindingPattern(\text{Yield}) \text{Initializer(\text{Yield})}}

13.2.2.1 Static Semantics: BoundNames

See also: 13.2.1.2, 12.1.2, 13.6.4.2, 14.1.3, 14.2.2, 14.4.2, 14.5.2, 15.2.1.2, 15.2.2.1.

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 Initializer
1. Return the BoundNames of BindingIdentifier.

VariableDeclaration : BindingPattern Initializer
1. Return the BoundNames of BindingPattern.

13.2.2.2 Static Semantics: VarDeclaredNames


VariableStatement : var VariableDeclarationList
1. Return BoundNames of VariableDeclarationList.

13.2.2.3 Static Semantics: VarScopedDeclarations


VariableDeclarationList : VariableDeclaration
1. Return a new List containing VariableDeclaration.

VariableDeclarationList : VariableDeclarationList, VariableDeclaration
1. Let declarations be VarScopedDeclarations of VariableDeclarationList.
2. Append VariableDeclaration to declarations.
3. Return declarations.

13.2.2.4 Runtime Semantics: BindingInitialization

With arguments \text{value} and \text{environment}.

See also: 12.1.4, 13.2.3.5, 13.6.4.5, 13.14.4.
NOTE undefined is passed for environment to indicate that a PutValue operation should be used to assign the initialization value. This is the case for var statements and the formal parameter lists of some non-strict functions (see 9.2.13). In those cases a lexical binding is hoisted and preinitialized prior to evaluation of its initializer.

VariableDeclaration : BindingIdentifier
1. Return the result of performing BindingInitialization for BindingIdentifier passing value and undefined as the arguments.

VariableDeclaration : BindingIdentifier Initializer
1. Return the result of performing BindingInitialization for BindingIdentifier passing value and undefined as the arguments.

VariableDeclaration : BindingPattern Initializer
1. Return the result of performing BindingInitialization for BindingPattern passing value and undefined as the arguments.

13.2.2.5 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 Initializer
1. Let rhs be the result of evaluating Initializer.
2. Let value be GetValue(rhs).
3. ReturnIfAbrupt(value).
4. If IsAnonymousFunctionDefinition(Initializer) is true, then
   a. Let hasNameProperty be HasOwnProperty(value, "name").
   b. ReturnIfAbrupt(hasNameProperty).
   c. If hasNameProperty is false, then
      i. Perform SetFunctionName(value, StringValue(BindingIdentifier)).
      ii. Assert: SetFunctionName will not return an abrupt completion.
5. Return the result of performing BindingInitialization for BindingIdentifier passing value and undefined as the arguments.

NOTE If a VariableDeclaration is nested within a with statement and the BindingIdentifier in the VariableDeclaration is the same as a property name of the binding object of the with statement’s object environment record, then step 5 will assign value to the property instead of assigning to the VariableEnvironment binding of the Identifier.
VariableDeclaration: BindingPattern Initializer

1. Let rhs be the result of evaluating Initializer.
2. Let rval be ToObject(GetValue(rhs)).
3. ReturnIfAbrupt(rval).
4. Return the result of performing BindingInitialization for BindingPattern passing rval and undefined as arguments.

13.2.3 Destructuring Binding Patterns

Syntax

```
BindingPattern[Yield,GeneratorParameter]
  ObjectBindingPattern[Yield,GeneratorParameter]
  ArrayBindingPattern[Yield,GeneratorParameter]

ObjectBindingPattern[Yield,GeneratorParameter]
  { }
  { BindingPropertyList[Yield,GeneratorParameter] }
  { BindingPropertyList[Yield,GeneratorParameter], }

ArrayBindingPattern[Yield,GeneratorParameter]
  [ ]
  [ BindingRestElement[Yield,GeneratorParameter] ]
  [ BindingElementList[Yield,GeneratorParameter] , BindingRestElement[Yield,GeneratorParameter] ]

BindingPropertyList[Yield,GeneratorParameter]
  BindingProperty[Yield,GeneratorParameter]
  BindingPropertyList[Yield,GeneratorParameter], BindingProperty[Yield,GeneratorParameter]

BindingElementList[Yield,GeneratorParameter]
  BindingElisionElement[Yield,GeneratorParameter]
  BindingElementList[Yield,GeneratorParameter], BindingElisionElement[Yield,GeneratorParameter]

BindingElisionElement[Yield,GeneratorParameter]
  Elisionopt BindingElement[Yield,GeneratorParameter]

BindingProperty[Yield,GeneratorParameter]
  SingleNameBinding[Yield,GeneratorParameter]
  PropertyName[Yield,GeneratorParameter]: BindingElement[Yield,GeneratorParameter]

BindingElement[Yield,GeneratorParameter]
  SingleNameBinding[Yield,GeneratorParameter]
  [ GeneratorParameter BindingPattern[Yield,GeneratorParameter] Initializer[opt]
  [ GeneratorParameter BindingPattern[Yield,GeneratorParameter] Initializer[~], Yield[opt]

SingleNameBinding[Yield,GeneratorParameter]
  [ GeneratorParameter BindingIdentifier[Yield] Initializer[opt]
  [ GeneratorParameter BindingIdentifier[Yield] Initializer[~], Yield[opt]

BindingRestElement[Yield,GeneratorParameter]
  [ GeneratorParameter . . . BindingIdentifier[Yield]
  [ GeneratorParameter . . . BindingIdentifier[~Yield]
```
13.2.3.1 Static Semantics: BoundNames

See also: 13.2.1.2, 13.2.2.1, 13.6.4.2, 14.1.3, 14.2.2, 14.4.2, 14.5.2, 15.2.1.2, 15.2.2.1.

ObjectBindingPattern : { }
  1. Return an empty List.

ArrayBindingPattern : [ ]
  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 Initializeropt
  1. Return the BoundNames of BindingIdentifier.

BindingElement : BindingPattern Initializeropt
  1. Return the BoundNames of BindingPattern.

13.2.3.2 Static Semantics: ContainsExpression

See also: 14.1.5, 14.2.4.
ObjectBindingPattern : { }
   1. Return false.

ArrayBindingPattern : [ Elisionopt ]
   1. Return false.

ArrayBindingPattern : [ Elisionopt BindingRestElement ]
   1. Return false.

ArrayBindingPattern : [ BindingElementList , Elisionopt ]
   1. Return ContainsExpression of BindingElementList.

ArrayBindingPattern : [ BindingElementList , Elisionopt BindingRestElement ]
   1. Return ContainsExpression of BindingElementList.

BindingPropertyList : BindingPropertyList , BindingProperty
   1. Let has be ContainsExpression of BindingPropertyList.
   2. If has is true, return true.
   3. Return ContainsExpression of BindingProperty.

BindingElementList : BindingElementList , BindingElisionElement
   1. Let has be ContainsExpression of BindingElementList.
   2. If has is true, return true.
   3. Return ContainsExpression of BindingElisionElement.

BindingElisionElement : Elisionopt BindingElement
   1. Return ContainsExpression of BindingElement.

BindingProperty : PropertyName : BindingElement
   1. Let has be IsComputedPropertyKey of PropertyName.
   2. If has is true, return true.
   3. Return the ContainsExpression of BindingElement.

BindingElement : BindingPattern Initializer
   1. Return true.

SingleNameBinding : BindingIdentifier
   1. Return false.

SingleNameBinding : BindingIdentifier Initializer
   1. Return true.

13.2.3.3 Static Semantics: HasInitializer

See also: 13.2.3.3, 14.1.7, 14.2.7.
BindingElement : BindingPattern
  1. Return false.

BindingElement : BindingPattern Initializer
  1. Return true.

SingleNameBinding : BindingIdentifier
  1. Return false.

SingleNameBinding : BindingIdentifier Initializer
  1. Return true.

13.2.3.4 Static Semantics: IsSimpleParameterList

See also: 14.1.11, 14.2.8.

BindingElement : BindingPattern
  1. Return false.

BindingElement : BindingPattern Initializer
  1. Return false.

SingleNameBinding : BindingIdentifier
  1. Return true.

SingleNameBinding : BindingIdentifier Initializer
  1. Return false.

13.2.3.5 Runtime Semantics: BindingInitialization

With parameters value and environment.

See also: 12.1.4, 13.2.2.4, 13.6.4.5, 13.14.4.

NOTE When undefined is passed for environment it indicates that a PutValue operation should be used to assign the initialization value. This is the case for formal parameter lists of non-strict functions. In that case the formal parameter bindings are preinitialized in order to deal with the possibility of multiple parameters with the same name.

BindingPattern : ObjectBindingPattern
  1. Assert: Type(value) is Object
  2. Return the result of performing BindingInitialization for ObjectBindingPattern using value and environment as arguments.

BindingPattern : ArrayBindingPattern
  1. Assert: Type(value) is Object
  2. Let iterator be GetIterator(value).
  3. ReturnIfAbrupt(iterator).
4. Return the result of performing IteratorBindingInitialization for ArrayBindingPattern using iterator, and environment as arguments.

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

BindingPropertyList : BindingPropertyList , BindingProperty
  1. Let status be the result of performing BindingInitialization for BindingPropertyList using value and environment as arguments.
  2. ReturnIfAbrupt(status).
  3. Return the result of performing BindingInitialization 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 KeyedBindingInitialization 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 KeyedBindingInitialization for BindingElement using value, environment, and P as arguments.

13.2.3.6 Runtime Semantics: IteratorBindingInitialization

With parameters iterator, and environment.

See also: 14.1.20, 14.2.15.

NOTE When undefined is passed for environment it indicates that a PutValue operation should be used to assign the initialization value. This is the case for formal parameter lists of non-strict functions. In that case the formal parameter bindings are preinitialized in order to deal with the possibility of multiple parameters with the same name.

ArrayBindingPattern : [ ]
  1. Return NormalCompletion(empty).

ArrayBindingPattern : [ Elision ]
  1. Return the result of performing IteratorDestructuringAssignmentEvaluation of Elision with iterator as the argument.

ArrayBindingPattern : [ Elisionopt 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 IteratorBindingInitialization for BindingRestElement using iterator and environment as arguments.
ArrayBindingPattern : [ BindingElementList ]
1. Return the result of performing IteratorBindingInitialization for BindingElementList using iterator and environment as arguments.

ArrayBindingPattern : [ BindingElementList , ]
1. Return the result of performing IteratorBindingInitialization for BindingElementList using iterator and environment as arguments.

ArrayBindingPattern : [ BindingElementList , Elision ]
1. Let status be the result of performing IteratorBindingInitialization 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 , Elisionopt BindingRestElement ]
1. Let status be the result of performing IteratorBindingInitialization 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 IteratorBindingInitialization for BindingRestElement using iterator and environment as arguments.

BindingElementList : BindingElisionElement
1. Return the result of performing IteratorBindingInitialization for BindingElisionElement using iterator and environment as arguments.

BindingElementList : BindingElementList , BindingElisionElement
1. Let status be the result of performing IteratorBindingInitialization for BindingElementList using iterator and environment as arguments.
2. ReturnIfAbrupt(status).
3. Return the result of performing IteratorBindingInitialization for BindingElisionElement using iterator and environment as arguments.

BindingElisionElement : BindingElement
1. Return the result of performing IteratorBindingInitialization of BindingElement with iterator and environment as the arguments.

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 IteratorBindingInitialization of BindingElement with iterator and environment as the arguments.
BindingElement : SimpleNameBinding
1. Return the result of performing IteratorBindingInitialization for SimpleNameBinding using iterator and environment as the arguments.

SimpleNameBinding : BindingIdentifier InitializerOpt
1. Let next be 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 Initializer is present and v is undefined, then
   a. Let defaultValue be the result of evaluating Initializer.
   b. Let v be GetValue(defaultValue).
   c. ReturnIfAbrupt(v).
   d. If IsAnonymousFunctionDefinition(Initializer) is true, then
      i. Let hasNameProperty be HasOwnProperty(v, "name").
      ii. ReturnIfAbrupt(hasNameProperty).
      iii. If hasNameProperty is false, then
         1. SetFunctionName(v, StringValue(BindingIdentifier)).
         2. Assert: SetFunctionName will not return an abrupt completion.
6. Return the result of performing BindingInitialization for BindingIdentifier passing v and environment as arguments.

BindingElement : BindingPattern InitializerOpt
1. Let next be 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 Initializer is present and v is undefined, then
   a. Let defaultValue be the result of evaluating Initializer.
   b. Let v be ToObject(GetValue(defaultValue)).
   c. ReturnIfAbrupt(v).
6. Return the result of performing BindingInitialization of BindingPattern with v and environment as the arguments.

BindingRestElement : . . . BindingIdentifier
1. Let A be ArrayCreate(0).
2. Let n=0.
3. Repeat,
   a. Let next be IteratorStep(iterator).
   b. ReturnIfAbrupt(next).
   c. If next is false, then
      i. Return the result of performing BindingInitialization for BindingIdentifier using A and environment as arguments.
   d. Let nextValue be IteratorValue(next).
   e. ReturnIfAbrupt(nextValue).
   f. Let defineStatus be CreateDataPropertyOrThrow(A, ToString(ToUint32(n)), nextValue).
   g. SetDataPropertyOrThrow(A, n, nextValue).
   h. n=n+1.
13.2.3.7 Runtime Semantics: KeyedBindingInitialization

With parameters obj, environment, and propertyName.

**NOTE** When undefined is passed for environment it indicates that a PutValue operation should be used to assign the initialization value. This is the case for formal parameter lists of non-strict functions. In that case the formal parameter bindings are preinitialized in order to deal with the possibility of multiple parameters with the same name.

**BindingElement** : **BindingPattern** Initializeopt

1. Let v be Get(obj, propertyName).
2. ReturnIfAbrupt(v).
3. If Initialize is present and v is undefined, then
   a. Let defaultValue be the result of evaluating Initialize.
   b. Let v be ToObject(GetValue(defaultValue)).
   c. ReturnIfAbrupt(v).
4. Return the result of performing BindingInitialization for BindingPattern passing v and environment as arguments.

**SingleNameBinding** : **BindingIdentifier** Initializeopt

1. Let v be Get(obj, propertyName).
2. ReturnIfAbrupt(v).
3. If Initialize is present and v is undefined, then
   a. Let defaultValue be the result of evaluating Initialize.
   b. Let v be GetValue(defaultValue).
   c. ReturnIfAbrupt(v).
   d. If IsAnonymousFunctionDefinition(Initialize) is true, then
      i. Let hasNameProperty be HasOwnProperty(v, "name").
      ii. ReturnIfAbrupt(hasNameProperty).
      iii. If hasNameProperty is false, then
         1. SetFunctionName(v, StringValue(BindingIdentifier)).
         2. Assert: SetFunctionName will not return an abrupt completion.
4. Return the result of performing BindingInitialization 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 : Expression ;

NOTE: An ExpressionStatement cannot start with an opening curly brace because that might make it ambiguous with a Block. Also, an ExpressionStatement cannot start with the function or class keywords because that would make it ambiguous with a FunctionDeclaration, a GeneratorDeclaration, or a ClassDeclaration. 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. Return GetValue(exprRef).

13.5 The if Statement

Syntax

IfStatement : if ( Expression ) Statement else Statement ;

Each else for which the choice of associated if is ambiguous shall be associated with the nearest possible if that would otherwise have no corresponding else.

13.5.1 Static Semantics: Early Errors

IfStatement :

if ( Expression ) Statement else Statement

It is a Syntax Error if IsLabelledFunction(Statement) is true for any occurrence of Statement in these rules.

NOTE: It is only necessary to apply this rule if the extension specified in B.3.2 is implemented.

13.5.2 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: \texttt{if} \ (Expression) \ Statement
\begin{enumerate}
\item Return the VarDeclaredNames of Statement.
\end{enumerate}

13.5.3 Static Semantics: VarScopedDeclarations


IfStatement: \texttt{if} \ (Expression) \ Statement \texttt{else} \ Statement
\begin{enumerate}
\item Let declarations be VarScopedDeclarations of the first Statement.
\item Append to declarations the elements of the VarScopedDeclarations of the second Statement.
\item Return declarations.
\end{enumerate}

IfStatement: \texttt{if} \ (Expression) \ Statement
\begin{enumerate}
\item Return the VarDeclaredNames of Statement.
\end{enumerate}

13.5.4 Runtime Semantics: Evaluation

IfStatement: \texttt{if} \ (Expression) \ Statement \texttt{else} \ Statement
\begin{enumerate}
\item Let exprRef be the result of evaluating Expression.
\item Let exprValue be ToBoolean(GetValue(exprRef)).
\item ReturnIfAbrupt(exprValue).
\item If exprValue is \texttt{true}, then
\begin{enumerate}
\item Let stmtValue be the result of evaluating the first Statement.
\item Return(stmtValue).
\end{enumerate}
\item Else,
\begin{enumerate}
\item Let stmtValue be the result of evaluating the second Statement.
\item If stmtValue.\[\text{[\text{type}]}\] is normal and stmtValue.\[\text{[\text{value}]}\] is empty, then
\begin{enumerate}
\item Return NormalCompletion(\texttt{undefined}).
\end{enumerate}
\item Return stmtValue.
\end{enumerate}
\end{enumerate}

IfStatement: \texttt{if} \ (Expression) \ Statement
\begin{enumerate}
\item Let exprRef be the result of evaluating Expression.
\item Let exprValue be ToBoolean(GetValue(exprRef)).
\item ReturnIfAbrupt(exprValue).
\item If exprValue is \texttt{false}, then
\begin{enumerate}
\item Let stmtValue be the result of evaluating Statement.
\item If stmtValue.\[\text{[\text{type}]}\] is normal and stmtValue.\[\text{[\text{value}]}\] is empty, then
\begin{enumerate}
\item Return NormalCompletion(\texttt{undefined}).
\end{enumerate}
\item Return stmtValue.
\end{enumerate}
\item Else,
\begin{enumerate}
\item Let stmtValue be the result of evaluating Statement.
\item Return NormalCompletion(\texttt{undefined}).
\item Return stmtValue.
\end{enumerate}
\end{enumerate}

13.6 Iteration Statements

Syntax

\texttt{IterationStatement} \|
\begin{align*}
\texttt{do} \ \texttt{Statement} \texttt{while} \ (Expression) \ &; \texttt{opt} \\
\texttt{while} \ (Expression) \ &; \texttt{opt} \ 	exttt{Statement}
\end{align*}
for ( [lookahead = \[let \{\}] Expressionopt ; Expressionopt ; Expressionopt )
    Statementopt
for ( [lookahead = \[let \{\}] VariableDeclarationListopt ; Expressionopt ; Expressionopt )
    Statementopt
for ( [lookahead = \[let \{\}] LexicalDeclarationopt Expressionopt ; Expressionopt )
    Statementopt
for ( [lookahead = \[let \{\}] LeftHandSideExpressionopt in Expressionopt )
    Statementopt
for ( [lookahead = \[let \{\}] ForBindingopt In Expressionopt )
    Statementopt
for ( [lookahead = \[let \{\}] LeftHandSideExpressionopt of AssignmentExpressionopt )
    Statementopt
for ( [lookahead = \[let \{\}] ForDeclarationopt of AssignmentExpressionopt )
    Statementopt
for ( [lookahead = \[let \{\}] ForBindingopt of AssignmentExpressionopt )
    Statementopt

ForDeclarationopt:
    let or const ForBindingopt

ForBindingopt:
    BindingIdentifieropt
    BindingPatternopt

NOTE 1 A semicolon is not required after a do-while statement

13.6.0 Semantics

13.6.0.1 Static Semantics: Early Errors

IterationStatement:
    do Statement while ( Expression ) ; opt
    while ( Expression ) Statement
for ( [lookahead = \[let \{\}] Expressionopt ; Expressionopt ; Expressionopt )
    Statementopt
for ( [lookahead = \[let \{\}] VariableDeclarationListopt ; Expressionopt ; Expressionopt )
    Statementopt
for ( [lookahead = \[let \{\}] LexicalDeclarationopt Expressionopt ; Expressionopt )
    Statementopt
for ( [lookahead = \[let \{\}] LeftHandSideExpressionopt in Expressionopt )
    Statementopt
for ( [lookahead = \[let \{\}] ForBindingopt In Expressionopt )
    Statementopt
for ( [lookahead = \[let \{\}] LeftHandSideExpressionopt of AssignmentExpressionopt )
    Statementopt
for ( [lookahead = \[let \{\}] ForBindingopt of AssignmentExpressionopt )
    Statementopt
for ( [lookahead = \[let \{\}] ForDeclarationopt of AssignmentExpressionopt )
    Statementopt
for ( [lookahead = \[let \{\}] ForBindingopt of AssignmentExpressionopt )
    Statementopt

- It is a Syntax Error if IsLabelledFunction(Statement) is true for any occurrence of Statement in these rules.

NOTE It is only necessary to apply this rule if the extension specified in B.3.2 is implemented.

13.6.0.2 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 \( \text{completion}[[\text{target}]] \) is empty, then return true.
4. If \( \text{completion}[[\text{target}]] \) 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


IterationStatement : do Statement while ( Expression );

1. Return the VarDeclaredNames of Statement.

13.6.1.2 Static Semantics: VarScopedDeclarations


IterationStatement : do Statement while ( Expression );

1. Return the VarScopedDeclarations of Statement.

13.6.1.3 Runtime Semantics: LabelledEvaluation

With argument labelSet.

See also: 13.0.3, 13.6.2.2, 13.6.3.3, 13.6.4.7, 13.12.12.

IterationStatement : do Statement while ( Expression );

1. Let \( V = \text{undefined} \).
2. Repeat
   a. Let stmt be the result of evaluating Statement.
   b. If LoopContinues (stmt,labelSet) is false, return stmt.
   c. If stmt.[[value]] is not empty, let \( V = \text{stmt}[[\text{value}]] \).
   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

IterationStatement: `while (Expression) Statement`

1. Return the `VarDeclaredNames` of `Statement`.

13.6.2.2 Static Semantics: `VarScopedDeclarations`


IterationStatement: `while (Expression) Statement`

1. Return the `VarScoped` Declarations of `Statement`.

13.6.2.3 Runtime Semantics: `LabelledEvaluation`

With argument `labelSet`.


IterationStatement: `while (Expression) Statement`

1. Let `V = undefined`.
2. Repeat
   a. Let `exprRef` be the result of evaluating `Expression`.
   b. Let `exprValue` be `ToBoolean(GetValue(exprRef))`.
   c. If `exprValue` is `false`, return NormalCompletion(`V`).
   d. If `exprValue` is not `true`, then
      i. Assert: `exprValue` is an abrupt completion.
      ii. If `LoopContinues(exprValue, labelSet)` is `false`, return `exprValue`.
   e. Let `stmt` be the result of evaluating `Statement`.
   f. If `LoopContinues(stmt, labelSet)` is `false`, return `stmt`.
   g. If `stmt.[[value]]` is not empty, let `V = stmt.[[value]]`.

13.6.3 The `for` Statement

13.6.3.1 Static Semantics: Early Errors

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

- It is a Syntax Error if any element of the `BoundNames` of `LexicalDeclaration` also occurs in the `VarDeclaredNames` of `Statement`.

13.6.3.2 Static Semantics: `VarDeclaredNames`


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`. 

---

Commented [AWB643]: ES5 breaking change: completion reform

Commented [AWB644]: Break/continue/return in the expression works normally (future for do {} or block lambda expressions)
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.

13.6.3.3 Static Semantics: VarScopedDeclarations


IterationStatement: for ( Expressionopt ; Expressionopt ; Expressionopt ) Statement
   1. Return the VarScopedDeclarations of Statement.

IterationStatement: for ( var VariableDeclarationList ; Expressionopt ; Expressionopt ) Statement
   1. Let declarations be VarScopedDeclarations of VariableDeclarationList.
   2. Append to declarations the elements of the VarScopedDeclarations of Statement.
   3. Return declarations.

IterationStatement: for ( LexicalDeclaration Expressionopt ; Expressionopt ) Statement
   1. Return the VarScopedDeclarations of Statement.

13.6.3.4 Runtime Semantics: LabelledEvaluation

With argument labelSet.


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 second Expression as the testExpr argument, the third Expression as the incrementExpr argument, Statement as the stmt argument, () as the perIterationBindings, and with labelSet.

IterationStatement: for ( var VariableDeclarationList ; Expressionopt ; Expressionopt ) Statement
   1. Let varDel be the result of evaluating VariableDeclarationList.
   2. If LoopContinues(varDel, labelSet) is false, return varDel.
   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, () as the perIterationBindings, and with labelSet.

IterationStatement: for ( LexicalDeclaration Expressionopt ; Expressionopt ) Statement
   1. Let oldEnv be the running execution context’s LexicalEnvironment.
   2. Let loopEnv be NewDeclarativeEnvironment(oldEnv).
   3. Let isConst be the result of performing IsConstantDeclaration of LexicalDeclaration.
4. Let `boundNames` be the BoundNames of `LexicalDeclaration`.
5. For each element `dn` of `boundNames` 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.
6. Set the running execution context's LexicalEnvironment to `loopEnv`.
7. Let `forDcl` be the result of evaluating `LexicalDeclaration`.
8. If `LoopContinues(forDcl, labelSet)` is `false`, then
   a. Set the running execution context's LexicalEnvironment to `oldEnv`.
   b. Return `forDcl`.
9. If `isConst` is `false`, let `perIterationLets` be `boundNames` otherwise let `perIterationLets` be `()`.  
10. 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, `perIterationLets` as the `perIterationBindings`, and with `labelSet`.
11. Set the running execution context's LexicalEnvironment to `oldEnv`.
12. Return `bodyResult`.

13.6.3.5 Runtime Semantics: ForBodyEvaluation

The abstract operation `ForBodyEvaluation` with arguments `testExpr`, `incrementExpr`, `stmt`, `perIterationBindings`, and `labelSet` is performed as follows:

1. Let `V = undefined`.
2. Let `status` be `CreatePerIterationEnvironment(perIterationBindings)`.
3. ReturnIfAbrupt(`status`).
4. Repeat
   a. If `testExpr` is not `[empty]`, then
      i. Let `testExprRef` be the result of evaluating `testExpr`.
      ii. Let `testExprValue` be `ToBoolean(GetValue(testExprRef))`.
      iii. If `testExprValue` is `false`, return NormalCompletion(`V`).
      iv. Else if `LoopContinues(testExprValue, labelSet)` is `false`, return `testExprValue`.
   b. Let `result` be the result of evaluating `stmt`.
   c. If `LoopContinues(result, labelSet)` is `false`, return `result`.
   d. If `result`.[[value]] is not `[empty]`.
      i. Set `V = result`.[[value]].
   e. ReturnIfAbrupt(`status`).
   f. If `incrementExpr` is not `[empty]`, then
      i. Let `incExprRef` be the result of evaluating `incrementExpr`.
      ii. Let `incExprValue` be `GetValue(incExprRef)`.
      iii. If `LoopContinues(incExprValue, labelSet)` is `false`, return `incExprValue`.

13.6.3.6 Runtime Semantics: CreatePerIterationEnvironment

The abstract operation `CreatePerIterationEnvironment` with argument `perIterationBindings` is performed as follows:

1. If `perIterationBindings` has any elements, then
   a. Let `lastIterationEnv` be the running execution context's LexicalEnvironment.
   b. Let `outer` be `lastIterationEnv`'s outer environment reference.
   c. Assert: `outer` is not `null`.

Commented [AWB645]: ESS breaking change: Completion reform
d. Let thisIterationEnv be NewDeclarativeEnvironment(outer).
e. For each element bn of perIterationBindings do,
   i. Let status be the result of calling thisIterationEnv’s CreateMutableBinding concrete
      method passing bn and false as the arguments.
   ii. Assert: status is never an abrupt completion.
   iii. Let lastValue be the result of calling lastIterationEnv’s GetBindingValue concrete method
        passing bn and true as the arguments.
   iv. ReturnIfAbrupt(lastValue).
   v. Call the InitializeBinding concrete method of thisIterationEnv passing bn and lastValue
      as the arguments.
   f. Set the running execution context’s LexicalEnvironment to thisIterationEnv.
2. Return undefined.

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 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 neither an ObjectLiteral nor an ArrayLiteral and
  IsValidSimpleAssignmentTarget of LeftHandSideExpression is false.
• It is a Syntax Error if the LeftHandSideExpression is
  CoverParenthesizedExpressionAndArrowParameterList : (Expression) and Expression 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 parentheses surround Expression.

IterationStatement:
  for (ForDeclaration in Expression) Statement
  for (ForDeclaration of AssignmentExpression) Statement

• It is a Syntax Error if the BoundNames of ForDeclaration contains “let”.
• It is a Syntax Error if any element of the BoundNames of ForDeclaration also occurs in the
  VarDeclaredNames of Statement.

ForBinding : BindingPattern

• It is a Syntax Error if the BoundNames of BindingPattern contains any duplicate entries.
13.6.4.2 Static Semantics: BoundNames

See also: 13.2.1.2, 13.2.2.1, 12.1.2, 14.1.3, 14.2.2, 14.4.2, 14.5.2, 15.2.1.2, 15.2.2.1.

ForDeclaration : LetOrConst ForBinding
  1. Return the BoundNames of ForBinding.

13.6.4.3 Static Semantics: VarDeclaredNames


IterationStatement : for ( LeftHandSideExpression in Expression ) Statement
  1. Return the VarDeclaredNames of Statement.

IterationStatement : for ( var ForBinding in Expression ) Statement
  1. Let names be the BoundNames of ForBinding.
  2. Append to names the elements of the VarDeclaredNames of Statement.
  3. Return names.

IterationStatement : for ( ForDeclaration in Expression ) Statement
  1. Return the VarDeclaredNames of Statement.

IterationStatement : for ( LeftHandSideExpression of AssignmentExpression ) Statement
  1. Return the VarDeclaredNames of Statement.

IterationStatement : for ( var ForBinding of AssignmentExpression ) Statement
  1. Let names be the BoundNames of ForBinding.
  2. Append to names the elements of the VarDeclaredNames of Statement.
  3. Return names.

IterationStatement : for ( ForDeclaration of AssignmentExpression ) Statement
  1. Return the VarDeclaredNames of Statement.

13.6.4.4 Static Semantics: VarScopedDeclarations


IterationStatement : for ( LeftHandSideExpression in Expression ) Statement
  1. Return the VarScopedDeclarations of Statement.

IterationStatement : for ( var ForBinding in Expression ) Statement
  1. Let declarations be a List containing ForBinding.
  2. Append to declarations the elements of the VarScopedDeclarations of Statement.
  3. Return declarations.
IterationStatement: `for (ForDeclaration in Expression) Statement`
1. Return the VarScopedDeclarations of Statement.

IterationStatement: `for (LeftHandSideExpression of AssignmentExpression) Statement`
1. Return the VarScopedDeclarations of Statement.

IterationStatement: `for (var ForBinding of AssignmentExpression) Statement`
1. Let declarations be a List containing ForBinding.
2. Append to declarations the elements of the VarScopedDeclarations of Statement.
3. Return declarations.

IterationStatement: `for (ForDeclaration of AssignmentExpression) Statement`
1. Return the VarScopedDeclarations of Statement.

13.6.4.5 Runtime Semantics: BindingInitialization

With arguments `value` and `environment`.

See also: 12.1.4, 13.2.2.4, 13.2.3.5, 13.14.4.

NOTE `undefined` is passed for `environment` to indicate that a PutValue operation should be used to assign the initialization value. This is the case for `var` statements and the formal parameter lists of some non-strict functions (see 9.2.13). In those cases a lexical binding is hoisted and preinitialized prior to evaluation of its initializer.

ForBinding: BindingPattern
1. If Type(`value`) is not Object, then throw a `TypeError` exception.
2. Return the result of performing BindingInitialization for BindingPattern passing `value` and `environment` as the arguments.

13.6.4.6 Runtime Semantics: BindingInstantiation

With arguments `value` and `environment`.


ForDeclaration: LetOrConst ForBinding
1. For each element `name` of the BoundNames of ForBinding do
   a. If IsConstantDeclaration of LetOrConst is true, then
      i. Call `environment`’s CreateImmutableBinding concrete method with argument `name`.
   b. Else
      i. Call `environment`’s CreateMutableBinding concrete method with argument `name`.
      ii. Assert: The above call to CreateMutableBinding will never return an abrupt completion.
2. Return the result of performing BindingInitialization for ForBinding passing `value` and `environment` as the arguments.

13.6.4.7 Runtime Semantics: LabelledEvaluation

With argument `labelSet`. 

**IterationStatement** 
\( \text{for ( LeftHandSideExpression in Expression ) Statement} \)

1. Let \( \text{keyResult} \) be \( \text{ForIn/OfExpressionEvaluation( } \), Expression, enumerate, labelSet\).
2. ReturnIfAbrupt(\( \text{keyResult} \)).
3. Return \( \text{ForIn/OfBodyEvaluation(LeftHandSideExpression, Statement, keyResult, assignment, labelSet)} \).

**IterationStatement** 
\( \text{for ( var ForBinding in Expression ) Statement} \)

1. Let \( \text{keyResult} \) be \( \text{ForIn/OfExpressionEvaluation( } \), Expression, enumerate, labelSet\).
2. ReturnIfAbrupt(\( \text{keyResult} \)).
3. Return \( \text{ForIn/OfBodyEvaluation(ForBinding, Statement, keyResult, varBinding, labelSet)} \).

**IterationStatement** 
\( \text{for ( ForDeclaration in Expression ) Statement} \)

1. Let \( \text{keyResult} \) be the result of performing \( \text{ForIn/OfExpressionEvaluation(BoundNames of ForDeclaration, Expression, enumerate, labelSet)} \).
2. ReturnIfAbrupt(\( \text{keyResult} \)).
3. Return \( \text{ForIn/OfBodyEvaluation(ForDeclaration, Statement, keyResult, lexicalBinding, labelSet)} \).

**IterationStatement** 
\( \text{for ( LeftHandSideExpression of AssignmentExpression ) Statement} \)

1. Let \( \text{keyResult} \) be the result of performing \( \text{ForIn/OfExpressionEvaluation( } \), AssignmentExpression, iterate, labelSet\).
2. ReturnIfAbrupt(\( \text{keyResult} \)).
3. Return \( \text{ForIn/OfBodyEvaluation(LeftHandSideExpression, Statement, keyResult, assignment, labelSet)} \).

**IterationStatement** 
\( \text{for ( var ForBinding of AssignmentExpression ) Statement} \)

1. Let \( \text{keyResult} \) be the result of performing \( \text{ForIn/OfExpressionEvaluation( } \), AssignmentExpression, iterate, labelSet\).
2. ReturnIfAbrupt(\( \text{keyResult} \)).
3. Return \( \text{ForIn/OfBodyEvaluation(ForBinding, Statement, keyResult, varBinding, labelSet)} \).

**IterationStatement** 
\( \text{for ( ForDeclaration of AssignmentExpression ) Statement} \)

1. Let \( \text{keyResult} \) be the result of performing \( \text{ForIn/OfExpressionEvaluation( BoundNames of ForDeclaration, AssignmentExpression, iterate, labelSet)} \).
2. ReturnIfAbrupt(\( \text{keyResult} \)).
3. Return \( \text{ForIn/OfBodyEvaluation(ForDeclaration, Statement, keyResult, lexicalBinding, labelSet)} \).

13.6.4.8 Runtime Semantics: ForIn/OfExpressionEvaluation Abstract Operation

The abstract operation \( \text{ForIn/OfExpressionEvaluation} \) is called with arguments \( \text{TDZnames}, \text{expr}, \text{iterationKind}, \text{and labelSet} \). The value of \text{iterationKind} is either \text{enumerate} or \text{iterate}.

1. Let \( \text{oldEnv} \) be the running execution context’s LexicalEnvironment.
2. If \( \text{TDZnames} \) is not an empty List, then
   a. Assert: \( \text{TDZnames} \) has no duplicate entries.
   b. Let \( \text{TDZ} \) be NewDeclarativeEnvironment(\( \text{oldEnv} \)).
   c. For each string \text{name} in \( \text{TDZnames} \), do
i. Let \( \text{status} \) be the result of calling TDZ’s `CreateMutableBinding` concrete method passing `name` and `false` as the arguments.
ii. Assert: `status` is never an abrupt completion.
iii. Set the running execution context’s LexicalEnvironment to TDZ.
iv. Let `exprRef` be the result of evaluating the production that is `expr`.
v. Set the running execution context’s LexicalEnvironment to `oldEnv`.
vi. Let `exprValue` be `GetValue(exprRef)`.
7. If `exprValue` is an abrupt completion,
a. If `LoopContinues(exprValue, labelSet)` is `false`, then return `exprValue`.
8. If `iterationKind` is `enumerate`, then
   a. If `exprValue.[value]` is `null` or `undefined`, then
      i. Return `Completion{[type]: `break`,`[value]`: empty, [target]: empty}`.
9. Let `obj` be `ToObject(exprValue)`.
10. If `iterationKind` is `enumerate`, then
    a. Let `keys` be the result of calling the `[[Enumerate]]` internal method of `obj` with no arguments.
11. Else,
    a. Assert: `iterationKind` is `iterate`.
    b. Let `keys` be `GetIterator(obj)`.
12. If `keys` is an abrupt completion, then
    a. If `LoopContinues(keys, labelSet)` is `false`, then return `keys`.
    b. Assert: `keys.[type]` is `continue`.
13. Return `keys`.

### 13.6.4.9 Runtime Semantics: `ForIn/OfBodyEvaluation`

The abstract operation `ForIn/OfBodyEvaluation` is called with arguments `lhs`, `stmt`, `iterator`, `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 `IteratorStep(iterator)`.
   b. `ReturnIfAbrupt(nextResult)`.
   c. If `nextResult` is `false`, then return `NormalCompletion(V)`.
   d. Let `nextValue` be `IteratorValue(nextResult)`.
   e. `ReturnIfAbrupt(nextValue)`.
   f. If `lhsKind` is `assignment`, then
      i. Assert: `lhs` is a `LeftHandSideExpression`.
      ii. If `lhs` is neither an `ObjectLiteral` nor an `ArrayLiteral` then
          1. Let `lhsRef` be the result of evaluating `lhs` (it may be evaluated repeatedly).
          2. Let `status` be `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. 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 `BindingInitialization 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 `NewDeclarativeEnvironment(oldEnv)`.
iv. Set the running execution context’s LexicalEnvironment to `iterationEnv`.
v. Let `status` be the result of performing BindingInstantiation for `lhs` passing `nextValue` and `iterationEnv` as arguments.
i. If `status` is an abrupt completion, then
   ii. Set the running execution context’s LexicalEnvironment to `oldEnv`.
   iii. Return `IteratorClose(iterator, status)`.
j. Let `status` be the result of evaluating `stmt`.
k. If `status`.[[type]] is `normal` and `status`.[[value]] is not empty, then
   i. Let `V = status`.[[value]]
   ii. Set the running execution context’s LexicalEnvironment to `oldEnv`.
m. If `LoopContinues(status, labelSet)` is false, then
   i. Return `IteratorClose(iterator, status)`.

13.7 The `continue` Statement

Syntax

```
ContinueStatement[\[Yield\]] :
  continue ;
  continue [\[no LineTerminator here\]] LabelIdentifier[\[Yield\]] ;
```

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 LabelIdentifier ;

• It is a Syntax Error if this production is not nested, directly or indirectly (but not crossing function boundaries), within an `IterationStatement`.
• It is a Syntax Error if `StringValue(LabelIdentifier)` does not appear in the enclosing `IterationLabelSet` of this `ContinueStatement`.
```

13.7.2 Runtime Semantics: Evaluation

```
ContinueStatement : continue ;

1. Return Completion{[[type]]: continue, [[value]]: empty, [[target]]: empty}.
```

```
ContinueStatement : continue LabelIdentifier ;

1. Let `label` be the `StringValue` of `LabelIdentifier`.
2. Return Completion{[[type]]: continue, [[value]]: empty, [[target]]: label }.
```
13.8 The **break** Statement

**Syntax**

\[
\text{BreakStatement} \rightarrow \text{break} ; \\
\text{break} \quad \text{LabelIdentifier} ;
\]

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 **LabelIdentifier** ;

- It is a Syntax Error if **StringValue**(LabelIdentifier) does not appear in the enclosing **CurrentLabelSet** of this **BreakStatement**.

13.8.2 Runtime Semantics: Evaluation

**BreakStatement** : **break** ;

1. Return Completion(\{[type]: break, [value]: empty, [target]: empty\}).

**BreakStatement** : break **LabelIdentifier** ;

1. Let label be the **StringValue** of **LabelIdentifier**.
2. Return Completion(\{[type]: break, [value]: empty, [target]: label\}).

13.9 The **return** Statement

**Syntax**

\[
\text{ReturnStatement} \rightarrow \text{return} ; \\
\text{return} \quad \text{Expression} ;
\]

**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

\[
\text{WithStatement}[\text{Yield, Return}] : \\
\quad \text{with} \ \{ \text{Expression}\} \ \text{Statement}[\text{Yield, Return}]
\]

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

\[\text{WithStatement} : \text{with} \ \{ \text{Expression} \} \ \text{Statement} \]

- It is a Syntax Error if the code that matches this production is contained in strict code.
- It is a Syntax Error if IsLabelledFunction(Statement) is true.

NOTE It is only necessary to apply the second rule if the extension specified in B.3.2 is implemented.

13.10.2 Static Semantics: VarDeclaredNames


\[\text{WithStatement} : \text{with} \ \{ \text{Expression} \} \ \text{Statement} \]

1. Return the VarDeclaredNames of Statement.

13.10.3 Static Semantics: VarScopedDeclarations


\[\text{WithStatement} : \text{with} \ \{ \text{Expression} \} \ \text{Statement} \]

1. Return the VarScopedDeclarations of Statement.

13.10.4 Runtime Semantics: Evaluation

\[\text{WithStatement} : \text{with} \ \{ \text{Expression} \} \ \text{Statement} \]

1. Let \( \text{val} \) be the result of evaluating Expression.
2. Let \( \text{obj} \) be \( \text{ToObject(GetValue(val))} \).
3. ReturnIfAbrupt(obj).
4. Let \( \text{oldEnv} \) be the running execution context’s LexicalEnvironment.
5. Let \( \text{newEnv} \) be \( \text{NewObjectEnvironment(obj, oldEnv)} \).
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

\[
\text{SwitchStatement}::= \text{switch} ( \text{Expression} ) \text{ CaseBlock } \text{ Return} \\
\text{CaseBlock} ::= \text{CaseClauses } \text{ Return} \\
\text{CaseClauses} ::= \{ \text{CaseClause}, \text{ DefaultClause} \} \\
\text{CaseClause} ::= \text{case} \text{ Expression} : \text{StatementList } \text{ Return} \\
\text{DefaultClause} ::= \text{default} : \text{StatementList } \text{ Return} \\
\]

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: LexicallyDeclaredNames


CaseBlock : { }

1. Return a new empty List.

CaseBlock : { CaseClauses, DefaultClause CaseClauses.tail }

1. If the first CaseClauses is present, let names be the LexicallyDeclaredNames of the first CaseClauses.
2. Else let names be a new empty List.
3. Append to names the elements of the LexicallyDeclaredNames of the DefaultClause.
4. If the second CaseClauses is not present, return names.
5. Else return the result of appending to names the elements of the LexicallyDeclaredNames of the second CaseClauses.

CaseClauses : CaseClauses CaseClause

1. Let names be LexicallyDeclaredNames of CaseClauses.
2. Append to names the elements of the LexicallyDeclaredNames of CaseClause.
3. Return names.

CaseClause: case Expression : StatementList
  1. If the StatementList is present, return the LexicallyDeclaredNames of StatementList.
  2. Else return a new empty List.

DefaultClause: default : StatementList
  1. If the StatementList is present, return the LexicallyDeclaredNames of StatementList.
  2. Else return a new empty List.

13.11.3 Static Semantics: LexicallyScopedDeclarations

See also: 13.1.3, 13.12.5, 14.1.15, 14.2.11, 15.1.4, 15.2.0.11.

CaseBlock: { }
  1. Return a new empty List.

CaseBlock: { CaseClauses DefaultClause CaseClauses } }}
  1. If the first CaseClauses is present, let declarations be the LexicallyScopedDeclarations of the first CaseClauses.
  2. Else let declarations be a new empty List.
  3. Append to declarations the elements of the LexicallyScopedDeclarations 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 LexicallyScopedDeclarations of the second CaseClauses.

CaseClauses: CaseClauses CaseClause
  1. Let declarations be LexicallyScopedDeclarations of CaseClauses.
  2. Append to declarations the elements of the LexicallyScopedDeclarations of CaseClause.
  3. Return declarations.

CaseClause: case Expression : StatementList
  1. If the StatementList is present, return the LexicallyScopedDeclarations of StatementList.
  2. Else return a new empty List.

DefaultClause: default : StatementList
  1. If the StatementList is present, return the LexicallyScopedDeclarations of StatementList.
  2. Else return a new empty List.

13.11.4 Static Semantics: VarDeclaredNames


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 : 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.

13.11.5 Static Semantics: VarScopedDeclarations


SwitchStatement : switch ( Expression ) CaseBlock
  1. Return the VarScopedDeclarations of CaseBlock.

CaseBlock : { }
  1. Return a new empty List.

CaseBlock : { CaseClausesopt DefaultClause CaseClausesopt }
  1. If the first CaseClauses is present, let declarations be the VarScopedDeclarations of the first CaseClauses.
  2. Else let declarations be a new empty List.
  3. Append to declarations the elements of the VarScopedDeclarations 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 VarScopedDeclarations of the second CaseClauses.

CaseClauses : CaseClauses CaseClause
  1. Let declarations be VarScopedDeclarations of CaseClauses.
  2. Append to declarations the elements of the VarScopedDeclarations of CaseClause.
3. Return declarations.

CaseClause : case Expression : StatementListopt
   1. If the StatementList is present, return the VarScopedDeclarations of StatementList.
   2. Else return a new empty List.

DefaultClause : default : StatementListopt
   1. If the StatementList is present, return the VarScopedDeclarations of StatementList.
   2. Else return a new empty List.

13.11.6 Runtime Semantics: CaseBlockEvaluation

With argument input.

CaseBlock : { }
   1. Return NormalCompletion(undefined).

CaseBlock : { CaseClauses }
   1. Let V = undefined.
   2. Let A be the List of CaseClause items in CaseClauses, in source text order.
   3. Let searching be true.
   4. Repeat, for each CaseClause, C, in A
      a. If searching is true, then
         i. Let clauseSelector be the result of CaseSelectorEvaluation of C.
         ii. ReturnIfAbrupt(clauseSelector).
         iii. Let matched be the result of performing Strict Equality Comparison input === clauseSelector.
         iv. If matched is true, then
            1. Set searching to false.
            2. If C has a StatementList, then
               a. Let V be the result of evaluating C’s StatementList.
               b. ReturnIfAbrupt(V).
      b. Else searching is false,
         i. If C has a StatementList, then
            1. Let R be the result of evaluating C’s StatementList.
            2. If R.[[value]] is not empty, then let V = R.[[value]].
            3. If R is an abrupt completion, then return Completion{[[type]]: R.[[type]], [[value]]: V, [[target]]: R.[[target]]}.
         5. Return NormalCompletion(V).

CaseBlock : { CaseClauses DefaultClause CaseClausesopt }
   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 \( \text{input} \equiv \text{clauseSelector} \).
   
   b. If found is \textbf{true}, then
      
      i. Let \( R \) be the result of evaluating \textit{CaseClause} \( C \).
      
      ii. If \( R.([\text{value}]) \) is not \texttt{empty}, then let \( V = R.([\text{value}]) \).
      
      iii. If \( R \) is an abrupt completion, then return Completion\{\[\text{type}\]: \( R.([\text{type}]) \), \[\text{value}\]: \( V \), \[\text{target}\]: \( R.([\text{target}]) \}\}.
   
   5. Let \textit{foundInB} be \textbf{false}.
   
   6. If found is \textbf{false}, then
      
      a. Let \( B \) be a new List containing the \textit{CaseClause} items in the second \textit{CaseClauses}, in source text order.
      
      b. Repeat, letting \( C \) be in order each \textit{CaseClause} in \( B \)
         
         i. If \textit{foundInB} is \textbf{false}, then
            
            1. Let clauseSelector be the result of CaseSelectorEvaluation of \( C \).
            
            2. If clauseSelector is an abrupt completion, then
               
               a. If clauseSelector.\[\text{value}\] is \texttt{empty}, then return Completion\{\[\text{type}\]: clauseSelector.\[\text{type}\], \[\text{value}\]: \texttt{undefined}, \[\text{target}\]: clauseSelector.\[\text{target}\]\}\}.
               
               b. Else, return clauseSelector.
            
            3. Let \textit{foundInB} be the result of performing Strict Equality Comparison \( \text{input} \equiv \text{clauseSelector} \).
         
         ii. If \textit{foundInB} is \textbf{true}, then
            
            1. Let \( R \) be the result of evaluating \textit{CaseClause} \( C \).
            
            2. If \( R.([\text{value}]) \) is not \texttt{empty}, then let \( V = R.([\text{value}]) \).
            
            3. If \( R \) is an abrupt completion, then return Completion\{\[\text{type}\]: \( R.([\text{type}]) \), \[\text{value}\]: \( V \), \[\text{target}\]: \( R.([\text{target}]) \}\}.
      
      7. If \textit{foundInB} is \textbf{true}, then return NormalCompletion\( (V) \).
      
      8. Let \( R \) be the result of evaluating DefaultClause.
      
      9. If \( R.([\text{value}]) \) is not \texttt{empty}, then let \( V = R.([\text{value}]) \).
      
      10. If \( R \) is an abrupt completion, then return Completion\{\[\text{type}\]: \( R.([\text{type}]) \), \[\text{value}\]: \( V \), \[\text{target}\]: \( R.([\text{target}]) \}\}.
   
      11. Let \( B \) be a new List containing the \textit{CaseClause} items in the second \textit{CaseClauses}, in source text order.
      
      12. Repeat, letting \( C \) be in order each \textit{CaseClause} in \( B \) (NOTE this is another complete iteration of the second CaseClauses)
         
         a. Let \( R \) be the result of evaluating \textit{CaseClause} \( C \).
         
         b. If \( R.([\text{value}]) \) is not \texttt{empty}, then let \( V = R.([\text{value}]) \).
         
         c. If \( R \) is an abrupt completion, then return Completion\{\[\text{type}\]: \( R.([\text{type}]) \), \[\text{value}\]: \( V \), \[\text{target}\]: \( R.([\text{target}]) \}\}.
      
      13. Return NormalCompletion\( (V) \).

### 13.11.7 Runtime Semantics: CaseSelectorEvaluation

\textbf{CaseClause} : \textbf{case} Expression : StatementList\opt

1. Let exprRef be the result of evaluating Expression.
2. Return GetValue(exprRef).

\textbf{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.
13.11.8 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 NewDeclarativeEnvironment(oldEnv).
6. Perform BlockDeclarationInstantiation(CaseBlock, 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 :

1. Return NormalCompletion(empty).

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[Valid, Return] :

LabelledItem[Valid, Return] :
    Statement[Valid, ?Return]
    FunctionDeclaration[Valid, ?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.
13.12.1 Static Semantics: Early Errors

LabelledStatement : LabelIdentifier : LabelledItem

- It is a Syntax Error if the immediately enclosing CurrentLabelSet contains the StringValue of LabelIdentifier.

LabelledItem : FunctionDeclaration

- It is a Syntax Error if any source code matches this rule.

NOTE An alternative definition for this rule is provided in B.3.2.

13.12.2 Static Semantics: CurrentLabelSet

LabelledStatement : LabelIdentifier : LabelledItem

1. The CurrentLabelSet of this LabelledStatement is a List that includes the StringValue of LabelIdentifier and all elements of the immediately enclosing CurrentLabelSet.

13.12.3 Static Semantics: IsLabelledFunction ( stmt )

The abstract operation IsLabelledFunction with argument stmt performs the following steps:

1. If stmt is not a LabelledStatement, then return false.
2. Let item be the LabelledItem component of stmt.
3. If item is LabelledItem : FunctionDeclaration, then return true.
4. Let subStmt be the Statement component of item.
5. Return IsLabelledFunction(subStmt).

13.12.4 Static Semantics: LexicallyDeclaredNames

See also: 13.1.2, 13.11.2, 14.1.14, 14.2.10, 15.1.3, 15.2.0.10.

LabelledStatement : LabelIdentifier : LabelledItem

1. Return the LexicallyDeclaredNames of LabelledItem.

LabelledItem : Statement

1. Return a new empty List.

LabelledItem : FunctionDeclaration

1. Return BoundNames of FunctionDeclaration.

13.12.5 Static Semantics: LexicallyScopedDeclarations

See also: 13.1.3, 13.11.3, 14.1.15, 14.2.11, 15.1.4, 15.2.0.11.

LabelledStatement : LabelIdentifier : LabelledItem

1. Return the LexicallyScopedDeclarations of LabelledItem.
LabelledItem : Statement
    1. Return a new empty List.

LabelledItem : FunctionDeclaration
    1. Return a new List containing FunctionDeclaration.

13.12.6  Static Semantics: TopLevelLexicallyDeclaredNames

See also: 13.1.4.

LabelledStatement : LabelIdentifier : LabelledItem
    1. Return a new empty List.

13.12.7  Static Semantics: TopLevelLexicallyScopedDeclarations

See also: 13.1.5.

LabelledStatement : LabelIdentifier : LabelledItem
    1. Return a new empty List.

13.12.8  Static Semantics: TopLevelVarDeclaredNames

See also: 13.1.6.

LabelledStatement : LabelIdentifier : LabelledItem
    1. Return the TopLevelVarDeclaredNames of LabelledItem.

LabelledItem : Statement
    1. If Statement is Statement : LabelledStatement, then TopLevelVarDeclaredNames of Statement.
    2. Return VarDeclaredNames of Statement.

LabelledItem : FunctionDeclaration
    1. Return BoundNames of FunctionDeclaration.

13.12.9  Static Semantics: TopLevelVarScopedDeclarations

See also: 13.1.7.

LabelledStatement : LabelIdentifier : LabelledItem
    1. Return the TopLevelVarScopedDeclarations of LabelledItem.

LabelledItem : Statement
    1. If Statement is Statement : LabelledStatement, then TopLevelVarScopedDeclarations of Statement.
    2. Return VarScopedDeclarations of Statement.

LabelledItem : FunctionDeclaration
    1. Return a new List containing FunctionDeclaration.
13.12.10 Static Semantics: VarDeclaredNames


LabelledStatement : LabelIdentifier : LabelledItem
  1. Return the VarDeclaredNames of LabelledItem.

LabelledItem : FunctionDeclaration
  1. Return a new empty List.

13.12.11 Static Semantics: VarScopedDeclarations


LabelledStatement : LabelIdentifier : LabelledItem
  1. Return the VarScopedDeclarations of LabelledItem.

LabelledItem : FunctionDeclaration
  1. Return a new empty List.


With argument labelSet.

See also: 13.0.3, 13.6.1.2, 13.6.2.2, 13.6.3.3, 13.6.4.7.

LabelledStatement : LabelIdentifier : LabelledItem
  1. Let label be the StringValue of LabelIdentifier.
  2. Let newLabelSet be a new List containing label and the elements of labelSet.
  3. Let stmtResult be the result of performing LabelledEvaluation of LabelledItem with argument newLabelSet.
  4. If stmtResult.[[type]] is break and SameValue(stmtResult.[[target]], label), then
     a. Return NormalCompletion(stmtResult.[[value]]).
  5. Return stmtResult.

LabelledItem : Statement
  1. If Statement is either a LabelledStatement or a BreakableStatement, then
     a. Return the result of performing LabelledEvaluation of Statement with argument labelSet.
  2. Else,
     a. Return the result of evaluating Statement.

LabelledItem: FunctionDeclaration
  1. Return the result of evaluating FunctionDeclaration.
13.12.12.1 Runtime Semantics: Evaluation

LabelledStatement : LabelIdentifier : LabelledItem
  1. Let newLabelSet be a new empty List.
  2. Return the result of performing LabelledEvaluation of LabelledItem with argument newLabelSet.

13.13 The throw Statement

Syntax

ThrowStatementSyntax : 
  throw [no LineTerminator here] ExpressionSyntax ;

13.13.1 Runtime Semantics: Evaluation

ThrowStatement : throw Expression ;
  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

TryStatementSyntax : 

CatchSyntax : 
  catch ( CatchParameterSyntax ) BlockSyntax

FinallySyntax : 
  finally BlockSyntax

CatchParameterSyntax : 
  BindingIdentifierSyntax ?Yield
  BindingPatternSyntax

NOTE The try statement encloses a block of code in which an exceptional condition can occur, such as a runtime error or a throw statement. The catch clause provides the exception-handling code. When a catch clause catches an exception, its 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.
NOTE An alternative static semantics for this production is given in B.3.5.

13.14.2 Static Semantics: VarDeclaredNames

See also: 13.0.1, 13.1.8, 13.2.2.2, 13.5.2, 13.6.1.1, 13.6.2.2, 13.6.3.1, 13.6.4.3, 13.10.2, 13.11.4, 13.12.10, 14.1.17, 14.2.13, 15.1.5, 15.2.0.13.

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 Static Semantics: VarScopedDeclarations

See also: 13.0.3, 13.1.9, 13.2.2.3, 13.5.3, 13.6.1.2, 13.6.2.2, 13.6.3.3, 13.6.4.4, 13.10.3, 13.11.5, 13.12.11, 14.1.18, 14.2.14, 15.1.6, 15.2.0.14.

TryStatement : try Block Catch
  1. Let declarations be VarScopedDeclarations of Block.
  2. Append to declarations the elements of the VarScopedDeclarations of Catch.
  3. Return declarations.

TryStatement : try Block Finally
  1. Let declarations be VarScopedDeclarations of Block.
  2. Append to declarations the elements of the VarScopedDeclarations of Finally.
  3. Return declarations.

TryStatement : try Block Catch Finally
  1. Let declarations be VarScopedDeclarations of Block.
  2. Append to declarations the elements of the VarScopedDeclarations of Catch.
  3. Append to declarations the elements of the VarScopedDeclarations of Finally.
  4. Return declarations.
Catch: \texttt{catch} (CatchParameter) Block

1. Return the VarScopedDeclarations of Block.

13.14.4 Runtime Semantics: BindingInitialization

With arguments value and environment.

NOTE \texttt{undefined} is passed for environment to indicate that a PutValue operation should be used to assign the initialization value. This is the case for \texttt{var} statements and the formal parameter lists of some non-strict functions (see 9.2.13). In those cases a lexical binding is hoisted and preinitialized prior to evaluation of its initializer.

See also: 12.1.4, 13.2.2.4, 13.6.4.5, 13.2.3.5.

CatchParameter: BindingPattern

1. If Type(value) is not Object, then throw a \texttt{TypeError} exception.
2. Return the result of performing BindingInitialization for BindingPattern passing value and environment as the arguments.

13.14.5 Runtime Semantics: CatchClauseEvaluation

with parameter thrownValue

Catch: \texttt{catch} (CatchParameter) Block

1. Let oldEnv be the running execution context’s LexicalEnvironment.
2. Let catchEnv be NewDeclarativeEnvironment(oldEnv).
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. Set the running execution context’s LexicalEnvironment to catchEnv.
5. Let status be the result of performing BindingInitialization for CatchParameter passing thrownValue and catchEnv as arguments.
6. If status is an abrupt completion, then
   a. Set the running execution context’s LexicalEnvironment to oldEnv.
   b. Return status.
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.

13.14.6 Runtime Semantics: Evaluation

TryStatement: \texttt{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: \texttt{try} Block Finally

1. Let B be the result of evaluating Block.
2. Let $F$ be the result of evaluating $\text{Finally}$.
3. If $F$.[[type]] is normal, return $B$.
4. Return $F$.

TryStatement : try Block Catch Finally
1. Let $B$ be the result of evaluating $\text{Block}$.
2. If $B$.[[type]] is throw, then
   a. Let $C$ be the result of performing CatchClauseEvaluation of $\text{Catch}$ with parameter $B$.[[value]].
3. Else $B$.[[type]] is not throw,
   a. Let $C$ be $B$.
4. Let $F$ be the result of evaluating $\text{Finally}$.
5. If $F$.[[type]] is normal, return $C$.

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.

DebuggerStatement : debugger ;
1. If an implementation defined debugging facility is available and enabled, then
   a. Perform an implementation defined debugging action.
   b. Let $result$ be an implementation defined Completion value.
2. Else
   a. Let $result$ be NormalCompletion(empty).
3. Return $result$.

14 ECMAScript Language: Functions and Classes

NOTE Various ECMAScript language elements cause the creation of ECMAScript function objects (9.1.14). Evaluation of such functions starts with the execution of their [[Call]] internal method (9.2.2).

14.1 Function Definitions

Syntax
FunctionDeclaration[Yield, Default] :
  function BindingIdentifier[Yield, ?Default] ( FormalParameters ) { FunctionBody }

FunctionExpression :
  function BindingIdentifieropt ( FormalParameters ) { FunctionBody }

StrictFormalParameters[Yield, GeneratorParameter] :
  FormalParameters[Yield, ?GeneratorParameter]
A Directive Prologue is the longest sequence of ExpressionStatement productions occurring as the initial StatementListItem productions of a FunctionBody or a ScriptBody 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 code unit 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 and which does not have a meaning defined by the implementation.


14.1.2 Static Semantics: Early Errors

FunctionDeclaration : function BindingIdentifier ( FormalParameters ) { FunctionBody }
and
FunctionExpression : function BindingIdentifieropt ( FormalParameters ) { FunctionBody }
• If the source code matching this production is strict code, the Early Error rules for
  \textit{StrictFormalParameters} : \textit{FormalParameters} are applied.
• If the source code matching this production is strict code, it is a Syntax Error if \textit{BindingIdentifier} is
  the IdentifierName \texttt{eval} or the IdentifierName \texttt{arguments}.
• It is a Syntax Error if any element of the BoundNames of \textit{FormalParameters} also occurs in the
  LexicallyDeclaredNames of \textit{FunctionBody}.

\textbf{NOTE} \hspace{1em} The LexicallyDeclaredNames of a \textit{FunctionBody} does not include identifiers bound using \texttt{var} or function
declarations.

\textit{StrictFormalParameters} : \textit{FormalParameters}

• It is a Syntax Error if BoundNames of \textit{FormalParameters} contains any duplicate elements.

\textit{FormalParameters} : \textit{FormalParameterList}

• It is a Syntax Error if IsSimpleParameterList of \textit{FormalParameterList} is \texttt{false} and BoundNames of
  \textit{FormalParameterList} contains any duplicate elements.

\textbf{NOTE} \hspace{1em} Multiple occurrences of the same BindingIdentifier in a
\textit{FormalParameterList} is only allowed for non-strict
functions and generator functions that have simple parameter lists.

\textit{FunctionBody} : \textit{FunctionStatementList}

• It is a Syntax Error if the LexicallyDeclaredNames of \textit{FunctionStatementList} contains any duplicate
  entries.
• It is a Syntax Error if any element of the LexicallyDeclaredNames of \textit{FunctionStatementList} also
  occurs in the VarDeclaredNames of \textit{FunctionStatementList}.

14.1.3 Static Semantics: Bind\textit{Names}

See also: 13.2.1.2, 13.2.2.1, 12.1.2, 13.6.4.2, 14.2.2, 14.4.2, 14.5.2, 15.2.1.2, 15.2.2.1.

\textit{FunctionDeclaration} : \texttt{function} Binding\textit{Identifier} \hspace{0.5em} ( \textit{FormalParameters} ) \hspace{0.5em} \{ \textit{FunctionBody} \}

1. \hspace{0.5em} Return the BoundNames of Binding\textit{Identifier}.

\textit{FormalParameters} : [empty]

1. \hspace{0.5em} Return an empty List.

\textit{FormalParameterList} : \textit{FormalsList} \hspace{0.5em} , \hspace{0.5em} \textit{FunctionRestParameter}

1. \hspace{0.5em} Let \textit{names} be BoundNames of \textit{FormalsList}.
2. \hspace{0.5em} Append to \textit{names} the BoundNames of \textit{FunctionRestParameter}.
3. \hspace{0.5em} Return \textit{names}.

\textit{FormalsList} : \textit{FormalsList} \hspace{0.5em} , \hspace{0.5em} \textit{FormalParameter}

1. \hspace{0.5em} Let \textit{names} be BoundNames of \textit{FormalsList}.
2. \hspace{0.5em} Append to \textit{names} the elements of BoundNames of \textit{FormalParameter}.
3. \hspace{0.5em} Return \textit{names}.
14.1.4 Static Semantics: Contains

With parameter symbol.

See also: 5.3, 12.2.5.2, 12.3.1.1, 14.2.3, 14.4.3, 14.5.4

FunctionDeclaration : function BindingIdentifier ( FormalParameters ) { FunctionBody }
   1. Return false.

FunctionExpression : function BindingIdentifieropt ( FormalParameters ) { FunctionBody }
   1. Return false.

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

14.1.5 Static Semantics: ContainsExpression

See also: 13.2.3.2, 14.2.4.

FormalParameters : [empty]
   1. Return false.

FormalParameterList : FunctionRestParameter
   1. Return false.

FormalParameterList : FormalsList , FunctionRestParameter
   1. Return ContainsExpression of FormalsList.

FormalsList : FormalsList , FormalParameter
   1. If ContainsExpression of FormalsList is true, then return true.
   2. Return ContainsExpression of FormalParameter.

14.1.6 Static Semantics: ExpectedArgumentCount

See also: 14.2.6, 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 initializer. A FormalParameter without an initializer is allowed after the first parameter with an initializer but such parameters are considered to be optional with undefined as their default value.
FormalsList : FormalParameter
  1. If HasInitializer of FormalParameter is true return 0
  2. Return 1.

FormalsList : FormalsList , FormalParameter
  1. Let count be the ExpectedArgumentCount of FormalsList.
  2. If HasInitializer of FormalsList is true or HasInitializer of FormalParameter is true, then return count.

14.1.7 Static Semantics: HasInitializer

See also: 13.2.3.3, 14.2.7.

FormalParameters : [empty]
  1. Return false.

FormalParameterList : FunctionRestParameter
  1. Return false.

FormalParameterList : FormalsList , FunctionRestParameter
  1. If HasInitializer of FormalsList is true, then return true.
  2. Return false.

FormalsList : FormalsList , FormalParameter
  1. If HasInitializer of FormalsList is true, then return true.
  2. Return HasInitializer of FormalParameter.

14.1.8 Static Semantics: HasName

See also: 14.2.8, 14.4.6, 14.5.6.

FunctionExpression : function ( FormalParameters ) { FunctionBody }
  1. Return false.

FunctionExpression : function. BindingIdentifier ( FormalParameters ) { FunctionBody }
  1. Return true.

14.1.9 Static Semantics: IsAnonymousFunctionDefinition ( production) Abstract Operation

The abstract operation IsAnonymousFunctionDefinition determines if its argument is a function definition that does not bind a name. The argument production is the result of parsing an AssignmentExpression or Initializer. The following steps are taken:

  1. If IsFunctionDefinition of production is false, then return false.
  2. Let hasName be the result of HasName of production.
  3. If hasName is true, then return false.
  4. Return true.
14.1.10 Static Semantics: IsConstantDeclaration

See also: 13.2.1.3, 14.4.5, 14.5.5.

\[
\text{FunctionDeclaration} : \text{function} \ BindingIdentifier \ (\ FormalParameters \ ) \ {\ FunctionBody } \\
1. \ Return \ false.
\]

14.1.11 Static Semantics: IsFunctionDefinition

See also: 12.2.0.2, 12.2.9.2, 12.3.1.2, 12.4.2, 12.5.2, 12.6.1, 12.7.1, 12.8.1, 12.9.1, 12.10.1, 12.11.1, 12.12.1, 12.13.1, 12.14.2, 12.15.1, 14.4.8, 14.5.8.

\[
\text{FunctionExpression} : \text{function} \ (\ FormalParameters \ ) \ {\ FunctionBody } \\
1. \ Return \ true.
\]

\[
\text{FunctionExpression} : \text{function} \ BindingIdentifier \ (\ FormalParameters \ ) \ {\ FunctionBody } \\
1. \ Return \ true.
\]

14.1.12 Static Semantics: IsSimpleParameterList

See also: 13.2.3.4, 14.2.8

\[
\text{FormalParameters} : [\emptyset] \\
1. \ Return \ true.
\]

\[
\text{FormalParameterList} : \text{FunctionRestParameter} \\
1. \ Return \ false.
\]

\[
\text{FormalParameterList} : \text{FormalsList} , \ \text{FunctionRestParameter} \\
1. \ Return \ false.
\]

\[
\text{FormalsList} : \text{FormalsList} , \ \text{FormalParameter} \\
1. \ If \ \text{IsSimpleParameterList} \ \text{of} \ \text{FormalsList} \ \text{is} \ \text{false}, \ \text{return} \ \text{false}. \ \\
2. \ Return \ \text{IsSimpleParameterList} \ \text{of} \ \text{FormalParameter}.
\]

\[
\text{FormalParameter} : \text{BindingElement} \\
1. \ Return \ \text{IsSimpleParameterList} \ \text{of} \ \text{BindingElement}.
\]

14.1.13 Static Semantics: IsStrict

See also: 15.1.2, 15.2.0.7.

\[
\text{FunctionStatementList} : \text{StatementList}_opt \\
1. \ If \ this \ \text{FunctionStatementList} \ \text{is} \ \text{contained} \ \text{in} \ \text{strict} \ \text{code} \ \text{or} \ \text{if} \ \text{StatementList}_opt \ \text{is} \ \text{strict} \ \text{code}, \ \text{then} \ \text{return} \ true. \ \text{Otherwise}, \ \text{return} \ false.
\]

Commented [AWB1053]: Need a better definition
14.1.14 Static Semantics: LexicallyDeclaredNames

See also: 13.1.2, 13.11.2, 13.12.4, 14.2.10, 15.1.3, 15.2.0.10.

FunctionStatementList : [empty]
  1. Return an empty List.

FunctionStatementList : StatementList
  1. Return TopLevelLexicallyDeclaredNames of StatementList.

14.1.15 Static Semantics: LexicallyScopedDeclarations

See also: 13.1.3, 13.11.3, 13.12.5, 14.2.11, 15.1.4, 15.2.0.11.

FunctionStatementList : [empty]
  1. Return an empty List.

FunctionStatementList : StatementList
  1. Return the TopLevelLexicallyScopedDeclarations of StatementList.

14.1.16 Static Semantics: ReferencesSuper

See also: 14.2.12, 14.3.6, 14.4.10.

FunctionDeclaration : function BindingIdentifier ( FormalParameters ) { FunctionBody }
  1. If FormalParameters Contains super is true, then return true.
  2. Return FunctionBody Contains super.

FunctionExpression : function BindingIdentifieropt ( FormalParameters ) { FunctionBody }
  1. If FormalParameters Contains super is true, then return true.
  2. Return FunctionBody Contains super.

FormalParameters : [empty]
  1. Return false.

FormalParameters : FormalParameterList
  1. Return FormalParameterList Contains super.

FunctionBody : FunctionStatementList
  1. Return FunctionStatementList Contains super.

14.1.17 Static Semantics: VarDeclaredNames

FunctionStatementList : [empty]
  1. Return an empty List.

FunctionStatementList : StatementList
  1. Return TopLevelVarDeclaredNames of StatementList.

14.1.18 Static Semantics: VarScopedDeclarations

FunctionStatementList : [empty]
  1. Return an empty List.

FunctionStatementList : StatementList
  1. Return the TopLevelVarScopedDeclarations of StatementList.

14.1.19 Runtime Semantics: EvaluateBody
With parameter functionObject.
See also: 14.2.16, 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 (14.1.1) 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).

14.1.20 Runtime Semantics: IteratorBindingInitialization
With parameters iterator and environment.

NOTE When undefined is passed for environment it indicates that a PutValue operation should be used to assign the initialization 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: 13.2.3.6, 14.2.15.

FormalParameters : [empty]
  1. Return NormalCompletion(empty).
FormalParameterList : FormalsList , FunctionRestParameter

1. Let restIndex be the result of performing IteratorBindingInitialization for FormalsList using iterator and environment as the arguments.
2. ReturnIfAbrupt(restIndex).
3. Return the result of performing IteratorBindingInitialization for FunctionRestParameter using iterator and environment as the arguments.

FormalsList : FormalsList , FormalParameter

1. Let status be the result of performing IteratorBindingInitialization for FormalsList using iterator and environment as the arguments.
2. ReturnIfAbrupt(status).
3. Return the result of performing IteratorBindingInitialization for FormalParameter using iterator and environment as the arguments.

14.1.21 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.
4. If ReferencesSuper of FunctionDeclaration is true, then
   a. Perform MakeMethod(F, name, undefined).
5. Perform MakeConstructor(F).
6. SetFunctionName(F, name).
7. Assert: SetFunctionName will not return an abrupt completion.
8. Return F.

14.1.22 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.
4. If ReferencesSuper of FunctionExpression is true, then
   a. Perform MakeMethod(closure, undefined, undefined).
5. Perform MakeConstructor(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 `runningContext` be the running execution context’s Lexical Environment.
3. Let `funcEnv` be the `runningContext`’s environment record.
4. Let `name` be `StringValue` of `BindingIdentifier`.
5. Call the `CreateImmutableBinding` concrete method of `envRec` passing `name` as the argument.
7. If `ReferencesSuper` of `FunctionExpression` is `true`, then
   a. `Perform MakeMethod(closure, name, undefined)`.
9. `Perform MakeConstructor(closure)`.
10. `SetFunctionName(closure, name)`.
11. `Call the InitializeBinding concrete method of envRec` passing `name` and `closure` as the arguments.
13. `Return NormalCompletion()`.

**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**

```
ArrowFunction[?Yield] :
    ArrowParameters[?Yield] [no LineTerminator here] => ConciseBody[?Yield]
```

```
ArrowParameters[?Yield] :
    BindingIdentifier[?Yield]
    CoverParenthesizedExpressionAndArrowParameterList[?Yield]

ConciseBody[?Yield] :
    [lookahead ≠ {]} AssignmentExpression[?Yield]
    [ FunctionBody ]
```

**Supplemental Syntax**

When the production

```
```

is recognized the following grammar is used to refine the interpretation of `CoverParenthesizedExpressionAndArrowParameterList`:

```
ArrowFormalParameters[?Yield, ?GeneratorParameter] :
    ( StrictFormalParameters[?Yield, ?GeneratorParameter] )
```
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 LexicallyDeclaredNames of ConciseBody.

ArrowParameters[Yield] : CoverParenthesizedExpressionAndArrowParameterList[Yield]

- If the [Yield] grammar parameter is present on ArrowParameters, it is a Syntax Error if the lexical token sequence matched by CoverParenthesizedExpressionAndArrowParameterList[Yield] cannot be parsed with no tokens left over using ArrowFormalParameters[Yield, GeneratorParameter] as the goal symbol.
- If the [Yield] grammar parameter is not present on ArrowParameters, it is a Syntax Error if the lexical token sequence matched by CoverParenthesizedExpressionAndArrowParameterList[Yield] cannot be parsed with no tokens left over using ArrowFormalParameters as the goal symbol.
- All early errors rules for ArrowFormalParameters and its derived productions also apply to CoveredFormalsList of CoverParenthesizedExpressionAndArrowParameterList[Yield].

14.2.2 Static Semantics: BoundNames

See also: 13.2.1.2, 13.2.2.1, 12.1.2, 13.6.4.2, 14.1.3, 14.4.2, 14.5.2, 15.2.1.2, 15.2.2.1.

ArrowParameters[Yield] : CoverParenthesizedExpressionAndArrowParameterList[Yield]

1. Let formals be CoveredFormalsList of CoverParenthesizedExpressionAndArrowParameterList[Yield].
2. Return the BoundNames of formals.

14.2.3 Static Semantics: Contains

With parameter symbol.

See also: 5.3, 12.2.5.2, 12.3.1.1, 14.1.4, 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[Yield] : CoverParenthesizedExpressionAndArrowParameterList[Yield]

1. Let forms be CoveredFormalsList of CoverParenthesizedExpressionAndArrowParameterList[Yield].
2. Return forms Contains symbol.

14.2.4 Static Semantics: ContainsExpression

See also: 13.2.3.2, 14.1.5.
ArrowParameters : BindingIdentifier
1. Return false.

ArrowParameters[Yield] : CoverParenthesizedExpressionAndArrowParameterList[Yield]
1. Let formals be CoveredFormalsList of
   CoverParenthesizedExpressionAndArrowParameterList[Yield];
2. Return the ContainsExpression of formals.

14.2.5 Static Semantics: CoveredFormalsList

ArrowParameters : BindingIdentifier
1. Return BindingIdentifier.

CoverParenthesizedExpressionAndArrowParameterList[Yield]:
( Expression )
( )
( . . . BindingIdentifier )
( Expression , . . . BindingIdentifier )
1. If the [Yield] grammar parameter is present for CoverParenthesizedExpressionAndArrowParameterList[Yield] return the result of parsing the lexical token stream matched by CoverParenthesizedExpressionAndArrowParameterList[Yield] using ArrowFormalParameters[Yield, GeneratorParameter] as the goal symbol.
2. If the [Yield] grammar parameter is not present for CoverParenthesizedExpressionAndArrowParameterList[Yield] return the result of parsing the lexical token stream matched by CoverParenthesizedExpressionAndArrowParameterList[Yield] using ArrowFormalParameters as the goal symbol.

14.2.6 Static Semantics: ExpectedArgumentCount

See also: 14.1.5, 14.3.2.

ArrowParameters : BindingIdentifier
1. Return 1.

ArrowParameters[Yield] : CoverParenthesizedExpressionAndArrowParameterList[Yield]
1. Let formals be CoveredFormalsList of CoverParenthesizedExpressionAndArrowParameterList[Yield];
2. Return the ExpectedArgumentCount of formals.

14.2.7 Static Semantics: HasInitializer

See also: 13.2.3.3, 14.1.7.

ArrowParameters : BindingIdentifier
1. Return false.

ArrowParameters : CoverParenthesizedExpressionAndArrowParameterList
1. Let formals be CoveredFormalsList of CoverParenthesizedExpressionAndArrowParameterList.
2. Return the HasInitializer of `formals`.

**14.2.8 Static Semantics: HasName**

See also: 14.1.8, 14.4.6, 14.5.6.

*ArrowFunction : ArrowParameters => ConciseBody*

1. Return `false`.

**14.2.9 Static Semantics: IsSimpleParameterList**

See also: 13.2.3.4, 14.1.11.

*ArrowParameters : BindingIdentifier*

1. Return `true`.

*ArrowParameters_{prev} : CoverParenthesizedExpressionAndArrowParameterList_{prev}*

1. Let `formals` be `CoveredFormalsList` of `CoverParenthesizedExpressionAndArrowParameterList_{prev}`.
2. Return the `IsSimpleParameterList` of `formals`.

**14.2.10 Static Semantics: LexicallyDeclaredNames**


*ConciseBody : AssignmentExpression*

1. Return an empty List.

**14.2.11 Static Semantics: LexicallyScopedDeclarations**

See also: 13.1.3, 13.11.3, 13.12.5, 14.1.15, 15.1.4, 15.2.0.11.

*ConciseBody : AssignmentExpression*

1. Return an empty List.

**14.2.12 Static Semantics: ReferencesSuper**

See also: 14.1.16, 14.3.6, 14.4.10.

*ArrowFunction : ArrowParameters => ConciseBody*

1. Return `false`.

**NOTE** ReferencesSuper is used to determine whether a function requires its own super bindings. This is never the case for Arrow Functions.

**14.2.13 Static Semantics: VarDeclaredNames**

ConciseBody : AssignmentExpression
  1. Return an empty List.

14.2.14 Static Semantics: VarScopedDeclarations


ConciseBody : AssignmentExpression
  1. Return an empty List.

14.2.15 Runtime Semantics: IteratorBindingInitialization

With parameters iterator and environment.

See also: 13.2.3.6, 14.1.20.

NOTE When undefined is passed for environment it indicates that a PutValue operation should be used to assign the initialization value. This is the case for formal parameter lists of nonstrict functions. In that case the formal parameter bindings are preinitialized in order to deal with the possibility of multiple parameters with the same name.

ArrowParameters : BindingIdentifier
  1. Let next be 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 BindingInitialization for BindingIdentifier using v and environment as the arguments.

ArrowParameters\[Yield\] : CoverParenthesizedExpressionAndArrowParameterList\[Yield\]

  1. Let formals be CoveredFormalsList of CoverParenthesizedExpressionAndArrowParameterList\[Yield\].
  2. Return the result of performing IteratorBindingInitialization of formals with arguments iterator and environment.

14.2.16 Runtime Semantics: EvaluateBody

With parameter functionObject.

See also: 14.1.19, 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. Return `IfAbrupt(exprValue)`.
6. Return `NormalCompletion(exprValue)`.

NOTE In the absence of extensions to this specification, the test in step 4 will never be `true`.

14.2.17 Runtime Semantics: Evaluation

`ArrowFunction_Yield : ArrowParameters_Yield => 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_Yield`.
4. Let `closure` be `FunctionCreate(Arrow, parameters, ConciseBody, scope, strict)`.
5. Return `closure`.

NOTE Any reference to `arguments`, `super`, or `this` within an `ArrowFunction` are resolved to their bindings in the lexically enclosing function. Even though an `ArrowFunction` may contain references to `super`, the function object created in step 4 is not made into a method by performing `MakeMethod`. 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_Yield : PropertyName_Yield { StrictFormalParameters } { FunctionBody }``

`GeneratorMethod_Yield : PropertyName_Yield { } { FunctionBody }``

`set PropertyName_Yield { PropertySetParameterList } { FunctionBody }``

`PropertySetParameterList : FormalParameter`

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 LexicallyDeclaredNames of `FunctionBody`.

`MethodDefinition : set PropertyName { PropertySetParameterList } { FunctionBody }``

- 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: ComputedPropertyContains

With parameter `symbol`. 
See also: 12.2.5.2, 14.4.3, 14.5.5.

MethodDefinition:

```
Property Name (StrictFormalParameters) { FunctionBody }
get PropertyName () { FunctionBody }
set PropertyName (PropertySetParameterList) { FunctionBody }
```

1. Return the result of ComputedPropertyContains for `PropertyName` with argument `symbol`.

### 14.3.3 Static Semantics: ExpectedArgumentCount

See also: 14.1.5, 14.2.6.

PropertySetParameterList : FormalParameter

1. If HasInitializer of FormalParameter is `true` return 0
2. Return 1.

### 14.3.4 Static Semantics: HasComputedPropertyKey

See also: 12.2.5.4, 14.4.5

MethodDefinition:

```
Property Name (StrictFormalParameters) { FunctionBody }
get PropertyName () { FunctionBody }
set PropertyName (PropertySetParameterList) { FunctionBody }
```

1. Return HasComputedPropertyKey of `PropertyName`.

### 14.3.5 Static Semantics: PropName

See also: 12.2.5.6, 14.4.9, 14.5.12

MethodDefinition:

```
Property Name (StrictFormalParameters) { FunctionBody }
get PropertyName () { FunctionBody }
set PropertyName (PropertySetParameterList) { FunctionBody }
```

1. Return PropName of `PropertyName`.

### 14.3.6 Static Semantics: ReferencesSuper

See also: 14.1.16, 14.2.12, 14.4.10.

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`.

MethodDefinition: 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. Let `closure` be FunctionCreate(Method, StrictFormalParameters, FunctionBody, scope, strict). If `functionPrototype` was passed as a parameter then pass its value as the `functionPrototype` optional argument of FunctionCreate.
6. If ReferencesSuper of MethodDefinition is `true`, then
   a. Perform MakeMethod(closure, `propKey`, object).
7. Return the Record{[key]: `propKey`, [closure]: `closure`}.

14.3.9 Runtime Semantics: PropertyDefinitionEvaluation

With parameter `object`.

See also: 12.2.5.9, 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. ReturnIfAbrupt(`methodDef`).
3. SetFunctionName(`methodDef`, [[closure]], `methodDef`, [[key]]).
4. Assert: SetFunctionName will not return an abrupt completion.
5. Return CreateDataPropertyOrThrow(object, `methodDef`, [[key]], `methodDef`, [[closure]]).

MethodDefinition: GeneratorMethod

See 14.4.
MethodDefinition : **get** PropertyName ( ) { FunctionBody }

1. Let propKey be the result of evaluating PropertyName.
2. Return IfAbrupt(propKey).
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]
7. If ReferencesSuper of MethodDefinition is true, then
   a. Perform MakeMethod(closure, propKey, object).
8. SetFunctionName(closure, propKey, "get").
9. Assert: SetFunctionName will not return an abrupt completion.
10. Let desc be the PropertyDescriptor([[Get]], closure, [[Enumerable]], true, [[Configurable]], true)
11. Return DefinePropertyOrThrow(object, propKey, desc).

MethodDefinition : **set** PropertyName ( PropertySetParameterList ) { FunctionBody }

1. Let propKey be the result of evaluating PropertyName.
2. Return IfAbrupt(propKey).
3. Let strict be IsStrict of FunctionBody.
4. Let scope be the running execution context’s LexicalEnvironment.
5. Let closure be FunctionCreate(Method, PropertySetParameterList, FunctionBody, scope, strict).
6. If ReferencesSuper of MethodDefinition is true, then
   a. Perform MakeMethod(closure, propKey, object).
7. SetFunctionName(closure, propKey, "set").
8. Assert: SetFunctionName will not return an abrupt completion.
9. Let desc be the PropertyDescriptor([[Set]], closure, [[Enumerable]], true, [[Configurable]], true)
10. Return DefinePropertyOrThrow(object, propKey, desc).

14.4 Generator Function Definitions

Syntax

GeneratorMethod[Yield] :
  * PropertyName [Yield] (StrictFormalParameters, [YieldGeneratorParameter]) { GeneratorBody[Yield] }

GeneratorDeclaration[Yield, Default] :
  function * BindingIdentifier[Yield, ?Default] (FormalParameters[Yield, GeneratorParameter]) { GeneratorBody[Yield] }

GeneratorExpression :
  function * BindingIdentifier[Yield, ?Default] (FormalParameters[Yield, GeneratorParameter]) { GeneratorBody[Yield] }

YieldExpression[Yield] :
  yield
  yield [no LineTerminator here] [Lexical goal InputElement RegExp] AssignmentExpression[Yield, Yield]

NOTE 1 YieldExpression cannot be used within the FormalParameters of a generator function because any expressions that are part of FormalParameters are evaluate before the resulting generator object is in a resumable state.

NOTE 2 Abstract operations relating to generator objects are defined in 25.3.3.
14.4.1 Static Semantics: Early Errors

GeneratorMethod: * PropertyName ( StrictFormalParameters ) { GeneratorBody }

- It is a Syntax Error if any element of the BoundNames of StrictFormalParameters also occurs in the LexicallyDeclaredNames of GeneratorBody.

GeneratorDeclaration : function * BindingIdentifier ( FormalParameters ) { GeneratorBody }

and

GeneratorExpression : function * BindingIdentifier arg ( FormalParameters ) { GeneratorBody }

- If the source code matching this production is strict code, the Early Error rules for StrictFormalParameters ; FormalParameters are applied.
- If the source code matching this production is strict code, it is a Syntax Error if BindingIdentifier is the IdentifierName eval or the IdentifierName arguments.
- It is a Syntax Error if any element of the BoundNames of FormalParameters also occurs in the LexicallyDeclaredNames of GeneratorBody.

14.4.2 Static Semantics: BoundNames

See also: 13.2.1.2, 13.2.2.1, 12.1.2, 13.6.4.2, 14.1.3, 14.2.2, 14.5.2, 15.2.1.2, 15.2.2.1.

GeneratorDeclaration : function * BindingIdentifier ( FormalParameters ) { GeneratorBody }

1. Return the BoundNames of BindingIdentifier.

14.4.3 Static Semantics: ComputedPropertyContains

With parameter symbol.

See also: 12.2.5.2, 14.3.2, 14.5.5.

GeneratorMethod : * PropertyName ( StrictFormalParameters ) { GeneratorBody }

1. Return the result of ComputedPropertyContains for PropertyName with argument symbol.

14.4.4 Static Semantics: Contains

With parameter symbol.

See also: 5.3, 12.2.5.2, 12.3.1.1, 14.1.4, 14.2.3, 14.5.4

GeneratorDeclaration : function * BindingIdentifier ( FormalParameters ) { GeneratorBody }

1. Return false.

GeneratorExpression : function * BindingIdentifier arg ( FormalParameters ) { GeneratorBody }

1. Return false.

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

See also: 12.2.5.4, 14.3.4

GeneratorMethod:** PropertyName ( StrictFormalParameters ) { GeneratorBody }
  1. Return IsComputedPropertyKey of PropertyName.

14.4.6 Static Semantics: HasName

See also: 14.1.8, 14.2.8, 14.5.6.

GeneratorExpression:** function * ( FormalParameters ) { GeneratorBody }
  1. Return false.

GeneratorExpression:** function * BindingIdentifier ( FormalParameters ) { GeneratorBody }
  1. Return true.

14.4.7 Static Semantics: IsConstantDeclaration

See also: 13.2.1.3, 14.1.8, 14.5.5.

GeneratorDeclaration:** function * BindingIdentifier ( FormalParameters ) { GeneratorBody }
  1. Return false.

14.4.8 Static Semantics: IsFunctionDefinition

See also: 12.2.0.2, 12.2.9.2, 12.3.1.2, 12.4.2, 12.5.2, 12.6.1, 12.7.1, 12.8.1, 12.9.1, 12.10.1, 12.11.1, 12.12.1, 12.13.1, 12.14.2, 12.15.1, 14.3.11, 14.5.8.

GeneratorExpression:** function * ( FormalParameters ) { GeneratorBody }
  1. Return true.

GeneratorExpression:** function * BindingIdentifier ( FormalParameters ) { GeneratorBody }
  1. Return true.

14.4.9 Static Semantics: PropName

See also: 12.2.5.6, 14.3.5, 14.5.12

GeneratorMethod:** PropertyName ( StrictFormalParameters ) { GeneratorBody }
  1. Return PropName of PropertyName.

14.4.10 Static Semantics: ReferencesSuper

See also: 14.1.16, 14.2.12, 14.3.6.
GeneratorDeclaration: `function * BindingIdentifier ( FormalParameters ) { GeneratorBody }`

1. If `FormalParameters` Contains `super` is `true`, then return `true`.
2. Return `GeneratorBody` Contains `super`.

GeneratorExpression: `function * BindingIdentifieropt ( FormalParameters ) { GeneratorBody }`

1. If `FormalParameters` Contains `super` is `true`, then return `true`.
2. Return `GeneratorBody` Contains `super`.

GeneratorMethod: `*(PropertyName ( StrictFormalParameters ) { GeneratorBody })`

1. If `StrictFormalParameters` Contains `super` is `true`, then return `true`.
2. Return `GeneratorBody` Contains `super`.

14.4.11 Runtime Semantics: EvaluateBody

With parameter `functionObject`.

See also: 14.1.19, 14.2.16.

GeneratorBody: `FunctionBody`

1. Assert: A Function Environment Record containing a this binding has already been activated.
2. Let `env` be `GetThisEnvironment()`.
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 `OrdinaryCreateFromConstructor(functionObject, "%GeneratorPrototype%", ( [[GeneratorState]]: [[GeneratorContext]]) )`.
   b. Return `IfAbrupt(newG)`.
   c. Let `G` be `newG`.
5. Return `GeneratorStart(G, FunctionBody)`.

14.4.12 Runtime Semantics: InstantiateFunctionObject

With parameter `scope`.

See also: 14.1.21.

GeneratorDeclaration: `function * BindingIdentifier ( FormalParameters ) { GeneratorBody }`

1. If the `GeneratorDeclaration` is contained in strict code or if its `GeneratorBody` is strict code, then let `strict` be `true`. Otherwise let `strict` be `false`.
2. Let `name` be `StringValue of BindingIdentifier`.
3. Let `F` be `GeneratorFunctionCreate(Normal, FormalParameters, body, GeneratorBody, strict)`.
4. If `ReferencesSuper of GeneratorDeclaration` is `true`, then
   a. Perform `MakeMethod(F, name, undefined)`.
5. Let `prototype` be `ObjectCreate(%GeneratorPrototype%)`.
6. Perform `MakeConstructor(F, true, prototype)`.
7. Set `FunctionName(F, name)`.
8. Assert: `SetFunctionName` will not return an abrupt completion.
14.4.13 Runtime Semantics: PropertyDefinitionEvaluation

With parameter object.

See also: 12.2.5.9, 14.3.9, B.3.1

GeneratorMethod : * PropertyName ( StrictFormalParameters ) { GeneratorBody }

1. Let propKey be the result of evaluating PropertyName.
2. ReturnIfAbrupt(propKey).
3. Let strict be IsStrict of GeneratorBody.
4. Let scope be the running execution context’s LexicalEnvironment.
5. Let closure be GeneratorFunctionCreate(Method, StrictFormalParameters, GeneratorBody, scope, strict).
6. If ReferencesSuper of GeneratorMethod is true, then
   a. Perform MakeMethod(closure, propKey, object).
7. Perform MakeConstructor(closure, true, prototype).
8. SetFunctionName(closure, propKey).
9. Assert: SetFunctionName will not return an abrupt completion.
10. Return CreateDataPropertyOrThrow(object, propKey, closure).

14.4.14 Runtime Semantics: Evaluation

GeneratorDeclaration : function * BindingIdentifier ( FormalParameters ) { GeneratorBody }

1. Return NormalCompletion(empty)

GeneratorExpression : function * ( FormalParameters ) { GeneratorBody }

1. Let strict be IsStrict of GeneratorBody.
2. Let scope be the LexicalEnvironment of the running execution context.
3. Let closure be GeneratorFunctionCreate(Normal, FormalParameters, GeneratorBody, scope, strict).
4. If ReferencesSuper of GeneratorExpression is true, then
   a. Perform MakeMethod(closure, undefined, undefined).
5. Let prototype be ObjectCreate(%GeneratorPrototype%).
6. Perform MakeConstructor(closure, true, prototype).
7. Return closure.

GeneratorExpression : function * BindingIdentifier ( FormalParameters ) { GeneratorBody }

1. Let strict be IsStrict of GeneratorBody.
2. Let runningContext be the running execution context’s Lexical Environment.
3. Let funcEnv be NewDeclarativeEnvironment(runningContext).
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 GeneratorFunctionCreate(Normal, FormalParameters, GeneratorBody, funcEnv, strict).
8. If ReferencesSuper of GeneratorExpression is true, then
   a. Perform MakeMethod(closure, name, undefined).
9. Let prototype be ObjectCreate(%GeneratorPrototype%).
10. Perform MakeConstructor(closure, true, prototype).
11. SetFunctionName(closure, name).
12. Assert: SetFunctionName will not return an abrupt completion.
13. Call the InitializeBinding concrete method of envRec passing name and closure as the arguments.

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 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 GeneratorYield(CreateIterResultObject(value, false)).

YieldExpression : yield * AssignmentExpression
  1. Let exprRef be the result of evaluating AssignmentExpression.
  2. Let value be GetValue(exprRef).
  3. Let iterator be GetIterator(ToObject(value)).
  4. ReturnIfAbrupt(iterator).
  5. Let received be NormalCompletion(undefined).
  6. Repeat
     a. Let innerResult be empty.
        b. If received.[[type]] is normal, then
           i. Let innerResult be IteratorNext(iterator, received.[[value]]).
           ii. ReturnIfAbrupt(innerResult).
        c. Else if received.[[type]] is throw, then
           i. Let hasThrow be HasProperty(iterator, "throw");
           ii. ReturnIfAbrupt(hasThrow).
           iii. If hasThrow is true, then
                1. Let innerResult be Invoke(iterator, "throw", (received[[value]])).
                2. ReturnIfAbrupt(innerResult).
                3. NOTE: Exceptions from the inner iterator throw method are propagated.
                iv. Let done be IteratorComplete(innerResult).
                v. ReturnIfAbrupt(done).
        d. Else if received.[[type]] is return, then
           i. Let hasReturn be HasProperty(iterator, "return");
           ii. ReturnIfAbrupt(hasReturn).
           iii. If hasReturn is false, then return received.
           iv. Let innerReturnValue beInvoke(iterator, "return", (received.[[value]])).
           v. ReturnIfAbrupt(innerReturnValue).
           vi. If Type(innerReturnValue) is not Object, then throw a TypeError exception.
           vii. Return Completion{[[type]]: return, [[value]]: IteratorValue(innerReturnValue), [[target]]: empty}.
        e. If Type(innerResult) is not Object, then throw a TypeError exception.
        f. Let done be IteratorComplete(innerResult).
        g. ReturnIfAbrupt(done).
        h. If done is true, then
i. Return IteratorValue (innerResult).
  i. Let received be GeneratorYield(innerResult).

14.5 Class Definitions

Syntax

ClassDeclaration[Yield, Default] :
  class BindingIdentifier[Yield, ?Default] ClassTail[Yield]

ClassExpression[Yield, GeneratorParameter] :
  class BindingIdentifier[YieldOpt, ClassTail[Yield, ?GeneratorParameter]]

ClassTail[Yield, GeneratorParameter] :
  [+GeneratorParameter] ClassHeritageOpt { ClassBodyOpt }

ClassHeritage[Yield] :
  extends LeftHandSideExpression[Yield]

ClassBody[Yield] :
  ClassElementList[Yield]

ClassElementList[Yield] :
  ClassElement[Yield]
  ClassElementList[Yield] ClassElement[Yield]

ClassElement[Yield] :
  MethodDefinition[Yield]
  static MethodDefinition[Yield]

; 

NOTE A ClassBody is always strict code.

14.5.1 Static Semantics: Early Errors

ClassDeclaration : class BindingIdentifier ClassTail
ClassExpression : class BindingIdentifier ClassTail
  • It is a Syntax Error if the StringValue of BindingIdentifier is "let".

ClassBody : ClassElementList
  • It is a Syntax Error if PrototypePropertyNameList of ClassElementList contains more than one occurrence of "constructor".

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, 12.1.2, 13.6.4.2, 14.1.3, 14.2.2, 14.4.2, 15.2.1.2, 15.2.2.1.

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 is
not an accessor property or generator definition.

14.5.4 Static Semantics: Contains

With parameter symbol.

See also: 5.3, 12.2.5.2, 12.3.1.1, 14.1.4, 14.2.3, 14.4.3

ClassTail : ClassHeritage opt { ClassBody }
1. If symbol is ClassBody, return true.
2. If symbol is ClassHeritage, then
   a. If ClassHeritage is present, return true otherwise return false.
3. Let inHeritage be ClassHeritage Contains symbol.
4. If inHeritage is true, then return true.
5. Return the result of ComputedPropertyContains for ClassBody with argument symbol.

NOTE Static semantic rules that depend upon substructure generally do not look into class bodies except for
PropertyName productions.

14.5.5 Static Semantics: ComputedPropertyContains

With parameter symbol.

See also: 12.2.5.2, 14.3.2, 14.4.3.

ClassElementList : ClassElementList ClassElement
1. Let \texttt{inList} be the result of \texttt{ComputedPropertyContains} for \texttt{ClassElementList} with argument \texttt{symbol}.
2. If \texttt{inList} is \texttt{true}, then return \texttt{true}.
3. Return the result of \texttt{ComputedPropertyContains} for \texttt{ClassElement} with argument \texttt{symbol}.

\texttt{ClassElement : MethodDefinition}
1. Return the result of \texttt{ComputedPropertyContains} for \texttt{MethodDefinition} with argument \texttt{symbol}.

\texttt{ClassElement : static MethodDefinition}
1. Return the result of \texttt{ComputedPropertyContains} for \texttt{MethodDefinition} with argument \texttt{symbol}.

\texttt{ClassElement : ;}
1. Return \texttt{false}.

14.5.6 Static Semantics: HasName

See also: 14.1.8, 14.2.8, 14.4.6.

\texttt{ClassExpression : class ClassTail}
1. Return \texttt{false}.

\texttt{ClassExpression : class BindingIdentifier ClassTail}
1. Return \texttt{true}.

14.5.7 Static Semantics: IsConstantDeclaration

See also: 13.2.1.3, 14.1.8, 14.4.5.

\texttt{ClassDeclaration : class BindingIdentifier ClassTail}
1. Return \texttt{false}.

14.5.8 Static Semantics: IsFunctionDefinition

See also: 12.2.0.2, 12.2.9.2, 12.3.1.2, 12.4.2, 12.5.2, 12.6.1, 12.7.1, 12.8.1, 12.9.1, 12.10.1, 12.11.1, 12.12.1, 12.13.1, 12.14.2, 12.15.1, 14.1.11, 14.4.8.

\texttt{ClassExpression : class ClassTail}
1. Return \texttt{true}.

\texttt{ClassExpression : class BindingIdentifier ClassTail}
1. Return \texttt{true}.

14.5.9 Static Semantics: IsStatic

\texttt{ClassElement : MethodDefinition}
1. Return \texttt{false}.
ClassElement : `static` MethodDefinition
  1. Return `true`.

ClassElement : ;
  1. Return `false`.

14.5.10 Static Semantics: NonConstructorMethodDefinitions

ClassElementList : ClassElement
  1. If `ClassElement` is the production `ClassElement` ; then, return a new empty List.
  2. If IsStatic of `ClassElement` is `false` and PropName of `ClassElement` is "constructor", return a new empty List.
  3. Return a List containing `ClassElement`.

ClassElementList : ClassElementList ClassElement
  1. Let `list` be NonConstructorMethodDefinitions of `ClassElementList`.
  2. If `ClassElement` is the production `ClassElement` ; then, return list.
  3. If IsStatic of `ClassElement` is `false` and PropName of `ClassElement` is "constructor", return list.
  4. Append `ClassElement` to the end of list.
  5. Return list.

14.5.11 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.

14.5.12 Static Semantics: PropName

See also: 12.2.5.6, 14.3.5, 14.4.9

ClassElement : ;
  1. Return empty.

14.5.13 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.

14.5.14 Runtime Semantics: ClassDefinitionEvaluation

With parameter className.

ClassTail : ClassHeritageopt { ClassBodyopt }
   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. ReturnIfAbrupt(superclass).
      c. If superclass is null, then
         i. Let protoParent be null.
      d. Else if IsConstructor(superclass) is false, then throw a TypeError exception.
      e. Else
         i. Let protoParent be Get(superclass, "prototype").
         ii. ReturnIfAbrupt(protoParent).
         iii. If Type(protoParent) is neither Object nor Null, throw a TypeError exception.
         iv. Let constructorParent be superclass.
   3. Let proto be ObjectCreate(protoParent).
   4. Let lex be the LexicalEnvironment of the running execution context.
   5. If className is not undefined, then
      a. Let scope be NewDeclarativeEnvironment(lex).
      b. Let envRec be scope's environment record.
      c. Call the CreateImmutableBinding concrete method of envRec passing className as the argument.
      d. Set the running execution context's LexicalEnvironment to scope.
   6. If ClassBodyopt is not present, then let constructor be empty.
   7. Else, let constructor be ConstructorMethod of ClassBody.
   8. 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.
   9. Let strict be true.
   10. Let constructorInfo be the result of performing DefineMethod for constructor with arguments proto and constructorParent as the optional functionPrototype argument.
   11. Let F be constructorInfo.[[closure]].

Commented [AWB956]: Note that this variable currently isn't used in this algorithm
Commented [AWB857]: As it now stands, this will never be an abrupt completion
12. Perform the abstract operation `MakeConstructor` with argument `F` and `false` as the optional `writablePrototype` argument and `proto` as the optional `prototype` argument.
13. Let `desc` be the `PropertyDescriptor` `[[Value]]: F, [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true`. Call the `[[DefineOwnProperty]]` internal method of `proto` with arguments "constructor" and `desc`.
14. If `ClassBodyopt` is not present, then let `methods` be a new empty `List`.
15. Else, let `methods` be `NonConstructorMethodDefinitions` of `ClassBody`.
16. For each `ClassElement m` in order from `methods`
   a. If `IsStatic` of `m` is `false`, then
      i. Let `status` be the result of performing `PropertyDefinitionEvaluation` for `m` with argument `proto`.
   b. Else,
      i. Let `status` be the result of performing `PropertyDefinitionEvaluation` for `m` with argument `F`.
   c. If `status` is an abrupt completion, then
      i. Set the running execution context’s LexicalEnvironment to `lex`.
      ii. Return `status`.
17. Set the running execution context’s LexicalEnvironment to `lex`.
18. If `className` is not `undefined`, then
   a. Call the `InitializeBinding` concrete method of `envRec` passing `className` and `F` as the arguments.

14.5.15 Runtime Semantics: Evaluation

**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 `hasOwnProperty` be `HasOwnProperty(value, "name")`.
5. ReturnIfAbrupt(`hasOwnProperty`).
6. If `hasOwnProperty` is `false`, then
   a. Perform `SetFunctionName(value, className)`.
   b. Assert: `SetFunctionName` will not return an abrupt completion.
7. Let `env` be the running execution context’s LexicalEnvironment.
8. Let `status` be the result of performing `BindingInitialization` for `BindingIdentifier` passing `value` and `env` as the arguments.
9. ReturnIfAbrupt(`status`).
10. Return NormalCompletion(`empty`).

**ClassExpression** : `class BindingIdentifieropt ClassTail`
1. If `BindingIdentifieropt` 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 `hasOwnProperty` be `HasOwnProperty(value, "name")`.
   b. ReturnIfAbrupt(`hasOwnProperty`).
   c. If `hasOwnProperty` is `false`, then
      i. Perform `SetFunctionName(value, className)`.
      ii. Assert: `SetFunctionName` will not return an abrupt completion.
6. Return NormalCompletion(value).

NOTE If the class definition included a "name" static method then that method is not over-written with a "name" data property for the class name.

14.6 Tail Position Calls

14.6.1 Static Semantics: IsInTailPosition(nonterminal) Abstract Operation

The abstract operation IsInTailPosition with argument nonterminal performs the following steps:

1. Assert: nonterminal is a parsed grammar production.
2. If the source code matching nonterminal is not strict code, then return false.
3. If nonterminal is not contained within a FunctionBody or ConciseBody, then return false.
4. Let body be the FunctionBody or ConciseBody that most closely contains nonterminal.
5. If body is the FunctionBody of a GeneratorMethod, GeneratorDeclaration, or a GeneratorExpression, then return false.
6. Return the result of 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.

NOTE nonterminal is a parsed grammar production that represent a specific range of source code. When the following algorithms compare nonterminal to other grammar symbols they are testing whether the same source code was matched by both symbols.

14.6.2.1 Statement Rules

ConciseBody : AssignmentExpression

1. Return HasProductionInTailPosition of AssignmentExpression with argument nonterminal.

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.

FunctionStatementList : [empty]

StatementList : Declaration

Statement :

VariableStatement
EmptyStatement
ExpressionStatement
ContinueStatement
BreakStatement
ThrowStatement
DebuggerStatement
Block: {}  
ReturnStatement: return ;  
LabelledItem: FunctionDeclaration  
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 ) ; 
  while ( Expression ) Statement 
  for ( Expressionopt ; Expressionopt ; Expressionopt ) Statement 
  for ( var VariableDeclarationList ; Expressionopt ; Expressionopt ) Statement 
  for ( LexicalDeclaration Expressionopt ; Expressionopt ) Statement 
WithStatement: with ( Expression ) Statement  
  1. Return HasProductionInTailPosition of Statement with argument nonterminal.

LabelledStatement:  
  LabelIdentifier : LabelledItem  
  2. Return HasProductionInTailPosition of LabelledItem 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: { CaseClausesopt 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 : \texttt{default} : StatementList_opt

1. If \texttt{StatementList} is present, return HasProductionInTailPosition of \texttt{StatementList} with argument nonterminal.
2. Return \texttt{false}.

TryStatement : \texttt{try} Block Catch

1. Return HasProductionInTailPosition of \texttt{Catch} with argument nonterminal.

TryStatement : \texttt{try} Block Finally

TryStatement : \texttt{try} Block Catch Finally

1. Return HasProductionInTailPosition of Finally with argument nonterminal.

Catch : \texttt{catch} ( CatchParameter ) Block

1. Return HasProductionInTailPosition of Block 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. Function calls cannot return reference values, so such a GetValue operation will always returns the same value as the actual function call result.

AssignmentExpression :

\texttt{YieldExpression}
\texttt{ArrowFunction}
\texttt{LeftHandSideExpression = AssignmentExpression}
\texttt{LeftHandSideExpression AssignmentOperator AssignmentExpression}

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

EqualityExpression :

EqualityExpression \texttt{==} RelationalExpression
EqualityExpression \texttt{=} RelationalExpression
EqualityExpression \texttt{===} RelationalExpression
EqualityExpression \texttt{!==} RelationalExpression

RelationalExpression :

RelationalExpression \texttt{<} ShiftExpression
RelationalExpression \texttt{>} ShiftExpression
RelationalExpression \texttt{<=} ShiftExpression
RelationalExpression \texttt{>=} ShiftExpression
RelationalExpression \texttt{instanceof} ShiftExpression
RelationalExpression \texttt{in} ShiftExpression

ShiftExpression :

ShiftExpression \texttt{<<} AdditiveExpression
ShiftExpression \texttt{>>} AdditiveExpression
ShiftExpression \texttt{>>>} 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
    ArrayInitializer
    ObjectLiteral
    FunctionExpression
    ClassExpression
    GeneratorExpression
    RegExpLiteral
    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 false.

MemberExpression :
  MemberExpression TemplateLiteral
    new super Arguments
    new MemberExpression Arguments
1. If this MemberExpression is nonterminal, then return true.
2. Return false.

NewExpression :
  new NewExpression
    new super
1. If this NewExpression is nonterminal, then return true.
2. Return false.

PrimaryExpression : ( CoverParenthesizedExpressionAndArrowParameterList
1. Let expr be CoveredParenthesizedExpressionAndArrowParameterList.
2. Return HasProductionInTailPosition of expr with argument nonterminal.

ParenthesizedExpression :
  ( 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.

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: Scripts and Modules

15.1 Scripts

Syntax

Script : ScriptBody\textsubscript{opt}

ScriptBody : StatementList

15.1.1 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 StatementList.
- It is a Syntax Error if StatementList 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.

15.1.2 Static Semantics: IsStrict

See also: 14.1.13, 15.2.0.7.

ScriptBody : StatementList

1. If this ScriptBody is contained in strict code or if StatementList is strict code, then return true.
   Otherwise, return false.

15.1.3 Static Semantics: LexicallyDeclaredNames


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.

15.1.4 Static Semantics: LexicallyScopedDeclarations

See also: 13.1.3, 13.11.3, 13.12.5, 14.1.15, 14.2.11, 15.2.0.11.

ScriptBody : StatementList
  1.  Return TopLevelLexicallyScopedDeclarations of StatementList.

15.1.5 Static Semantics: VarDeclaredNames


ScriptBody : StatementList
  1.  Return TopLevelVarDeclaredNames of StatementList.

15.1.6 Static Semantics: VarScopedDeclarations


ScriptBody : StatementList
  1.  Return TopLevelVarScopedDeclarations of StatementList.

15.1.7 Runtime Semantics: ScriptEvaluation

  With argument realm and deletableBindings.

  Script : ScriptBodyopt
  1.  The code of this Script is strict mode code if the Directive Prologue (14.1.1) 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 GlobalDeclarationInstantiation(ScriptBody, globalEnv, and deletableBindings).
  5.  ReturnIfAbrupt(status).
  6.  Let scriptCxt be a new ECMAScript code execution context.
  7.  Set the Function of scriptCxt to null.
  8.  Set the Realm of scriptCxt to realm.
  9.  Set the VariableEnvironment of scriptCxt to globalEnv.
  10.  Set the LexicalEnvironment of scriptCxt to globalEnv.
  11.  If there is a currently running execution context, suspend it.
  12.  Push scriptCxt on to the execution context stack; scriptCxt is now the running execution context.
  13.  Let result be the result of evaluating ScriptBody.
  14.  Suspend scriptCxt and remove it from the execution context stack.
  15.  Assert: the execution context stack is not empty.
16. Resume the context that is now on the top of the execution context stack as the running execution context.
17. 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.

15.1.8 Runtime Semantics: GlobalDeclarationInstantiation

NOTE When a 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 neither a VariableDeclaration or a ForBinding, then
      i. Assert: d is either a FunctionDeclaration or a GeneratorDeclaration.
      ii. NOTE If there are multiple FunctionDeclarations for the same name, the last declaration is used.
      iii. Let fn be the sole element of the BoundNames of d.
      iv. If fn is not an element of declaredFunctionNames, then
         1. Let fnDefinable be the result of calling env's CanDeclareGlobalFunction concrete method passing fn as the argument.
         2. If fnDefinable is false, throw TypeError exception.
         3. Append fn to declaredFunctionNames.
   4. Insert d as the first element of functionsToInitialize.
10. Let declaredVarNames be an empty List.
11. For each d in varDeclarations, do
   a. If d is a VariableDeclaration or a ForBinding 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 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 if the global object is an ordinary object. However, if the global object is a Proxy exotic object it may exhibit behaviours that cause abnormal terminations in some of the following steps.

13. Let lexDeclarations be the LexicallyScopedDeclarations of script.
14. For each element d in lexDeclarations do
   a. NOTE: Lexically declared names are only instantiated here but not initialized.
   b. For each element dn of the BoundNames of d do
      i. If IsConstantDeclaration of d is true, then
         1. 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. ReturnIfAbrupt(status).
15. For each production 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. ReturnIfAbrupt(status).
16. 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. ReturnIfAbrupt(status).
17. Return NormalCompletion(empty)

NOTE: Early errors specified in 15.1.1 prevent name conflicts between function/var declarations and let/const/class declarations as well as redeclaration of let/const/class 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. However, if the global object is defined using Proxy exotic objects the runtime tests for conflicting declarations may be unreliable resulting in an abrupt completion and some global declarations not being instantiated. If this occurs, the code for the Script is not evaluated.

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 declarations.

15.1.9 Runtime Semantics: ScriptEvaluationJob ( source )

The job ScriptEvaluationJob with parameter source parses, validates, and evaluates the Script represented by source.
1. Assert: source is a SourceCharacter sequence (see 10).
2. Parse source using Script as the goal symbol and analyze the parse result for any Early Error conditions. If the parse was successful and no early errors were found, then let script be the resulting parse tree. Otherwise, let script be an indication of one or more parsing errors and/or early errors. Parsing and early error detection may be interleaved in an implementation dependent manner. If more than one parse or early error is present, the number and ordering of reported errors is implementation dependent but at least one error must be reported.
3. If script is an error indication, then
   a. Report or log the error(s) in an implementation dependent manner.
   b. Let status be NormalCompletion(undefined).
4. Else,
a. Let \texttt{realm} be the running execution context’s Realm.

b. Let \texttt{status} be the result of ScriptEvaluation of \texttt{script} with arguments \texttt{realm} and \texttt{false}.

5. NextJob \texttt{status}.

\textbf{NOTE} An implementation may parse a \texttt{Script} and analyze it for Early Error conditions prior to the execution of the \texttt{ScriptEvaluation}Job for that \texttt{Script}. However, the reporting of any errors must be deferred until the \texttt{ScriptEvaluation}Job is actually executed.

\section*{15.2 Modules}

\textbf{Syntax}

\begin{verbatim}
Module : ModuleBody

ModuleBody : ModuleItemList

ModuleItemList : ModuleItem ModuleItemList

ModuleItem : ImportDeclaration ExportDeclaration StatementListItem
\end{verbatim}

\subsection*{15.2.0 Module Static Semantics}

\textbf{15.2.0.1 Static Semantics: Early Errors}

\begin{verbatim}
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 \texttt{super}.
\end{verbatim}

\textbf{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.

\textbf{15.2.0.2 Static Semantics: DeclaredNames}

\begin{verbatim}
Module : [empty]
1. Return a new empty List.

Module : ModuleBody
1. Let \texttt{names} be LexicallyDeclaredNames of ModuleBody.
2. Append to \texttt{names} the elements of the VarDeclaredNames of ModuleBody.
3. Return \texttt{names}.
\end{verbatim}
15.2.0.3 Static Semantics: ExportedBindings

See also: 15.2.2.2.

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
    StatementList
    Item
    1. Return a new empty List.

15.2.0.4 Static Semantics: ExportEntries

See also: 15.2.2.3.

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
    StatementList
    Item
    1. Return a new empty List.

15.2.0.5 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
  StatementListItem
  
  1. Return a new empty List.

15.2.0.6 Static Semantics: ImportEntries

See also: 15.2.1.3.

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
  StatementListItem
  
  1. Return a new empty List.

15.2.0.7 Static Semantics: IsStrict

See also: 14.1.13, 15.1.2.

ModuleBody : ModuleItemList
  
  1. Return true.

15.2.0.8 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.

15.2.0.9 Static Semantics: ModuleRequests

See also: 15.2.1.5, 15.2.2.5.

ModuleItemList : [empty]
  
  1. Return a new empty List.

ModuleItemList : ModuleItemList ModuleItem
  
  1. Return ModuleRequests of ModuleItem.

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.

**15.2.0.10 Static Semantics: LexicallyDeclaredNames**


**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. If `ExportDeclaration` is `export VariableStatement`; then return a new empty List.
2. 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.

**15.2.0.11 Static Semantics: LexicallyScopedDeclarations**


**ModuleItemList : [empty]**

1. Return a new empty List.

**ModuleItemList : ModuleItemList ModuleItem**

1. Let `declarations` be `LexicallyScopedDeclarations` of `ModuleItemList`.
2. Append to `declarations` the elements of the `LexicallyScopedDeclarations` 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.

15.2.0.12 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.

15.2.0.13 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.

15.2.0.14 Static Semantics: VarScopedDeclarations


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 VarScopedDeclarations of VariableStatement.
2. Return a new empty List.

15.2.0.15 Runtime Semantics: ModuleDeclarationInstantiation( code, env )

1. Let declarations be the LexicallyScopedDeclarations of code.
2. Let functionsToInitialize 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 production or a FunctionDeclaration production, then
      i. Append d to functionsToInitialize.
4. For each production f in functionsToInitialize, in list order do
   a. Let fn be the sole element of the BoundNames of f.
   b. Let fo be the result of performing InstantiateFunctionObject for f with argument env.
   c. Call env’s InitializeBinding concrete method passing fn, and fo as the arguments.

15.2.1 Imports

Syntax

ImportDeclaration : ModuleImport
  import ImportClause FromClause ;
  import ModuleSpecifier ;

ModuleImport : module [no LineTerminator here] ImportedBinding FromClause ;

FromClause : from ModuleSpecifier

ImportClause : ImportedBinding
  ImportedBinding , NamedImports
  NamedImports

NamedImports : 
  { ImportSpecifier }
  { ImportSpecifier , }

ImportsList : ImportSpecifier
  ImportList , ImportSpecifier
**ImportSpecifier**: 
- **ImportedBinding**
  - **IdentifierName** as **ImportedBinding**

**ModuleSpecifier**: 
- **StringLiteral**

**ImportedBinding**: 
- **BindingIdentifier**

### 15.2.1 Static Semantics: Early Errors

**ModuleItem**: **ImportDeclaration**

- It is a Syntax Error if the **BoundNames** of **ImportDeclaration** contains any duplicate entries.

### 15.2.1.2 Static Semantics: BoundNames

See also: 13.2.1.2, 13.2.2.1, 12.1.2, 0, 14.1.3, 14.2.2, 14.4.2, 14.5.2, 15.2.2.1.

**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**.

**NamedImports**: `{ }`

1. Return a new empty List.

**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**.

### 15.2.1.3 Static Semantics: ImportEntries

See also: 15.2.0.6.
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`.

15.2.1.4 Static Semantics: ImportEntriesForModule

With parameter `module`.

ImportClause: `ImportedBinding`
1. Let `localName` be the StringValue of `ImportedBinding`.
2. Let `defaultEntry` be the Record `{[[ModuleRequest]]: module, [[ImportName]]: "default", [[LocalName]]: localName }`.
3. Return a new List containing `defaultEntry`.

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 ImportEntriesForModule of `NamedImports` with argument `module`.
5. Return `entries`.

NamedImports: `{ }`
1. Return a new empty List.

ImportsList: `ImportsList , ImportSpecifier`
1. Let `specs` be the ImportEntriesForModule of `ImportsList` with argument `module`.
2. Append to `specs` the elements of the ImportEntriesForModule of `ImportSpecifier` with argument `module`.
3. Return `specs`.

ImportSpecifier: `ImportedBinding`
1. Let `localName` be the StringValue of `ImportedBinding`.
2. Let `entry` be the Record `{[[ModuleRequest]]: module, [[ImportName]]: localName , [[LocalName]]: localName }`.
3. Return a new List containing `entry`.
ImportSpecifier : IdentifierName as ImportedBinding

1. Let importName be the StringValue of IdentifierName.
2. Let localName be the StringValue of ImportedBinding.
3. Let entry be the Record {[[ModuleRequest]]: module, [[ImportName]]: importName, [[LocalName]]: localName}.
4. Return a new List containing entry.

15.2.1.5 Static Semantics: ModuleRequests

See also: 15.2.0.9, 15.2.2.5.

ImportDeclaration : import ImportClause FromClause ;
1. Return ModuleRequests of FromClause.

ModuleImport : module ImportedBinding FromClause ;
1. Return ModuleRequests of FromClause.

ModuleSpecifier : StringLiteral
1. Return a List containing the StringValue of StringLiteral.

15.2.1.6 Runtime Semantics: Module Objects

ModuleImport : module ImportedBinding FromClause ;

An ModuleImport imports a module and introduces a single binding within the containing module environment. The value of such a binding as a Module object.

A Module object is an exotic object whose own properties corresponding to the ExportedBindings of the module identified by the ModuleImport FromClause. Each property name is the StringValue of the corresponding exported binding. These are the only properties of a Module object. Each one is a read-only property with attributes {
[[Configurable]]: false, [[Enumerable]]: true}. Module objects are not extensible.

TO DO

Needs to decide whether a module object is an ordinary or an exotic object. Whether properties are accessor or defined via [[Get]], etc.

15.2.2 Exports

Syntax

ExportDeclaration : export * FromClause ;
export ExportClause[null|undefined] FromClause ;
export ExportClause ;
export VariableStatement
export Declaration[null|undefined]
export default AssignmentExpression[null|undefined] ;

Commented [AWB2261]: Only string keyed properties? Do we need a @@iterable property? Etc.

Commented [AWB2262]: TODO
ExportClause[NoReference] :
    {  }
    { {ExportsList[NoReference] } }
    { {ExportsList[NoReference] , } }

ExportsList[NoReference] :  
    ExportSpecifier[NoReference] 
    ExportsList[NoReference] , ExportSpecifier[NoReference] 

ExportSpecifier[NoReference] : 
    [NoReference] IdentifierReference 
    [NoReference] IdentifierReference as IdentifierName 
    [NoReference] IdentifierName 
    [NoReference] IdentifierName as IdentifierName 

NOTE ExportSpecifier is used to export bindings from the enclosing module Module. ExportSpecifier[NoReference] is used to export bindings from a referenced Module. In that case IdentifierReference restrictions are not applied to the naming of the items to be exported because they are not used to create local bindings.

15.2.2.1 Static Semantics: BoundNames

See also: 13.2.1.2, 13.2.2.1, 12.1.2, 0, 14.1.3, 14.2.2, 14.4.2, 14.5.2, 15.2.1.2.

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"

15.2.2.2 Static Semantics: ExportedBindings

See also: 15.2.0.2.

ExportDeclaration : export * FromClause ;

1. Return a new empty List.

Commented [AWB2363]: Why is default considered a bound name???
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 : ExportsList, ExportSpecifier

  1. Let names be the ExportedBindings of ExportsList.
  2. Append to names the elements of the ExportedBindings of ExportSpecifier.
  3. Return names.

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.

15.2.2.3 Static Semantics: ExportEntries

See also:15.2.0.4.

ExportDeclaration : export * FromClause ;

  1. Let module be the sole element of ModuleRequests of FromClause.
  2. Let entry be the Record { [[ModuleRequest]]: module, [[ImportName]]: 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. 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. Repeat for each name in names,
   a. Append to entries the Record `[[ModuleRequest]]: null, [[ImportName]]: null, [[LocalName]]: name, [[ExportName]]: name`.
4. Return entries.

ExportDeclaration : `export Declaration`;

1. Let entries be a new empty List.
2. Let names be the BoundNames of Declaration.
3. Repeat for each name in names,
   a. Append to entries the Record `[[ModuleRequest]]: null, [[ImportName]]: null, [[LocalName]]: name, [[ExportName]]: name`.
4. Return entries.

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.

15.2.2.4 Static Semantics: ExportEntriesForModule

With parameter module.

ExportClause : `{ }`

1. Return a new empty List.

ExportsList : ExportsList , ExportSpecifier

1. Let specs be the ExportEntriesForModule of ExportsList with argument module.
2. Append to specs the elements of the ExportEntriesForModule of ExportSpecifier with argument module.
3. Return specs.

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`.
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 }.

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 }.

ExportSpecifier : IdentifierName as IdentifierName

1. Let sourceName be the StringValue of the first IdentifierName.
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 }.

15.2.2.5 Static Semantics: ModuleRequests

See also: 15.2.0.9, 15.2.1.5.

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.

15.2.3 Runtime Semantics: Loader State

15.2.3.1 LoaderRecords and Loader Objects

Loader Records contain the state of a of distinct module loading context. Each Loader Record has the fields defined in Table 35. Loader objects (26.2) are ECMAScript objects that permit ECMAScript code to define and manage module loading contexts.
### Table 35 — Loader Record Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[Realm]]</code></td>
<td>Realm Record</td>
<td>The Realm associated with the loader. All scripts and modules evaluated by this loader run in the scope of the global object associated with this Realm.</td>
</tr>
<tr>
<td><code>[[Modules]]</code></td>
<td>List of Record <code>[[key]], [[value]]</code> where <code>[[key]]</code> is a String and <code>[[Module]]</code> is a Module Object</td>
<td>Normalized names bound to fully linked Module records. The list can contain modules whose code has not yet been evaluated. However, except for the case of cyclic imports, such modules are not exposed to user code.</td>
</tr>
<tr>
<td><code>[[Loads]]</code></td>
<td>List of Load Record</td>
<td>Outstanding asynchronous module load requests that have been made to this loader.</td>
</tr>
<tr>
<td><code>[[LoaderObj]]</code></td>
<td>Object or Undefined</td>
<td>The Loader object (26.2) that reflects this Loader Record.</td>
</tr>
</tbody>
</table>

#### 15.2.3.1.1 `CreateLoaderRecord(realm, object)` Abstract Operation

The abstract operation `CreateLoaderRecord` creates and returns a new Loader Record. The argument `realm` is the Realm record that will be associated with Loader. The argument `object` is the either `undefined` or the Loader object that will reflect this Loader record.

The following steps are taken:
1. Let `loader` be a new Loader Record.
2. Set `loader.[[Realm]]` to `realm`.
3. Set `loader.[[Modules]]` to a new empty List.
4. Set `loader.[[Loads]]` to a new empty List.
5. Set `loader.[[LoaderObj]]` to `object`.
6. Return `loader`.

#### 15.2.3.2 Load Records and LoadRequest Objects

The Load Record represents an attempt to locate, fetch, translate, and parse a single module.

Each Load Record has the fields defined in Table 36:
Table 36 — Load Record Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Status]]</td>
<td>One of: &quot;loading&quot;, &quot;loaded&quot;, &quot;linked&quot;, &quot;failed&quot;</td>
<td>The current state of this Load request.</td>
</tr>
<tr>
<td>[[Name]]</td>
<td>String</td>
<td>undefined</td>
</tr>
<tr>
<td>[[LinkSets]]</td>
<td>List of LinkSet Record</td>
<td>A List of all LinkSets that require this Load request to succeed. There is a many-to-many relation between Load records and LinkSets. A single import() call can have a large dependency tree, involving many Load records. Many import() calls, if they depend on the same module, can be waiting for a single Load to complete.</td>
</tr>
<tr>
<td>[[Metadata]]</td>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>[[Address]]</td>
<td>Object</td>
<td>undefined</td>
</tr>
<tr>
<td>[[Source]]</td>
<td>String</td>
<td>undefined</td>
</tr>
<tr>
<td>[[Kind]]</td>
<td>One of: undefined, dynamic, declarative</td>
<td>Once the Load reaches the &quot;loaded&quot; state, either declarative or dynamic. If the instantiate hook returned undefined, the module is declarative, and load.([[Body]]) contains a Module parse. Otherwise, the instantiate hook returned a ModuleFactory object and [[Execute]] contains the .execute callable object.</td>
</tr>
<tr>
<td>[[Body]]</td>
<td>undefined or a parse result</td>
<td>If [[Kind]] is declarative, the parse of a Module production. Otherwise undefined.</td>
</tr>
<tr>
<td>[[Execute]]</td>
<td>If [[Kind]] is dynamic, the value of factory.execute</td>
<td>Otherwise undefined.</td>
</tr>
<tr>
<td>[[Dependencies]]</td>
<td>Undefined or List of Records</td>
<td>If [[Status]] is not &quot;loading&quot;, a List of pairs. Each pair consists of two strings: a module name as it appears in a module, import, or export from declaration in load.([[Body]]), and the corresponding normalized module name.</td>
</tr>
<tr>
<td>[[GroupIndex]]</td>
<td>Number</td>
<td>Largest Dependency group count</td>
</tr>
<tr>
<td>[[Exception]]</td>
<td>If [[Status]] is &quot;failed&quot;, the exception value that was thrown, causing the load to fail. Otherwise, null</td>
<td></td>
</tr>
<tr>
<td>[[Module]]</td>
<td>The Module object produced by this load, or undefined.</td>
<td></td>
</tr>
</tbody>
</table>

A LoadRequest object is an ordinary Object, inheriting from Object.prototype with own data properties whose values corresponding certain fields of a corresponding Load Record. A LoadRequest object is created when the value of those fields need to be passed to an ECMAScript function. Every LoadRequest object has name, and metadata properties corresponding to the [[Name]] and [[Metadata]] fields of a Load Record. A LoadRequest object may also have address and source properties corresponding to the [[Address]] and [[Source]] fields of a Load record.
15.2.3.2.1 CreateLoad(name) Abstract Operation

The abstract operation CreateLoad creates and returns a new Load Record. The argument name is either undefined, indicating an anonymous module, or a normalized module name.

The following steps are taken:
1. Let load be a new Load Record.
2. Set load.[[Status]] to "loading".
3. Set load.[[Name]] to name.
4. Set load.[[LinkSets]] to a new empty List.
5. Set load.[[Metadata]] to ObjectCreate(%ObjectPrototype%).
6. Set all other fields of load to undefined.
7. Return load.

15.2.3.2.2 CreateLoadRequestObject(name, metadata, address, source) Abstract Operation

The abstract operation CreateLoadRequestObject performed with arguments name, metadata, and optional arguments address and source returns a new LoadRequest Object. It performs the following steps:

1. Let obj be ObjectCreate(%ObjectPrototype%, ()).
2. Assert: The following operations will never result in abrupt completions.
3. Perform CreateDataProperty (obj, "name", name).
4. Perform CreateDataProperty (obj, "metadata", metadata).
5. If address was passed, then perform CreateDataProperty (obj, "address", address).
6. If source was passed, then perform CreateDataProperty (obj, "source", source).
7. Return obj.

15.2.4 Runtime Semantics: Module Loading

15.2.4.1 LoadModule(loader, name, options) Abstract Operation

The following steps are taken:
1. Assert: loader is a Loader record.
2. Let name be ToString(name).
3. ReturnIfAbrupt(name).
4. Let address be GetOption(options, "address").
5. ReturnIfAbrupt(address).
6. If address is undefined, let step be "locate".
7. Else, let step be "fetch".
8. Let metadata be ObjectCreate(%ObjectPrototype%).

15.2.4.2 RequestLoad(loader, request, refererName, refererAddress) Abstract Operation

The RequestLoad abstract operation normalizes the given module name, request, and returns a Promise object that resolves to the value of a Load object for the given module.

The loader argument is a Loader record.
request is the (non-normalized) name of the module to be imported, as it appears in the import-declaration or as the argument to loader.load() or loader.import().

referrerName and refererAddress provide information about the context of the import() call or import-declaration. This information is passed to all the loader hooks.

If the requested module is already in the loader's module registry, RequestLoad returns a Promise object for a Load with the [[Status]] field set to “linked”. If the requested module is loading or loaded but not yet linked, RequestLoad returns a Promise object for an existing Load object from loader.[[Loads]]. Otherwise, RequestLoad starts loading the module and returns a Promise object for a new Load Record.

The following steps are taken:
1. Let F be a new anonymous function as defined by CallNormalize.
2. Set F's [[Loader]] internal slot to loader.
3. Set F's [[Request]] internal slot to request.
4. Set F's [[RefererName]] internal slot to refererName.
5. Set F's [[RefererAddress]] internal slot to refererAddress.
6. Let p be PromiseNew(F).
7. Let G be a new built-in function as defined by GetOrCreateLoad.
8. Set G's [[Loader]] internal slot to loader.
9. Return PromiseThen(p, G).

15.2.4.2.1 CallNormalize(resolve, reject) Functions

A CallNormalize function is an anonymous built-in function that calls a loader's normalize hook. Each CallNormalize function has internal slots [[Loader]], [[Request]], [[RefererName]], and [[RefererAddress]].

When a CallNormalize function F is called with arguments resolve and reject, the following steps are taken:
1. Let loader be the value of F's [[Loader]] internal slot.
2. Set request be F's [[Request]] internal slot.
3. Set refererName be the value of F's [[RefererName]] internal slot.
4. Set refererAddress be the value of F's [[RefererAddress]] internal slot.
5. Set loaderObj be loader.[[LoaderObj]].
6. Let normalizeHook be Get(loaderObj, "normalize").
7. Let name be the result of calling the [[Call]] internal method of normalizeHook passing loaderObj and (request, refererName, refererAddress) as arguments.
8. ReturnIfAbrupt(name).
9. Return the result of calling the [[Call]] internal method of resolve passing undefined and (name) as arguments.

15.2.4.2.2 GetOrCreateLoad(name) Functions

A GetOrCreateLoad function is an anonymous function that gets or creates a Load Record for a given module name.

Each GetOrCreateLoad function has a [[Loader]] internal slot.

When a GetOrCreateLoad function F is called with argument name, the following steps are taken:
1. Let `loader` be `F`'s `[[Loader]]` internal slot.
2. Let `name` be `ToString(name)`.
3. ReturnIfAbrupt(`name`).
4. Let `modules` be the value of `loader. [[Modules]]`.
5. Repeat for each Record `[[key]], [[value]]` `p` that is an element of `modules`, do
   a. If `SameValue(p. [[key]], name)` is `true`, then
      i. Let `existingModule` be the `[[value]]` field of that Record.
      ii. Let `load` be `CreateLoad(name)`.
      iii. Set `load. [[Status]]` to "linked".
      iv. Set `load. [[Module]]` to `existingModule`.
      v. Return `load`.
6. Repeat for each Record `load` that is an element of `loader. [[Loads]]`, do
   a. If `SameValue(load. [[Name]], name)` is `true`, then
      i. Assert: `load. status` is either "loading" or "loaded".
      ii. Return `load`.
7. Let `load` be `CreateLoad(name)`.
9. Call `ProceedToLocate(loader, load)`.
10. Return `load`.

15.2.4.3 `ProceedToLocate(loader, load, p)` Abstract Operation

The `ProceedToLocate` abstract operation continues the asynchronous loading process at the `locate` hook.

`ProceedToLocate` performs the following steps:
1. Let `p` be `PromiseOf(undefined)`.
2. Let `F` be a new built-in function object as defined in `CallLocate`.
3. Set `F`'s `[[Loader]]` internal slot to `loader`.
4. Set `F`'s `[[Load]]` internal slot to `load`.
5. Let `p` be `PromiseThen(p, F)`.
6. Return `ProceedToFetch(loader, load, p)`.

15.2.4.3.1 `CallLocate` Functions

A `CallLocate` function is an anonymous built-in function that calls the `locate` loader hook. Each `CallLocate` function has `[[Loader]]` and `[[Load]]` internal slots.

When a `CallLocate` function `F` is called, the following steps are taken:
1. Let `loader` be the value of `F`'s `[[Loader]]` internal slot.
2. Let `load` be the value of `F`'s `[[Load]]` internal slot.
3. Let `loaderObj` be `loader. [[LoaderObj]]`.
4. Let `hook` be `Get(loaderObj, "locate")`.
5. ReturnIfAbrupt(`hook`).
6. If `IsCallable(hook)` is `false`, throw a `TypeError` exception.
7. Let `obj` be CreateLoadRequestObject(`load. [[Name]]`, `load. [[Metadata]]`).
8. Return the result of calling the `[[Call]]` internal method of `hook` with `loaderObj` and `obj` as arguments.
15.2.4.4 ProceedToFetch(loader, load, p) Abstract Operation

The ProceedToFetch abstract operation continues the asynchronous loading process at the `fetch` hook by performing the following steps:

1. Let \( F \) be a new built-in function object as defined in CallFetch.
2. Set \( F \)'s \([\text{Loader}]\) internal slot to \( \text{loader} \).
3. Set \( F \)'s \([\text{Load}]\) internal slot to \( \text{load} \).
4. Set \( F \)'s \([\text{AddressPromise}]\) internal slot to \( p \).
5. Let \( p \) be PromiseThen(\( p, F \)).
6. Return ProceedToTranslate(\( \text{loader}, \text{load}, p \)).

15.2.4.4.1 CallFetch(address) Functions

A CallFetch function is an anonymous built-in function that calls the `fetch` loader hook. Each CallFetch function has \([\text{Loader}]\) and \([\text{Load}]\) internal slots.

When a CallFetch function \( F \) is called with argument \( \text{address} \), the following steps are taken:

1. Let \( \text{loader} \) be the value of \( F \)'s \([\text{Loader}]\) internal slot.
2. Let \( \text{load} \) be the value of \( F \)'s \([\text{Load}]\) internal slot.
3. If \( \text{load}.[\text{LinkSets}] \) is an empty List, return \( \text{undefined} \).
4. Set \( \text{load}.[\text{Address}] \) to \( \text{address} \).
5. Let \( \text{loaderObj} \) be \( \text{loader}.[\text{LoaderObj}] \).
6. Let \( \text{hook} \) be Get(\( \text{loaderObj}, \text{"fetch"} \)).
7. ReturnIfAbrupt(\( \text{hook} \)).
8. If IsCallable(\( \text{hook} \)) is false, throw a `TypeError` exception.
9. Let \( \text{obj} \) be CreateLoadRequestObject(\( \text{load}.[\text{Name}], \text{load}.[\text{Metadata}], \text{address} \)).
10. Return the result of calling the \([\text{Call}]\) internal method of \( \text{hook} \) with \( \text{loaderObj} \) and (\( \text{obj} \)) as arguments.

15.2.4.5 ProceedToTranslate(loader, load, p) Abstract Operation

The ProceedToTranslate abstract operation continues the asynchronous loading process at the `translate` hook by performing the following steps:

1. Let \( F \) be a new function object as defined in CallTranslate.
2. Set \( F \)'s \([\text{Loader}]\) internal slot to \( \text{loader} \).
3. Set \( F \)'s \([\text{Load}]\) internal slot to \( \text{load} \).
4. Set \( F \)'s \([\text{AddressPromise}]\) internal slot to \( p \).
5. Let \( p \) be PromiseThen(\( p, F \)).
6. Let \( F \) be a new function object as defined in CallInstantiate.
7. Set \( F \)'s \([\text{Loader}]\) internal slot to \( \text{loader} \).
8. Set \( F \)'s \([\text{Load}]\) internal slot to \( \text{load} \).
9. Return PromiseCatch(\( p, F \)).
15.2.4.5.1 CallTranslate Functions

A CallTranslate function is an anonymous built-in function that calls the translate loader hook. Each CallTranslate function has [[Loader]] and [[Load]] internal slots.

When a CallTranslate function F is called with argument source, the following steps are taken:

1. Let loader be the value of F’s [[Loader]] internal slot.
2. Let load be the value of F’s [[Load]] internal slot.
3. If load.[[LinkSets]] is an empty List, return undefined.
4. Let loaderObj be loader.[[LoaderObj]].
5. Let hook be Get(loaderObj, "translate").
6. ReturnIfAbrupt(hook).
7. If IsCallable(hook) is false, throw a TypeError exception.
8. Let obj be CreateLoadRequestObject(load.[[Name]], load.[[Metadata]], load.[[Address]], source).
9. Return the result of calling the [[Call]] internal method of hook with loaderObj and (obj) as arguments.

15.2.4.5.2 CallInstantiate Functions

A CallInstantiate function is an anonymous built-in function that calls the instantiate loader hook. Each CallInstantiate function has [[Loader]] and [[Load]] internal slots.

When a CallInstantiate function F is called with argument source, the following steps are taken:

1. Let loader be the value of F’s [[Loader]] internal slot.
2. Let load be the value of F’s [[Load]] internal slot.
3. If load.[[LinkSets]] is an empty List, return undefined.
4. Set load.[[Source]] to source.
5. Let loaderObj be loader.[[LoaderObj]].
6. Let hook be Get(loaderObj, "instantiate").
7. ReturnIfAbrupt(hook).
8. If IsCallable(hook) is false, throw a TypeError exception.
9. Let obj be CreateLoadRequestObject(load.[[Name]], load.[[Metadata]], load.[[Address]], source).
10. Return the result of calling the [[Call]] internal method of hook with loaderObj and (obj) as arguments.

15.2.4.5.3 InstantiateSucceeded(instantiateResult) Functions

An InstantiateSucceeded function is an anonymous function that handles the result of the instantiate hook.

Each InstantiateSucceeded function has [[Loader]] and [[Load]] internal slots.

When an InstantiateSucceeded function F is called with argument instantiateResult, the following steps are taken:

1. Let loader be the value of F’s [[Loader]] internal slot.
2. Let load be the value of F’s [[Load]] internal slot.
3. If load.[[LinkSets]] is an empty List, return undefined.
4. If instantiateResult is undefined, then
a. Let body be the result of parsing load.\[\text{[Source]}\], interpreted as UTF-16 encoded Unicode text as described in 6.1.4, using Module as the goal symbol. Throw a SyntaxError exception if the parse fails or if any static semantics errors are detected.
b. Set load.\[\text{[Body]}\] to body.
c. Set load.\[\text{[Kind]}\] to declarative.
d. Let depsList be the ModuleRequests of body.
5. Else if Type(instantiateResult) is Object, then
   a. Let deps be Get(instantiateResult, "deps").
   b. ReturnIfAbrupt(deps).
   c. If deps is undefined, then let depsList be a new empty List.
   d. Else:
      i. Let depsList be IterableToArray(deps).
      ii. ReturnIfAbrupt(depsList).
   e. Let execute be Get(instantiateResult, "execute").
   f. ReturnIfAbrupt(execute).
   g. Set load.\[\text{[[Execute]}\] to execute.
   h. Set load.\[\text{[[Kind]}\] to dynamic.
5. Else,
   a. Throw a TypeError exception.

15.2.4.5.4 LoadFailed Functions

A LoadFailed function is an anonymous function that marks a Load Record as having failed. All LinkSets that depend on the Load also fail.

Each LoadFailed function has a [Load] internal slot.

When a LoadFailed function F is called with argument exc, the following steps are taken:

1. Let load be the value of F’s [Load] internal slot.
2. Assert: load.\[\text{[Status]}\] is “loading”.
3. Set load.\[\text{[Status]}\] to “failed”.
4. Set load.\[\text{[Exception]}\] to exc.
5. Let linkSets be a copy of the List load.\[\text{[LinkSets]}\].
6. For each linkSet in linkSets, in the order in which the LinkSet Records were created,
   a. Call LinkSetFailed(linkSet, exc).
7. Assert: load.\[\text{[LinkSets]}\] is empty.

15.2.4.6 ProcessLoadDependencies(load, loader, depsList) Abstract Operation

The ProcessLoadDependencies abstract operation is called after one module has nearly finished loading. It starts new loads as needed to load the module’s dependencies.

ProcessLoadDependencies also arranges for LoadSucceeded to be called.

The following steps are taken:
1. Let refererName be load.\[\text{[Name]}\].
2. Set load.\[\text{[Dependencies]}\] to a new empty List.
3. Let loadPromises be a new empty List.
4. For each request in depsList, do
15.2.4.6.1 **AddDependencyLoad** (depLoad) Functions

An AddDependencyLoad function is an anonymous function that adds a Load Record for a dependency to any LinkSets associated with the parent Load.

Each AddDependencyLoad function has `[[ParentLoad]]` and `[[Request]]` internal slots.

When an AddDependencyLoad function `F` is called with argument `depLoad`, the following steps are taken:

1. Let `parentLoad` be the value of `F`'s `[[ParentLoad]]` internal slot.
2. Let `request` be the value of `F`'s `[[Request]]` internal slot.
3. Assert: There is no Record in the List `parentLoad`.[[Dependencies]] whose `[[key]]` field is equal to `request`.
4. Append the Record `{[[key]]: request, [[value]]: depLoad.[[Name]]}` to the end of the List `parentLoad`.[[Dependencies]].
5. If `depLoad.[[Status]]` is not "linked", then
   a. Let `linkSets` be a copy of the List `parentLoad`.[[LinkSets]].
   b. For each `linkSet` in `linkSets`, do
      i. Call `AddLoadToLinkSet(linkSet, depLoad)`.  

15.2.4.6.2 **LoadSucceeded** Functions

A LoadSucceeded function is an anonymous function that transitions a Load Record from "loading" to "loaded" and notifies all associated LinkSet Records of the change. This function concludes the loader pipeline. It is called after all a newly loaded module's dependencies are successfully processed.

Each LoadSucceeded function has `[[Load]]` internal slot.

When a LoadSucceeded function `F` is called, the following steps are taken:

1. Let `load` be the value of `F`'s `[[Load]]` internal slot.
2. Assert: `load`.[[Status]] is "loading".
3. Set `load`.[[Status]] to "loaded".
4. Let `linkSets` be a copy of `load`.[[LinkSets]].
5. For each `linkSet` in `linkSets` in List order, do
   a. Call `UpdateLinkSetOnLoad(linkSet, load)`.  

15.2.4.7 **PromiseOfStartLoadPartwayThrough** (step, loader, name, metadata, source, address)

1. Let `F` be a new anonymous function object as defined in AsyncStartLoadPartwayThrough.
2. Let `state` be the Record `{[[Step]],[step], [[Loader]]: loader, [[ModuleName]]: name, [[ModuleMetadata]]: metadata, [[ModuleSource]]: source, [[ModuleAddress]]: address}`.
3. Set $F$’s $\text{[[StepState]]}$ internal slot to $\text{state}$.
4. Return $\text{PromiseNew}(F)$.

15.2.4.7.1 AsyncStartLoadPartwayThrough Functions

An AsyncStartLoadPartwayThrough function is an anonymous function that is used as a Promise executor. When called it creates a new Load Record and populates it with some information provided by the caller, so that loading can proceed from either the locate hook, the fetch hook, or the translate hook. This functionality is used to implement built-in methods like $\text{Reflect.Loader.prototype.load}$, which permits the user to specify both the normalized module name and the address.

Each AsyncStartLoadPartwayThrough function has internal slots $\text{[[StepState]]}$.

When an AsyncStartLoadPartwayThrough function $F$ is called with arguments $\text{resolve}$ and $\text{reject}$, the following steps are taken:

1. Let $\text{state}$ be the value of $F$’s $\text{[[StepState]]}$ internal slot.
2. Let $\text{loader}$ be $\text{state}.\text{[[Loader]]}$.
3. Let $\text{name}$ be $\text{state}.\text{[[ModuleName]]}$.
4. Let $\text{step}$ be $\text{state}.\text{[[Step]]}$.
5. Repeat for each Record $\text{[[key]], [value]]}$ $p$ that is an element of $\text{loader}.\text{[[Modules]]}$, do
   a. If $\text{SameValue}(p.\text{[[key]], name})$ is true, then throw a $\text{TypeError}$ exception.
6. Repeat for element $\text{load}$ of $\text{loader}.\text{[[Modules]]}$, do
   a. If $\text{SameValue}(\text{load}.\text{[[Name]], name})$ is true, then throw a $\text{TypeError}$ exception.
7. Let $\text{load}$ be $\text{CreateLoad}(\text{name})$.
8. Set $\text{load}.\text{[[Metadata]]}$ to $\text{state}.\text{[[ModuleMetadata]]}$.
9. Let linkSet be $\text{CreateLinkSet}(\text{loader}, \text{load})$.
10. Append $\text{load}$ to the end of $\text{loader}.\text{[[Loads]]}$.
11. Call the $\text{[[Call]]}$ internal method of $\text{resolve}$ with arguments $\text{undefined}$ and $(\text{linkSet}.\text{[[Done]]})$.
12. If $\text{step}$ is “locate”,
   a. Call ProceedToLocate($\text{loader}, \text{load}$).
13. Else if $\text{step}$ is “fetch”,
   a. Let $\text{addressPromise}$ be $\text{PromiseOf}(\text{state}.\text{[[ModuleAddress]]})$.
   b. Call ProceedToFetch($\text{loader}, \text{load}, \text{addressPromise}$).
14. Else,
   a. Assert: $\text{step}$ is “translate”.
   b. Set $\text{load}.\text{[[Address]]}$ to $\text{state}.\text{[[ModuleAddress]]}$.
   c. Let $\text{sourcePromise}$ be $\text{PromiseOf}(\text{state}.\text{[[ModuleSource]]})$.
   d. Call ProceedToTranslate($\text{loader}, \text{load}, \text{sourcePromise}$).

15.2.5 Runtime Semantics: Module Linking

15.2.5.1 ModuleLinkage Record

A ModuleLinkage Record contains the state needed to link a specific module.

Each ModuleLinkage Record has the fields defined in Table 37.
Table 37 — ModuleLinkage Record Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Body]]</td>
<td>a parse result</td>
<td>The parse of a Module production</td>
</tr>
<tr>
<td>[[BoundNames]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[[KnownExportEntries]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[[KnownExportEntries]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[[ExportDefinitions]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[[Exports]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[[Dependencies]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[[UnlinkedDependencies]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[[ImportedEntries]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[[ImportedDefinitions]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[[Evaluated]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[[LinkErrors]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[[Environment]]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

15.2.5.1.1 CreateModuleLinkageRecord (loader, body) Abstract Operation

The abstract operation CreateModuleLinkageRecord with arguments `loader` and `body` performs the following steps:

1. Assert: `body` is a ModuleBody parse.
2. Let `M` be a new ModuleLinkage record.
3. Set `M.[[Body]]` to `body`.
4. Set `M.[[BoundNames]]` to DeclaredNames of `body`.
5. Set `M.[[KnownExportEntries]]` to KnownExportEntries of `body`.
6. Set `M.[[UnknownExportEntries]]` to UnknownExportEntries of `body`.
7. Set `M.[[ExportDefinitions]]` to `undefined`.
8. Set `M.[[Exports]]` to `undefined`.
9. Set `M.[[Dependencies]]` to `undefined`.
10. Set `M.[[UnlinkedDependencies]]` to `undefined`.
11. Set `M.[[ImportedEntries]]` to ImportEntries of `body`.
12. Set `M.[[ImportDefinitions]]` to `undefined`.
13. Set `M.[[LinkErrors]]` to a new empty List.
14. Let `realm` be `loader`.[[Realm]]
15. Let `globalEnv` be `realm`.[[globalEnv]]
16. Let `env` be NewModuleEnvironment(`globalEnv`).
17. Set `M.[[Environment]]` to `env`.
18. Return `M`.

15.2.5.1.2 LookupExport (M, exportName)

The abstract operation LookupExport with arguments `M` and `exportName` performs the following:

1. If `M.[[Exports]]` does not contain a record `export` such that `export.[[ExportName]]` is equal to `exportName`, then return `undefined`.
2. Let `export` be the record in `M.[[Exports]]` such that `export.[[ExportName]]` is equal to `exportName`.
3. Return `export.[[Binding]]`.

Commented [AWB2674]: TODO: need to fill in table.

Commented [AWB2275]: Not currently referenced
15.2.5.1.3 `LookupModuleDependency (M, requestName)`

The abstract operation `LookupModuleDependency` with arguments `M` and `requestName` performs the following steps:

1. Assert: `M` is a ModuleLinkage Record.
2. If `requestName` is `null` then return `M`.
3. Let `pair` be the record in `M.[[Dependencies]]` such that `pair.[[Key]]` is equal to `requestName`.
4. Return `pair.[[Module]]`.

15.2.5.2 LinkSet Records

A LinkSet Record represents a call to `loader.define()`, `.load()`, `.module()`, or `.import()`.

Each LinkSet Record has the fields defined in Table 38.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[Loader]]</code></td>
<td>Loader Record</td>
<td>The Loader record that created this LinkSet.</td>
</tr>
<tr>
<td><code>[[Loads]]</code></td>
<td>List of Load Record</td>
<td>A List of the Load Records that must finish loading before the modules can be linked and evaluated.</td>
</tr>
<tr>
<td><code>[[Done]]</code></td>
<td>Promise Object</td>
<td>The Promise that becomes fulfilled when all dependencies are loaded and linked together.</td>
</tr>
<tr>
<td><code>[[Resolve]]</code></td>
<td>Function Object</td>
<td>Function used to resolve <code>[[Done]]</code>.</td>
</tr>
<tr>
<td><code>[[Reject]]</code></td>
<td>Function Object</td>
<td>Function used to reject <code>[[Done]]</code>.</td>
</tr>
</tbody>
</table>

15.2.5.2.1 `CreateLinkSet(loader, startingLoad)` Abstract Operation

The `CreateLinkSet` abstract operation creates a new LinkSet record by performing the following steps:

1. Assert: `loader` is a Loader Record.
2. Let `promiseCapability` be `PromiseBuiltInCapability()`.  
3. Return `IfAbrupt(promiseCapability)`. 
4. Let `linkSet` be `LinkSet ( loader, [[Loads]]: (), [[Done]]: promiseCapability.[[Promise]], [[Resolve]]: promiseCapability.[[Resolve]], [[Reject]]: promiseCapability.[[Reject]] )`. 
5. Perform `AddLoadToLinkSet(linkSet, startingLoad)`. 
6. Return `linkSet`.

15.2.5.2.2 `AddLoadToLinkSet(linkSet, load)` Abstract Operation

The `AddLoadToLinkSet` abstract operation associates a LinkSet Record with a Load Record and each of its currently known dependencies, indicating that the LinkSet cannot be linked until those Loads have finished successfully.

The following steps are taken:

1. Assert: `load.[[Status]]` is either "loading" or "loaded".
2. Let `loader` be `linkSet.[[Loader]]`.
3. If `load` is not already an element of the List `linkSet.[[Loads]]`, 
   a. Append `load` to the end of the List `linkSet.[[Loads]]`. 

b. Append `linkSet` to the end of the List `load.[[LinkSets]]`.

c. If `load.[[Status]]` is "loaded", then
   i. Repeat for each `r` that is a Record `[[[[Name]], [[NormalizedName]]]]` in `load.[[Dependencies]]`,
      1. If there is no element of `loader.[[Modules]]` whose `[[key]]` field is equal to `name`
         a. If there is an element of `loader.[[Loads]]` whose `[[Name]]` field is equal to `name`,
            i. Let `depLoad` be that Load Record.
            ii. Perform `AddLoadToLinkSet(linkSet, depLoad)`.

15.2.5.2.3 UpdateLinkSetOnLoad(linkSet, load) Abstract Operation

The UpdateLinkSetOnLoad abstract operation is called immediately after a Load successfully finishes, after starting Loads for any dependencies that were not already loading, loaded, or in the module registry.

This operation determines whether `linkSet` is ready to link, and if so, calls Link.

The following steps are taken:

1. Assert: `load` is an element of `linkSet.[[Loads]]`.
2. Assert: `load.[[Status]]` is either "loaded" or "linked".
3. Repeat for each `element` in `linkSet.[[Loads]]`,
   a. If `element.[[Status]]` is "loading", then return.
4. Assert: All Loads in `linkSet.[[Loads]]` have finished loading.
5. Let `startingLoad` be the first element of the List `linkSet.[[Loads]]`.
6. Let `status` be `Link(linkSet.[[Loads]], linkSet.[[Loader]])`.
7. If `status` is an abrupt completion, then
   a. Return `LinkSetFailed(linkSet, status.[[value]])`.
8. Assert: `linkSet.[[Loads]]` is an empty List.
9. Call the `[[Call]]` internal method of `linkSet.[[Resolve]]` passing `undefined` and `(startingLoad)` as arguments.
10. Assert: The call performed by step 9 completed normally.

15.2.5.2.4 LinkSetFailed(linkSet, exc) Abstract Operation

The LinkSetFailed abstract operation is called when a LinkSet fails. It detaches the given LinkSet Record from all Load Records and rejects the `linkSet.[[Done]]` Promise.

The following steps are taken:

1. Let `loader` be `linkSet.[[Loader]]`.
2. Let `loads` be a copy of the List `linkSet.[[Loads]]`.
3. For each `load` in `loads`,
   a. Assert: `linkSet` is an element of the List `load.[[LinkSets]]`.
   b. Remove `linkSet` from the List `load.[[LinkSets]]`.
   c. If `load.[[LinkSets]]` is empty and `load` is an element of `loader.[[Loads]]`, then
      i. Remove `load` from the List `loader.[[Loads]]`.
4. Return the result of calling `[[Call]]` internal method of `linkSet.[[Reject]]` passing `undefined` and `(exc)` as arguments.
5. Assert: The call performed by step 4 completed normally.
15.2.5.2.5 FinishLoad(loader, load) Abstract Operation

The FinishLoad Abstract Operation removes a completed Load Record from all LinkSets and commits the newly loaded Module to the registry. It performs the following steps:

1. Let name be load.\[[\text{Name}]\].
2. If name is not undefined, then
   a. Assert: There is no Record \([[key]], [[value]]\) \(p\) that is an element of loader.\[[\text{Modules}]\], such that SameValue(\(p.\[[\text{Name}]\], load.\[[\text{Name}]\]) is true.
   b. Append the Record \([[key]: load.\[[\text{Name}]\], [[value]: load.\[[\text{Module}]\]]\) as the last element of loader.\[[\text{Modules}]\].
3. If load is an element of the List loader.\[[\text{Loads}]\], then
   a. Remove load from the List loader.\[[\text{Loads}]\].
4. For each linkSet in load.\[[\text{LinkSets}]\],
   a. Remove load from linkSet.\[[\text{Loads}]\].
5. Remove all elements from the List load.\[[\text{LinkSets}]\].

15.2.5.3 Module Linking Groups

A load record \(load_1\) has a linkage dependency on a load record \(load_2\) if \(load_2\) is contained in \(load_1.\[[\text{UnlinkedDependencies}]\]\) or there exists a load record \(load\) in \(load_1.\[[\text{UnlinkedDependencies}]\]\) such that \(load\) has a linkage dependency on \(load_2\).

The linkage graph of a List, list, of load records is the set of load records \(load\) such that some load record in list has a linkage dependency on \(load\).

A dependency chain from \(load_1\) to \(load_2\) is a List of load records demonstrating the transitive linkage dependency from \(load_1\) to \(load_2\).

A dependency cycle is a dependency chain whose first and last elements' \([[\text{Name}]\] fields have the same value.

A dependency chain is cyclic if it contains a subsequence that is a dependency cycle. A dependency chain is acyclic if it is not cyclic.

A dependency chain is mixed if there are two elements with distinct values for their \([[\text{Kind}]\] fields. A dependency group transition of kind \(kind\) is a two-element subsequence \(load_1, load_2\) of a dependency chain such that \(load_1.\[[\text{Kind}]\]\) is not equal to \(kind\) and \(load_2.\[[\text{Kind}]\]\) is equal to \(kind\).

The dependency group count of a dependency chain with first element \(load_1\) is the number of distinct dependency group transitions of kind \(load_1.\[[\text{Kind}]\]\).

15.2.5.3.1 LinkageGroups ( start )

The abstract operation LinkageGroups with argument \(start\) performs the following steps:

1. Assert: \(start\) is a List of LinkSet Records.
2. Let \(G\) be the linkage graph of \(start\).
3. If there are any mixed dependency cycles in \(G\), throw a SyntaxError exception.
4. For each \(load\) in \(G\), do
   a. Let \(n\) be the largest dependency group count of all acyclic dependency chains in \(G\) starting from \(load\).
   b. Set \(load.\[[\text{GroupIndex}]\]\) to \(n\).
5. Let $declarativeGroupCount$ be the largest [[GroupIndex]] of any load in $G$ such that load.[[Kind]] is declarative.
6. Let $declarativeGroups$ be a new List of length $declarativeGroupCount$ where each element is a new empty List.
7. Let $dynamicGroupCount$ be the largest [[GroupIndex]] of any load in $G$ such that load.[[Kind]] is dynamic.
8. Let $dynamicGroups$ be a new List of length $dynamicGroupCount$ where each element is a new empty List.
9. Let $visited$ be a new empty List.
10. For each load in $start$, do
    a. Perform $BuildLinkageGroups(load, declarativeGroups, dynamicGroups, visited)$.
11. If any load in the first element of $declarativeGroups$ has a dependency on a load record of [[Kind]] dynamic, then
    a. Let $groups$ be a List constructed by interleaving the elements of $dynamicGroups$ and $declarativeGroups$, starting with the former.
12. Else, a. Let $groups$ be a List constructed by interleaving the elements of $declarativeGroups$ and $dynamicGroups$, starting with the former.
13. Return $groups$.

15.2.5.3.2 $BuildLinkageGroups (load, declarativeGroups, dynamicGroups, visited)$

The abstract operation $BuildLinkageGroups$ with arguments $load$, $declarativeGroups$, and $dynamicGroups$ performs the following steps:

1. If $visited$ contains an element whose [[Name]] is equal to $load$.[[Name]], then return.
2. Add $load$ to $visited$.
3. For each dep of $load$.[[UnlinkedDependencies]], do
   a. Call the $BuildLinkageGroups$ abstract operation passing $dep$, $declarativeGroups$, $dynamicGroups$, and $visited$ as arguments.
4. Let $i$ be $load$.[[GroupIndex]].
5. If $load$.[[Kind]] is declarative let $groups$ be $declarativeGroups$; otherwise let $groups$ be $dynamicGroups$.
6. Let $group$ be the $i$th element of $groups$.
7. Add $load$ to $group$.

15.2.5.4 $Link (start, loader)$

The abstract operation $Link$ with argument $start$ performs the following steps:

1. Let $groups$ be $LinkageGroups(start)$.
2. For each group in $groups$:
   a. If the [[Kind]] of each element of $group$ is declarative, then perform $LinkDeclarativeModules(group, loader)$.
   b. Else, perform $LinkDynamicModules(group, loader)$.

15.2.5.5 $LinkDeclarativeModules (loads, loader)$

The abstract operation $LinkDeclarativeModules$ with arguments $loads$ and $loader$ performs the following steps:

1. Let $unlinked$ be a new empty List.
2. For each load in $loads$, do
a. If load.[[Status]] is not linked, then
   i. Let module be CreateModuleLinkageRecord (loader, load.[[Body]]).
   ii. Let pair be the record {[[Module]]: module, [[Load]]: load}.
   iii. Add pair to unlinked.
3. For each pair in unlinked, do
   a. Let resolvedDeps be a new empty List.
   b. Let unlinkedDeps be a new empty List.
   c. For each element dep in pair.[[Load]].[[Dependencies]], do
      i. Let requestName be dep.[[Key]].
      ii. Let normalizedName be dep.[[Value]].
      iii. If loads contains a record load such that SameValue(load.[[Name]], normalizedName) is true, then
         1. If load.[[Status]] is linked, then
            a. Let resolvedDep be the record {[[Key]]: requestName, [[Value]]: load.[[Module]]}.
            b. Add resolvedDep to resolvedDeps.
         2. Else, a. Let otherPair be the record in unlinked such that
            SameValue(otherPair.[[Load]].[[Name]], normalizedName) is true.
            b. Add the record {[[Key]]: requestName, [[Value]]: otherPair.[[Module]]} to resolvedDeps.
            c. Add otherPair.[[Load]] to unlinkedDeps.
      iv. Else, a. Let module be LoaderRegistryLookup (loader, normalizedName).
         1. If module is null then
            a. Let error be a new ReferenceError exception.
            b. Add error to pair.[[Module]].[[LinkErrors]].
         2. Else, add the record {[[Key]]: requestName, [[Value]]: module} to resolvedDeps.
      d. Set pair.[[[Module]].[[Dependencies]]] to resolvedDeps.
      e. Set pair.[[[Module]].[[UnlinkedDependencies]]] to unlinkedDeps.
   d. For each pair in unlinked, do
      a. Perform ResolveExportEntries (pair.[[Module]]. ( )).
      b. Perform ResolveExports (pair.[[Module]]).
   5. For each pair in unlinked, do
      a. Perform ResolveImportEntries (pair.[[Module]])..
      b. Perform LinkImports (pair.[[Module]])..
   6. If there exists a pair in unlinked such that pair.[[[Module]].[[LinkErrors]]] is not empty, choose one of the link errors and throw it.
7. For each pair in unlinked, do
   a. Set pair.[[[Load]].[[Module]]] to pair.[[Module]].
   b. Set pair.[[[Load]].[[Status]]] to linked.
   c. Let r be FinishLoad (loader, pair.[[Load]]).
   d. ReturnIfAbrupt (r).

15.2.5.5.1 LinkImports ( M )

The abstract operation LinkImports with argument M performs the following steps:
1. Let envRec be M.[[Environment]].
2. Let defs be M.[[ImportDefinitions]].
3. For each def in defs, do
   a. If def.[[ImportName]] is module, then the following steps are taken:
i. Call the CreateImmutableBinding concrete method of `envRec` passing `def.[[LocalName]]` as the argument.
ii. Call the InitializeImmutableBinding concrete method of `envRec` passing `def.[[LocalName]]` and `def.[[Module]]` as the arguments.

b. Otherwise, the following steps are taken:
   i. Let `binding` be `ResolveExport(def.[[Module]], def.[[ImportName]], ( ))`.
   ii. If `binding` is `undefined`, then the following steps are taken:
       1. Let `error` be a new Reference Error.
       2. Add `error` to `M.[[LinkErrors]]`.
   iii. Otherwise, call the CreateImportBinding concrete method of `envRec` passing `def.[[LocalName]]` and `binding` as the arguments.

15.2.5.6 `LinkDynamicModules (loads, loader)`

The abstract operation `LinkDynamicModules` with arguments `loads` and `loader` performs the following steps:

1. For each `load` in `loads`, do
   a. Let `exec` be `load.[[Execute]]`.
   b. Let `module` be the result of calling the `[[Call]]` internal method of `exec` with `undefined` as the `this` value and with no arguments.
   c. ReturnIfAbrupt(`module`).
   d. If `module` does not have all the internal data properties of a Module Object, then throw a `TypeError` exception.
   e. Set `load.[[Module]]` to `module`.
   f. Set `load.[[Status]]` to `linked`.
   g. Let `r` be `FinishLoad(loader, load)`.
   h. ReturnIfAbrupt(`r`).

15.2.5.7 `ResolveExportEntries (M, visited)`

The abstract operation `ResolveExportEntries` with arguments `M` and `visited` performs the following steps:

1. If `M.[[ExportDefinitions]]` is not `undefined`, then return `M.[[ExportDefinitions]]`.
2. Let `defs` be a new empty list.
3. Let `boundNames` be `M.[[BoundNames]]`.
4. For each `entry` in `M.[[KnownExportEntries]]`, do
   a. Let `modReq` be `entry.[[ModuleRequest]]`.
   b. Let `otherMod` be `LookupModuleDependency(M, modReq)`.
   c. If `entry.[[Module]]` is `null` and `entry.[[LocalName]]` is `null` and `boundNames` does not contain `entry.[[LocalName]]`, then the following steps are taken:
      i. Let `error` be a new Reference Error.
      ii. Add `error` to `M.[[LinkErrors]]`.
   d. Add the record `{[[Module]]: otherMod, [[ImportName]]: entry.[[ImportName]], [[LocalName]]: entry.[[LocalName]], [[ExportName]]: entry.[[ExportName]], [[Explicit]]: true}` to `defs`.
5. For each `modReq` in `M.[[UnknownExportEntries]]`, do
   a. Let `otherMod` be `LookupModuleDependency(M, modReq)`.
   b. If `otherMod` is in `visited`, then the following steps are taken:
      i. Let `error` be a new Syntax Error.
      ii. Add `error` to `M.[[LinkErrors]]`.
   c. Otherwise the following steps are taken:
      i. Add `otherMod` to `visited`.
      ii. Let `otherDefs` be `ResolveExportEntries(otherMod, visited)`.
iii. For each `def` of `otherDefs`, do
   1. Add the record `[[Module]]: otherMod, [[ModuleName]]: def.[[ExportName]], [[LocalName]]: null, [[ExportName]]: def.[[ExportName]], [[Explicit]]: false` to `defs`.
   6. Set `M.[[ExportDefinitions]]` to `defs`.
   7. Return `defs`.

15.2.5.8 ResolveExports (M)

The abstract operation `ResolveExports` with argument `M` performs the following steps:

1. For each `def` in `M.[[ExportDefinitions]]`, do
   a. Perform `ResolveExport(M, def.[[ExportName]], visited)`.

15.2.5.9 ResolveExport (M, exportName, visited)

The abstract operation `ResolveExport` with arguments `M`, `exportName`, and `visited` performs the following steps:

1. Let `exports` be `M.[[Exports]]`.
2. If `exports` has a record `export` such that `export.[[ExportName]]` is equal to `exportName`, return `export.[[Binding]]`.
3. Let `ref` be `[[Module]]: M, [[ModuleName]]: exportName`.
4. If `visited` contains a record equal to `ref` then the following steps are taken:
   a. Let `error` be a new Syntax Error.
   b. Add `error` to `M.[[LinkErrors]]`.
   c. Return `error`.
5. Let `defs` be `M.[[ExportDefinitions]]`.
6. Let `overlappingDefs` be the List of records `def` in `defs` such that `def.[[ExportName]]` is equal to `exportName`.
7. If `overlappingDefs` is empty, then the following steps are taken:
   a. Let `error` be a new Reference Error.
   b. Add `error` to `M.[[LinkErrors]]`.
   c. Return `error`.
8. If `overlappingDefs` has more than one record `def` such that `def.[[Explicit]]` is `true`, or if it has length greater than 1 but contains no records `def` such that `def.[[Explicit]]` is `true`, then the following steps are taken:
   a. Let `error` be a new Syntax Error.
   b. Add `error` to `M.[[LinkErrors]]`.
   c. Return `error`.
9. Let `def` be the unique record in `overlappingDefs` such that `def.[[Explicit]]` is `true`, or if there is no such record let `def` be the unique record in `overlappingDefs`.
10. If `def.[[LocalName]]` is not `null`, then the following steps are taken:
    a. Let `binding` be the record `[[Module]]: M, [[ModuleName]]: def.[[LocalName]]`.
    b. Let `export` be the record `[[ModuleName]]: exportName, [[Binding]]: binding`.
    c. Add `export` to `exports`.
    d. Return `binding`.
11. Add `ref` to `visited`.
12. Let `binding` be `ResolveExport(def.[[Module]], def.[[ModuleName]], visited)`.
13. Return `binding`.

15.2.5.10 ResolveImportEntries (M)

The abstract operation `ResolveImportEntries` called with argument `M` performs the following steps:
1. Let entries be $M.\llbracket \text{ImportEntries}\rrbracket$.
2. Let defs be a new empty List.
3. For each entry in entries, do
   a. Let $\text{modReq} \equiv \text{entry}.\llbracket \text{ModuleRequest}\rrbracket$.
   b. Let $\text{otherMod} \equiv \text{LookupModuleDependency}(M, \text{modReq})$.
   c. Add the record \{\text{Module} : $\text{otherMod}$, \text{ImportName} : entry.\llbracket \text{ImportName}\rrbracket$, \text{LocalName} : entry.\llbracket \text{LocalName}\rrbracket\} to defs.
4. Return defs.

15.2.6 Runtime Semantics: Module Evaluation

Module bodies are evaluated on demand, as late as possible. The loader uses the function $\text{EnsureEvaluated}$ defined below, to run scripts. The loader always calls $\text{EnsureEvaluated}$ before returning a Module object to user code.

There is one way a module can be exposed to script before its body has been evaluated. In the case of an import cycle, whichever module is evaluated first can observe the others before they are evaluated. Simply put, we have to start somewhere: one of the modules in the cycle must run before the others.

15.2.6.1 $\text{EvaluateLoadedModule}(\text{load})$ Functions

An $\text{EvaluateLoadedModule}$ function is an anonymous built-in function that is used by Reflect.Loader.prototype.module and Reflect.Loader.prototype.import to ensure that a module has been evaluated before it is passed to script code.

Each $\text{EvaluateLoadedModule}$ function has a \[\text{Loader}\] internal slot.

When a $\text{EvaluateLoadedModule}$ function $F$ is called with argument $\text{load}$, the following steps are taken:

1. Let $\text{loader} \equiv F.\llbracket \text{Loader}\rrbracket$.
2. Assert: $\text{load.\llbracket \text{Status}\rrbracket}$ is "linked".
3. Let $\text{module} \equiv \text{load.\llbracket \text{Module}\rrbracket}$.
4. Let result be $\text{EnsureEvaluated}(\text{module}, () \text{, loader})$.
5. ReturnIfAbrupt(result).
6. Return $\text{module}$.

15.2.6.2 $\text{EnsureEvaluated}(\text{mod}, \text{seen}, \text{loader})$ Abstract Operation

The abstract operation $\text{EnsureEvaluated}$ walks the dependency graph of the module $\text{mod}$, evaluating any module bodies that have not already been evaluated (including, finally, $\text{mod}$ itself). Modules are evaluated in depth-first, left-to-right, post order, stopping at cycles.

$\text{mod}$ and its dependencies must already be linked.

The List $\text{seen}$ is used to detect cycles. $\text{mod}$ must not already be in the List $\text{seen}$.

On success, $\text{mod}$ and all its dependencies, transitively, will have started to evaluate exactly once.

$\text{EnsureEvaluated}$ performs the following steps:

1. If $\text{mod.\llbracket \text{Evaluated}\rrbracket}$ is true, return \text{undefined}.
2. Append $\text{mod}$ as the last element of $\text{seen}$.
3. TODO: Create the module environment for $\text{mod}$.

Commented [AWB2279]: Or some other sort of flag? Currently we are loose the normal completion value produced by a module..

Commented [AWB2380]: TODO
4. Let `deps` be `mod.[[Dependencies]]`.
5. For each pair in `deps`, in List order,
   a. Let `dep` be `pair.[[value]]`.
   b. If `dep` is not an element of `seen`, then
      i. Call `EnsureEvaluated` with the arguments `dep`, `seen`, and `loader`.
6. If `mod.[[Evaluated]]` is `true`, return `undefined`.
7. Set `mod.[[Evaluated]]` to `true`.
8. If `mod.[[Evaluated]]` is `undefined`, then return `undefined`.
9. Let `status` be `ModuleDeclarationInstantiation(mod.[[Body]], mod.[[Environment]])`.
10. Let `initContext` be a new ECMAScript code execution context.
11. Set `initContext`'s Realm to `loader.[[Realm]]`.
13. If there is a currently running execution context, suspend it.
14. Push `initContext` onto the execution context stack; `initContext` is now the running execution context.
15. Let `r` be the result of evaluating `mod.[[Body]]`.
16. Suspend `initContext` and remove it from the execution context stack.
17. Resume the context, if any, that is now on the top of the execution context stack as the running execution context.
18. Return `r`.

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. The presence of an early error prevents the evaluation of the construct. An implementation must report early errors in a Script as part of the ScriptEvaluationJob for that Script. Early errors in a Module are reported at the point when the Module would be evaluated and the Module is never initialized. Early errors in eval code are reported at the time eval is called and prevent evaluation of the eval code. All errors that are not early errors are runtime errors.

An implementation must report as an early error any occurrence of a condition that is listed in a “Static Semantics: Early Errors” subclause of this specification.

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, Module 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.
- Except as specified below (16.1), 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.

### 16.1 Forbidden Extensions

An implementation must not extend this specification in the following ways:

- Other than as defined in this specification, ECMAScript Function objects defined using syntactic constructors in strict code must not be created with own properties named "caller" or "arguments" other than those that are created by applying the `AddRestrictedFunctionProperties` abstract operation (9.2.8) to the function. Such own properties also may not be created for function objects created in non-code code by an `ArrowFunction`, `MethodDefinition`, `GeneratorDeclaration`, `GeneratorExpression`, `ClassDeclaration`, or `ClassExpression`. Built-in functions: Strict mode functions created using the `Function` or `Generator` constructors and functions created using the `bind` and `call` methods also must not be created with such own properties.
- When processing strict mode code, the syntax of `NumericLiteral` must not be extended to include `LegacyOctalIntegerLiteral` as defined in B.1.1.
  - `TemplateCharacter` (11.8.6) must not be extended to include `OctalEscapeSequence` as defined in B.1.2.
- When processing strict mode code, the extensions defined in B.3.3, and B.3.4 must not be provided.

### 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 `Function` 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. Such missing arguments are considered to be "not present" and may be identified in that manner by specification algorithms.

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. However, rest parameters shown using the form “...name” are not included in the default argument count.

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 symbol value.

Unless otherwise specified, the `name` property of a built-in Function object, if it exists, 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 is function is described the get accessor 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 6.1.6). 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 6.1.4, for the goal symbol Script. If the parse fails or any early errors are detected, throw a SyntaxError exception (but see also clause 16).
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 `ScriptEvaluation` for `script` with arguments `evalRealm` and `true`.
10. If `direct` is `true`, `strictScript` is `true`, strictCaller is `false`, and `ctx`'s LexicalEnvironment is the same as `evalRealm`.[[globalEnv]], then
    a. Return the result of `ScriptEvaluation` 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 `NewDeclarativeEnvironment`(lexEnv).
    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. ReturnIfAbrupt(`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. Suspend `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 will always have an environment record as its base value and its referenced name is "eval".
- The result of calling the abstract operation GetValue with that Reference as the argument is the standard built-in function defined in 18.2.1.
18.2.1.2 Eval Declaration Instantiation

18.2.2 isFinite (number)

Returns \texttt{false} if the argument coerces to \texttt{NaN}, \( +\infty \), or \( -\infty \), and otherwise returns \texttt{true}.

\begin{enumerate}
  \item Let \texttt{num} be \texttt{ToNumber(number)}.
  \item ReturnIfAbrupt(\texttt{num}).
  \item If \texttt{num} is \texttt{NaN}, \( +\infty \), or \( -\infty \), return \texttt{false}.
  \item Otherwise, return \texttt{true}.
\end{enumerate}

18.2.3 isNaN (number)

Returns \texttt{true} if the argument coerces to \texttt{NaN}, and otherwise returns \texttt{false}.

\begin{enumerate}
  \item Let \texttt{num} be \texttt{ToNumber(number)}.
  \item ReturnIfAbrupt(\texttt{num}).
  \item If \texttt{num} is \texttt{NaN}, return \texttt{true}.
  \item Otherwise, return \texttt{false}.
\end{enumerate}

\textbf{NOTE} A reliable way for ECMAScript code to test if a value \( x \) is a \texttt{NaN} is an expression of the form \( x \neq x \). The result will be \texttt{true} if and only if \( x \) is a \texttt{NaN}.

18.2.4 parseFloat (string)

The \texttt{parseFloat} function produces a Number value dictated by interpretation of the contents of the \texttt{string} argument as a decimal literal.

When the \texttt{parseFloat} function is called, the following steps are taken:

\begin{enumerate}
  \item Let \texttt{inputString} be \texttt{ToString(string)}.
  \item ReturnIfAbrupt(\texttt{inputString}).
  \item Let \texttt{trimmedString} be a substring of \texttt{inputString} consisting of the leftmost code unit that is not a \texttt{StrWhiteSpaceChar} and all code units to the right of that code unit. (In other words, remove leading white space.) If \texttt{inputString} does not contain any such code units, let \texttt{trimmedString} be the empty string.
  \item If neither \texttt{trimmedString} nor any prefix of \texttt{trimmedString} satisfies the syntax of a \texttt{StrDecimalLiteral} (see 7.1.3.1), return \texttt{NaN}.
  \item Let \texttt{numberString} be the longest prefix of \texttt{trimmedString}, which might be \texttt{trimmedString} itself, that satisfies the syntax of a \texttt{StrDecimalLiteral}.
  \item Return the Number value for the MV of \texttt{numberString}.
\end{enumerate}

\textbf{NOTE} \texttt{parseFloat} may interpret only a leading portion of \texttt{string} as a Number value; it ignores any code units that cannot be interpreted as part of the notation of an decimal literal, and no indication is given that any such code units were ignored.

18.2.5 parseInt (string , radix)

The \texttt{parseInt} function produces an integer value dictated by interpretation of the contents of the \texttt{string} argument according to the specified \texttt{radix}. Leading white space in \texttt{string} is ignored. If \texttt{radix} is \texttt{undefined} or 0, it is assumed to be 10 except when the number begins with the code unit pairs \texttt{0x} or \texttt{0X}, in which case a radix of 16 is assumed. If \texttt{radix} is 16, the number may also optionally begin with the code unit pairs \texttt{0x} or \texttt{0X}.
When the `parseInt` function is called, the following steps are taken:

1. Let `inputString` be `ToString(string)`. 
2. ReturnIfAbrupt(`inputString`).
3. Let `S` be a newly created substring of `inputString` consisting of the first code unit that is not a `StrWhiteSpaceChar` and all code units following that code unit. (In other words, remove leading white space.) If `inputString` does not contain any such code unit, let `S` be the empty string.
4. Let `sign` be 1.
5. If `S` is not empty and the first code unit of `S` is `U+002D` (HYPHEN-MINUS), let `sign` be `-1`.
6. If `S` is not empty and the first code unit of `S` is `U+002B` (PLUS SIGN) or `U+002D` (HYPHEN-MINUS), then remove the first code unit 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 code units of `S` are either “0x” or “0X”, then remove the first two code units from `S` and let `R` = 16.
13. If `S` contains any code units that is not a radix-`R` digit, then let `Z` be the substring of `S` consisting of all code units before the first such code unit; otherwise, let `Z` be `S`.
14. If `Z` is empty, return `NaN`.
15. Let `mathInt` be the mathematical integer value that is represented by `Z` in radix-`R` notation, using the letters `A`-`Z` and `a`-`z` for digits with values 10 through 35. (However, if `R` is 10 and `Z` contains more than 20 significant digits, every significant digit after the 20th may be replaced by a 0 digit, at the option of the implementation; and if `R` is not 2, 4, 8, 10, 16, or 32, then `mathInt` may be an implementation-dependent approximation to the mathematical integer value that is represented by `Z` in radix-`R` notation.)
16. Let `number` be the Number value for `mathInt`.
17. Return `sign` × `number`.

**NOTE**: `parseInt` may interpret only a leading portion of `string` as an integer value; it ignores any code units that cannot be interpreted as part of the notation of an integer, and no indication is given that any such code units were ignored.

### 18.2.6 URI Handling Functions

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 18.2.6.5.

**NOTE**: 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 italicized names represent components and ";", "/", "?" and "@" are reserved for use as separators. The **encodeURI** and **decodeURI** functions are intended to work with complete URIs; they assume that any reserved code units 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 code units represent text and so must be encoded so that they are not interpreted as reserved code units when the component is part of a complete URI.

The following lexical grammar specifies the form of encoded URIs.

**Syntax**

\[
\text{uri} ::= \text{uriCharacters}_{opt} \\
\text{uriCharacters} ::= \text{uriCharacter} \text{uriCharacters}_{opt} \\
\text{uriCharacter} ::= \text{uriReserved} \text{uriUnescaped} \text{uriEscaped} \\
\text{uriReserved} ::= \text{one of} \\
\; ; / ? : @ & = + $ , \\
\text{uriUnescaped} ::= \text{uriAlpha} \text{DecimalDigit} \text{uriMark} \\
\text{uriEscaped} ::= \% \text{HexDigit HexDigit} \\
\text{uriAlpha} ::= \text{one of} \\
\; a b c d e f g h i j k l m n o p q r s t u v w x y z \\
\; A B C D E F G H I J K L M N O P Q R S T U V W X Y Z \\
\text{uriMark} ::= \text{one of} \\
\; - . _ ! ~ * ' ( ) \\
\]

**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 code unit to be included in a URI is not listed above or is not intended to have the special meaning sometimes given to the reserved code units, that code unit must be encoded. The code unit 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 \texttt{string} and \texttt{unescapedSet}.

1. Let \texttt{strLen} be the number of code units in \texttt{string}.
2. Let \texttt{R} be the empty String.
3. Let \texttt{k} be 0.
4. Repeat
   a. If \texttt{k} equals \texttt{strLen}, return \texttt{R}.
   b. Let \texttt{C} be the code unit at position \texttt{k} within \texttt{string}.
   c. If \texttt{C} is in \texttt{unescapedSet}, then
      i. Let \texttt{S} be a String containing only the code unit \texttt{C}.
      ii. Let \texttt{R} be a new String value computed by concatenating the previous value of \texttt{R} and \texttt{S}.
   d. Else \texttt{C} is not in \texttt{unescapedSet},
      i. If the code unit value of \texttt{C} is not less than 0xDC00 and not greater than 0xDFFF, throw a \texttt{URIError} exception.
      ii. If the code unit value of \texttt{C} is less than 0xD800 or greater than 0xDBFF, then
         1. Let \texttt{V} be the code unit value of \texttt{C}.
         2. If \texttt{kChar} is less than 0xDC00 or greater than 0xDFFF, throw a \texttt{URIError} exception.
         3. Let \texttt{V} be (((the code unit value of \texttt{C}) – 0xD800) \times 0x400 + (\texttt{kChar} – 0xDC00) + 0x10000).
      iii. Let \texttt{Octets} be the array of octets resulting by applying the UTF-8 transformation to \texttt{V}, and let \texttt{L} be the array size.
      iv. Let \texttt{jOctet} be the value at position \texttt{j} within \texttt{Octets}.
         1. Let \texttt{j} be 0.
         2. Let \texttt{S} be a String containing three code units \"\%XY\" where \texttt{XY} are two uppercase hexadecimal digits encoding the value of \texttt{jOctet}.
         3. Let \texttt{R} be a new String value computed by concatenating the previous value of \texttt{R} and \texttt{S}.
         4. Increase \texttt{j} by 1.
   e. Increase \texttt{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 \texttt{string} and \texttt{reservedSet}.

1. Let \texttt{strLen} be the number of code units in \texttt{string}.
2. Let \texttt{R} be the empty String.
3. Let \texttt{k} be 0.
4. Repeat
   a. If \texttt{k} equals \texttt{strLen}, return \texttt{R}.
   b. Let \texttt{C} be the code unit at position \texttt{k} within \texttt{string}.
   c. If \texttt{C} is not \texttt{\%}, then
i. Let S be the String containing only the code unit C.

   d. Else C is '%'.
      i. Let start be k.
     
    ii. If k + 2 is greater than or equal to strLen, throw a URIError exception.
    
   iii. If the code units at position (k+1) and (k+2) within string do not represent hexadecimal digits, throw a URIError exception.

   iv. Let B be the 8-bit value represented by the two hexadecimal digits at position (k + 1) and (k + 2).

   v. Increment k by 2.

    vi. If the most significant bit in B is 0, then
        1. Let C be the code unit with code unit value B.
           a. If C is not in reservedSet, then
              i. Let S be the String containing only the code unit C.
           b. Else C is in reservedSet, then
              i. Let S be the substring of string from position start to position k included.

   vii. Else the most significant bit in B is 1,
    
      1. Let n be the smallest nonnegative integer such that (B << n) & 0x80 is equal to 0.
      2. If n equals 1 or n is greater than 4, throw a URIError exception.
      3. Let Octets be an array of 8-bit integers of size n.
      4. Put B into Octets at position 0.
      5. If k + (3 * (n – 1)) is greater than or equal to strLen, throw a URIError exception.
      7. Repeat, while j < n
          a. Increment k by 1.
           b. If the code unit at position k within string is not '%', throw a URIError exception.
           c. If the code units at position (k+1) and (k+2) within string do not represent hexadecimal digits, throw a URIError exception.
          d. Let B be the 8-bit value represented by the two hexadecimal digits at position (k + 1) and (k + 2).
           e. If the two most significant bits in B are not 10, throw a URIError exception.
           f. Increment k by 2.
           g. Put B into Octets at position j.
           h. Increment j by 1.

   8. Let V be the value obtained by applying the UTF-8 transformation to Octets, that is, from an array of octets into a 21-bit value. If Octets does not contain a valid UTF-8 encoding of a Unicode code point throw a URIError exception.

   9. If V < 0x10000, then
      a. Let C be the code unit V.
      b. If C is not in reservedSet, then
         i. Let S be the String containing only the code unit C.
      c. Else C is in reservedSet, then
         i. Let S be the substring of string from position start to position k included.

   10. Else V ≥ 0x10000,
      a. Let L be (((V – 0x10000) & 0x3FF) + 0xDC00).
      b. Let H be ((((V – 0x10000) >> 10) & 0x3FF) + 0xD800).
      c. Let S be the String containing the two code units 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 39.

### Table 39 — 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>0000yy yyzzzzzz</td>
<td>110yyyyy 1zzzzzz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0800 - 0x0FFF</td>
<td>11011yyy yyyyzzzzzz</td>
<td>1110xxxx 10yyyyy 10zzzzzz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{Encoded by}$</td>
<td>110111yy yyzzzzzz</td>
<td>$\text{Encoded by}$</td>
<td>11110uuu 10uuwww 10xyyyyy 10zzzzzz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xD800 - 0xDBFF</td>
<td>causes URIError</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xDC00 - 0xDFFF</td>
<td>causes URIError</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xE000 - 0xFFFF</td>
<td>xxxxyyyy yyyyzzzzzz</td>
<td>1110xxxx 10yyyyy 10zzzzzz</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Where

\[ uuu + 1 \]

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 code unit 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 UTF-16 encoding of the code points 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 code unit valid in uriReserved plus “#”.
4. Return the result of calling Decode(uriString, reservedURISet)

**NOTE**: The code point ‘#’ is not decoded from escape sequences even though it is not a reserved URI code point.
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 UTF-16 encoding of the code points 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 `Decode(componentString, reservedURIComponentSet)`.

18.2.6.4 encodeURI (uri)

The `encodeURI` function computes a new version of an UTF-16 encoded URI in which each instance of certain code points is replaced by one, two, three, or four escape sequences representing the UTF-8 encoding of the code points.

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 code unit valid in `uriReserved` and `uriUnescaped` plus "#".  
4. Return `Encode(uriString, unescapedURISet)`.

**NOTE** The code point "#" is not encoded to an escape sequence even though it is not a reserved or unescaped URI code point.

18.2.6.5 encodeURIComponent (uriComponent)

The `encodeURIComponent` function computes a new version of an UTF-16 encoded URI in which each instance of certain code points is replaced by one, two, three, or four escape sequences representing the UTF-8 encoding of the code point.

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 code unit valid in `uriUnescaped`.  
4. Return `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  Promise ( . . . )
See 25.4.3.

18.3.18  RangeError ( . . . )
See 19.5.5.2.

18.3.19  ReferenceError ( . . . )
See 19.5.5.3.

18.3.20  RegExp ( . . . )
See 21.2.4.

18.3.21  Set ( . . . )
See 23.2.1.

18.3.22  String ( . . . )
See 21.1.1.

18.3.23  Symbol ( . . . )
See 19.4.1.

18.3.24  SyntaxError ( . . . )
See 19.5.5.4.

18.3.25  TypeError ( . . . )
See 19.5.5.5.
18.3.26 Uint8Array ( . . . )
See 22.2.4.

18.3.27 Uint8ClampedArray ( . . . )
See 22.2.4.

18.3.28 Uint16Array ( . . . )
See 22.2.4.

18.3.29 Uint32Array ( . . . )
See 22.2.4.

18.3.30 URIError ( . . . )
See 19.5.5.6.

18.3.31 WeakMap ( . . . )
See 23.3.1.

18.3.32 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 26.4.1.

18.4.4 Reflect
See 26.1.

18.4.5 System
See 26.3.
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 ObjectCreate(%ObjectPrototype%).
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, ...sources)

The assign function is used to copy the values of all of the enumerable own properties from one or more source objects 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. If only one argument was passed, then return to.
4. Let sources be the List of argument values starting with the second argument.
5. For each element nextSource of sources, in ascending index order,
   a. If nextSource is undefined or null, then let keys be an empty List.
   b. Else,
      i. Let from be ToObject(nextSource).
      ii. ReturnIfAbrupt(from).
      iii. Let keys be the result of calling the [[OwnPropertyKeys]] internal method of nextSource.
      iv. ReturnIfAbrupt(keysArray).
   c. Let pendingException be undefined.
   d. Repeat for each element nextKey of keys in List order,
      i. Let desc be the result of calling the [[GetOwnProperty]] internal method of from with argument nextKey.
      ii. If desc is an abrupt completion, then
          1. If pendingException is undefined, then set pendingException to desc.
          2. 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.
      e. If pendingException is not undefined, then return pendingException.
   6. Return to.

The length property of the assign method is 2.

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 ObjectCreate(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.defineProperty ( 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.

19.1.2.3.1 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 keys be the result of calling the [[OwnPropertyKeys]] internal method of props.
4. ReturnIfAbrupt(keys).
5. Let descriptors be an empty List.
6. Repeat for each element nextKey of keys in List order,
   a. Let propDesc be the result of calling the [[GetOwnProperty]] internal method of props with argument nextKey.
   b. ReturnIfAbrupt(propDesc).
   c. If propDesc is not undefined and propDesc.[[Enumerable]] is true, then
      i. Let descObj be the result of Get( props, nextKey).
      ii. ReturnIfAbrupt(descObj).
      iii. Let desc be the result of calling ToPropertyDescriptor with descObj as the argument.
      iv. ReturnIfAbrupt(desc).
      v. Append the pair (a two element List) consisting of nextKey and desc to the end of descriptors.
7. Let pendingException be undefined.
8. For each pair from descriptors in list order,
   a. Let P be the first element of pair.
   b. Let desc be the second element of pair.
   c. Let status be the result of DefinePropertyOrThrow(O, P, desc).
   d. If status is an abrupt completion then
      i. If pendingException is undefined, then set pendingException to status.
9. ReturnIfAbrupt(pendingException).
10. Return O.

NOTE An exception in defining an individual property in step 13 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 \( \text{key} \) be ToPropertyKey(\( P \)).
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(\( \text{desc} \)).

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 \( \text{obj} \) be ToObject(\( O \)).
2. ReturnIfAbrupt(\( \text{obj} \)).
3. Let \( \text{keys} \) be the result of calling the [[OwnPropertyKeys]] internal method of \( \text{obj} \).
4. ReturnIfAbrupt(\( \text{keys} \)).
5. Let \( \text{nameList} \) be a new empty List.
6. Repeat for each element \( \text{nextKey} \) of \( \text{keys} \) in List order,
   a. If Type(\( \text{nextKey} \)) is \( Type \), then
      i. Append \( \text{nextKey} \) as the last element of \( \text{nameList} \).
7. Return CreateArrayFromList(\( \text{nameList} \)).

19.1.2.9 Object.getPrototypeOf ( \( O \) )
When the getPrototypeOf function is called with argument \( O \), the following steps are taken:
1. Let \( \text{obj} \) be ToObject(\( O \)).
2. ReturnIfAbrupt(\( \text{obj} \)).
3. Return the result of calling the [[GetPrototypeOf]] internal method of \( \text{obj} \).

19.1.2.10 Object.is ( \( \text{value1}, \text{value2} \) )
When the is function is called with arguments \( \text{value1} \) and \( \text{value2} \) the following steps are taken:
1. Return SameValue(\( \text{value1} \), \( \text{value2} \)).

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 nameList be EnumerableOwnNames(obj).
  4. 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 11.

19.1.2.15 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.

19.1.2.16 Object.prototype

The initial value of Object.prototype is the standard built-in Object prototype object (19.1.3).

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

19.1.2.17 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.
19.1.2.18 Object.setPrototypeOf ( O, proto )

When the setPrototypeOf function is called with arguments O and proto, the following steps are taken:

1. Let O be RequireObjectCoercible(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.
4. ReturnIfAbrupt(O).
5. 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 `ToPropertyKey(V)`.
2. ReturnIfAbrupt(`P`).
3. Let `O` be the result of calling `ToObject` passing the `this` value as the argument.
4. ReturnIfAbrupt(`O`).
5. Let `desc` be the result of calling the `[[GetOwnProperty]]` internal method of `O` passing `P` as the argument.
6. ReturnIfAbrupt(`desc`).
7. 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 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`.

19.1.3.5 Object.prototype.toLocaleString ([reserved][,reserved])

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")`.

The optional parameters to this function are not used but may be passed but are intended to correspond to the parameter pattern used by ECMA-402 `toLocaleString` functions. Implementations that do not include ECMA-402 support must not use those parameter positions for other purposes.

**NOTE 1** This function provides a generic `toLocaleString` implementation for objects that have no locale-specific `toString` behaviour. `Array`, `Number`, `Date`, and `Typed Arrays` provide their own locale-sensitive `toLocaleString` methods.

**NOTE 2** ECMA-402 intentionally does not provide an alternative to this default implementation.

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` has a `[[ParameterMap]]` internal slot, then let `builtinTag` be "Arguments".
7. Else, if `O` has a `[[Call]]` internal method, then let `builtinTag` be "Function".
8. Else, if `O` has an `[[ErrorData]]` internal slot, then let `builtinTag` be "Error".
9. Else, if `O` has a `[[BooleanData]]` internal slot, then let `builtinTag` be "Boolean".
10. Else, if `O` has a `[[NumberData]]` internal slot, then let `builtinTag` be "Number".
11. Else, if `O` has a `[[DateValue]]` internal slot, then let `builtinTag` be "Date".
12. Else, if `O` has a `[[RegExpMatcher]]` internal slot, then let `builtinTag` be "RegExp".
13. Else, let `builtinTag` be "Object".
14. Let `tag` be the result of `Get(O, @@toStringTag)`.
15. ReturnIfAbrupt(`tag`).
16. If `tag` is `undefined`, then let `tag` be `builtinTag`.
17. Else,
   a. If `Type(tag)` is not `String`, let `tag` be "???".
   b. If `tag` is any of "Arguments", "Array", "Boolean", "Date", "Error",
      "Function", "Number", "RegExp", or "String" and `SameValue(tag, builtinTag)` is `false`, then let `tag` be the string value "~" concatenated with the current value of `tag`.
18. Return the String value that is the result of concatenating the three Strings "[object ", `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 compatibility for legacy code that uses `toString` as a reliable test for those specific kinds of built-in objects. It does not provide a reliable type testing mechanism for other kinds of built-in or program defined objects. In addition, programs can use `@@toStringTag` in ways that will invalidate the reliability of such legacy type tests.

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`.

19.1.4 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 initializes 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 `[[ECMAScriptCode]]` internal slot whose value is `undefined`, it initializes the `this` value using the argument values. This permits `Function` to be used both as factory method and to perform constructor instance initialization.

`Function` may be subclassed and subclass constructors may perform a `super` invocation of the `Function` constructor to initialize 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:

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 6.1.4, using `FormalParameters` as the goal symbol. Throw a `SyntaxError` exception if the parse fails.
9. Let `body` be the result of parsing `bodyText`, interpreted as UTF-16 encoded Unicode text as described in 6.1.4, using `FunctionBody` as the goal symbol. Throw a `SyntaxError` exception if the parse fails or if any static semantics errors are detected.
10. If any element of the `BoundNames` of `parameters` also occurs in the `LexicallyDeclaredNames` of `body`, then throw a `SyntaxError` exception.
11. If `bodyText` is strict mode code (see 10.2.1) then let `strict` be `true`, else let `strict` be `false`.
12. Let `scope` be the Global Environment.
13. Let `F` be the this value.
14. If Type(`F`) is not Object or if `F` does not have a `[[ECMAScriptCode]]` internal slot or if the value of `[[ECMAScriptCode]]` is not `undefined`, then
   a. Let `C` be the active function object.
   b. Let `proto` be the result of `GetPrototypeFromConstructor(C, "\$FunctionPrototype\$")`.
   c. ReturnIfAbrupt(`proto`).
   d. Let `F` be `FunctionAllocate(C, strict)`.
   e. ReturnIfAbrupt(`F`).
15. If the value of `F`'s `[[FunctionKind]]` internal slot is not "normal", then throw a `TypeError` exception.
16. Let `isExtensible` be `IsExtensible(F)`.
17. ReturnIfAbrupt(`isExtensible`).
18. If `isExtensible` is `false`, then throw a `TypeError` exception.
19. Let `status` be `FunctionInitialize(F, Normal, strict, parameters, body, scope)`.
20. ReturnIfAbrupt(`status`).
21. If `ReferencesSuper` of `body` is `true` or `ReferencesSuper` of `parameters` is `true`, then
   a. Perform `MakeMethod`(F, `undefined`, `undefined`).
22. Let `status` be the result of `MakeConstructor` with argument `F`.
23. ReturnIfAbrupt(`status`).
24. Let `hasName` be `HasOwnProperty(F, "name")`.
25. ReturnIfAbrupt(`hasName`).

© Ecma International 2014
26. If `hasName` is `false`, then
   a. Let `status` be `SetFunctionName(F, "anonymous")`.
   b. ReturnIfAbrupt(`status`).
27. Return `F`.

The `length` property of the `Function` function is 1 (see 19.2.2.1).

NOTE 1 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 2 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:

```javascript
new Function("a", "b", "c", "return a+b+c")
new Function("a, b, c", "return a+b+c")
new Function("a,b", "c", "return a+b+c")
```

19.2.1.2 `new Function(...argumentsList)`

When `Function` is called as part of a `new` expression, it initializes the newly created object.

1. Let `F` be the `Function` 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 `Construct(F, argumentsList)`.

If `Function` is implemented as an ECMAScript function object, its `[[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 `[[Prototype]]` internal slot of the Function constructor is `%FunctionPrototype%`, the intrinsic `Function` prototype object (19.2.3).

The value of the `[[Extensible]]` internal slot of the Function constructor is `true`.

The Function constructor has the following properties:

19.2.2.1 `Function.length`

This is a data property with a value of 1. This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

19.2.2.2 `Function.prototype`

The value of `Function.prototype` is `%FunctionPrototype%`, the intrinsic `Function` prototype object (19.2.3).

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.`
19.2.3 Function[@@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 GetPrototypeFromConstructor(\( F \), "%FunctionPrototype%").
3. ReturnIfAbrupt(\( \text{proto} \)).
4. Return FunctionAllocate(\( \text{proto} \), false).

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 Function @@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 constructor function. The Function constructor may reset the functions [[Strict]] internal slot to 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 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 [[Prototype]] internal slot of the Function 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.

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 \( \text{func} \) with arguments \( \text{thisArg} \) and \( \text{argArray} \), the following steps are taken:

1. If IsCallable(\( \text{func} \)) is false, then throw a TypeError exception.
2. If \( \text{argArray} \) is null or undefined, then
   a. Return the result of calling the [[Call]] internal method of \( \text{func} \), providing \( \text{thisArg} \) as thisArgument and an empty List of arguments as argumentsList.
3. Let \( \text{argList} \) be the result of CreateListFromArrayLike(\( \text{argArray} \)).
4. ReturnIfAbrupt(\( \text{argList} \)).
5. Perform PrepareForTailCall().
6. Return the result of calling the [[Call]] internal method of \( \text{func} \), providing \( \text{thisArg} \) as thisArgument and argList as argumentsList.

The length property of the apply method is 2.
NOTE 1  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. Even though the thisArg is passed without modification, non-strict mode functions still perform these transformations upon entry to the function.

NOTE 2  If func is an arrow function or a bound found then the thisArg will be ignored by the function [[Call]] in step 6.

19.2.3.2  Function.prototype.bind (thisArg, ...args)

When the bind method is called with argument thisArg and zero or more args, it performs the following steps:

1. Let Target be the this value.
2. If IsCallable(Target) is false, throw a TypeError exception.
3. Let args be a new (possibly empty) List consisting of all of the argument values provided after thisArg in order.
4. Let F be BoundFunctionCreate(Target, thisArg, args).
5. Let targetHasLength be HasOwnProperty(Target, "length").
6. ReturnIfAbrupt(targetHasLength).
7. If targetHasLength is true, then
   a. Let targetLen be Get(Target, "length").
   b. ReturnIfAbrupt(targetLen).
   c. If Type(targetLen) is not Number, then let L be 0.
   d. Else
      i. Let L be the larger of 0 and the result of targetLen minus the number of elements of A.
8. Else let L be 0.
9. Let status be DefinePropertyOrThrow(F, "length", PropertyDescriptor {[[Value]]: L, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true}).
10. ReturnIfAbrupt(status).
11. Let targetName be Get(Target, "name").
12. ReturnIfAbrupt(targetName).
13. If Type(targetName) is not String, then let targetName be the empty string.
14. Let status be SetFunctionName(F, targetName, "bound").
15. ReturnIfAbrupt(status).
16. Return F.

The length property of the bind method is 1.

NOTE 1  Function objects created using Function.prototype.bind are exotic objects. They also do not have a prototype property.

NOTE 2  If Target is an arrow function or a bound found then the thisArg passed to this method will not be used by subsequent calls to F.

19.2.3.3  Function.prototype.call (thisArg, ...args)

When the call method is called on an object func with argument thisArg and zero or more args, 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 the second argument append each argument as the last element of argList.
4. Perform PrepareForTailCall() .
5. Return the result of calling the [[Call]] internal method of \textit{func}, providing \textit{thisArg} as \textit{thisArgument} and argList as argumentsList.

The length property of the \textit{call} method is 1.

\textbf{NOTE 1} The thisArg value is passed without modification as the this value. This is a change from Edition 3, where an \texttt{undefined} or \texttt{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. Even though the thisArg is passed without modification, non-strict mode functions still perform these transformations upon entry to the function.

\textbf{NOTE 2} If \textit{func} is an arrow function or a bound found then the thisArg will be ignored by the function [[Call]] in step 5.

\textbf{19.2.3.4 Function.prototype.constructor} 

The initial value of \texttt{Function.prototype.constructor} is the intrinsic object %Function%.

\textbf{19.2.3.5 Function.prototype.toMethod (newHome [ , methodName ])}

When the \textit{toMethod} method is called on an object \textit{func} with argument \textit{newHome} and optional argument \textit{methodName} the following steps are taken:

1. If Type(\textit{newHome}) is not Object, then throw a TypeError exception.
2. If \textit{func} is an ECMAScript function object or an exotic Built-in function object, then
   a. If \textit{methodName} is not \texttt{undefined}, then
      i. Let \textit{methodName} be ToPropertyKey(\textit{methodName}).
      ii. ReturnIfAbrupt(\textit{methodName}).
   b. Return CloneMethod(\textit{func}, \textit{newHome}, \textit{methodName}).
3. If \textit{func} is a Bound Function exotic object, then return BoundFunctionClone(\textit{func}).
4. If \textit{func} is any other exotic function object that supports the equivalent of the CloneMethod abstract operation, then return an appropriately cloned object.
5. Throw a TypeError exception.

The length property of the \textit{toMethod} method is 1.

\textbf{19.2.3.6 Function.prototype.toString ( )}

When the \textit{toString} method is called on an object \textit{func} the following steps are taken:

1. If \textit{func} is a Bound Function exotic object, then
   a. Return an implementation-dependent String source code representation of \textit{func}. The representation must conform to the rules below. It is implementation dependent whether the representation includes bound function information or information about the target function.
2. If Type(\textit{func}) is Object and is either a Built-in function object or has a [[ECMAScriptCode]] internal slot, then
   a. Return an implementation-dependent String source code representation of \textit{func}. The representation must conform to the rules below.
3. Return a TypeError exception.

\textbf{toString Representation Requirements:}

\textbf{Commented [AWB2287]}: Do we also want to copy \textit{func}'s length property? If so, it probably happens in clone method.
• The string representation must have the syntax of a `FunctionDeclaration` `FunctionExpression`, `GeneratorDeclaration`, `GeneratorExpression`, `ClassDeclaration`, `ClassExpression`, `ArrowFunction`, `MethodDefinition`, or `GeneratorMethod` depending upon the actual characteristics of the object.

• 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 not in the form of a `MethodDefinition` or `GeneratorMethod` 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. In that case 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 implementation cannot produce a source code string that meets these criteria then it must return a string for which `eval` will throw a `SyntaxError` exception.

19.2.3.7 `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 "@@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 over-ride it.

19.2.3.8 `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 "@@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

\[
\text{v instanceof F}
\]

evaluates as `F[@@hasInstance](v)`

A constructor function can control which objects are recognized 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 ECMAScript function object and has the internal slots listed in Table 26.
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 ECMAScript 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 an `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 ECMAScript 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%`.

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 initialize 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 }.

NOTE: Function objects created using `Function.prototype.bind`, or by evaluating a `MethodDefinition` (that is not a `GeneratorMethod` or an `ArrowFunction` grammar production do not have a `prototype` property.

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 uninitialized `[[BooleanData]]` internal slot, it initializes the `this` value using the argument value. This permits `Boolean` to be used both to perform type conversion and to perform constructor instance initialization.

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 initialize the `[[BooleanData]]` state of subclass instances.

### 19.3.1  `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 initializes 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 `Construct(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 `obj` be the result of calling `OrdinaryCreateFromConstructor(F, "%BooleanPrototype%", ( `[[BooleanData]]`) )
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  [[BooleanData]] is initially assigned the value `undefined` as a flag to indicate that the instance has not yet been initialized 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.

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 [[BooleanData]] internal slot.

The value of the [[Prototype]] internal slot of the Boolean prototype object is the standard built-in Object prototype object (19.1.3).

The abstract operation `thisBooleanValue(value)` performs the following steps:

1. If `Type(value)` is Boolean, return `value`.
2. If `Type(value)` is Object and `value` has a [[BooleanData]] internal slot, then
   a. Let `b` be the value of `value`’s [[BooleanData]] internal slot.
   b. If `b` is not `undefined`, then return `b`.
3. Throw a `TypeError` exception.

19.3.3.1 Boolean.prototype.constructor

The initial value of `Boolean.prototype.constructor` is the built-in `Boolean` constructor.

19.3.3.2 Boolean.prototype.toString()

The following steps are taken:

1. Let `b` be `thisBooleanValue(this value)`.
2. ReturnIfAbrupt(b).
3. If `b` is `true`, then return "true"; else return "false".

19.3.3.3 Boolean.prototype.valueOf()

The following steps are taken:

1. Return `thisBooleanValue(this value)`.

19.3.4 Properties of Boolean Instances

Boolean instances are ordinary objects that inherit properties from the Boolean prototype object. Boolean instances have a [[BooleanData]] internal slot. The [[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 initialize the state of subclass instances.

### 19.4.1.1 Symbol ([ description ])

When `Symbol` is called with optional 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. Return `IfAbrupt(descString)`.
4. Return a new unique `Symbol` value whose `[[Description]]` value is `descString`.

### 19.4.1.2 new Symbol ( ...argumentsList )

When `Symbol` is called as part of a new expression, it initializes 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 `Construct(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 1), 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.for ( key )

When `Symbol.for` is called with argument `key` it performs the following steps:

1. Let `stringKey` be `ToString(key)`.
2. Return `IfAbrupt(stringKey)`.
3. For each element `e` of the `GlobalSymbolRegistry` List,
   a. If `SameValue(e.[[key]], stringKey)` is `true`, then return `e.[[symbol]]`.
4. Assert: `GlobalSymbolRegistry` does not currently contain an entry for `stringKey`.
5. Let `newSymbol` be a new unique `Symbol` value whose `[[Description]]` is `stringKey`.
6. Append the record `{ [[key]]: stringKey, [[symbol]]: newSymbol }` to the `GlobalSymbolRegistry` List.
7. Return `newSymbol`. 

© Ecma International 2014
The GlobalSymbolRegistry is a List that is globally available. It is shared by all Code Realms. Prior to the evaluation of any ECMAScript code it is initialized as an empty List. Elements of the GlobalSymbolRegistry are Records with the structure defined in Table 40.

### Table 40 — GlobalSymbolRegistry Record Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[key]]</td>
<td>A String</td>
<td>A string key used to globally identify a Symbol.</td>
</tr>
<tr>
<td>[[symbol]]</td>
<td>A Symbol</td>
<td>A symbol that can be retrieved from any Realm.</td>
</tr>
</tbody>
</table>

19.4.2.3 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 }.

19.4.2.4 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 }.

19.4.2.5 Symbol.isRegExp

The initial value of `Symbol.isRegExp` is the well known symbol `@@isRegExp` (Table 1).

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

19.4.2.6 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 }.

19.4.2.7 Symbol.keyFor (sym)

When `Symbol.keyFor` is called with argument `sym` it performs the following steps:

1. If `Type(sym)` is not `Symbol`, then throw a `TypeError` exception.
2. For each element `e` of the GlobalSymbolRegistry List (see 19.4.2.2),
   a. If `SameValue(e. [[symbol]], sym)` is `true`, then return `e. [[key]].
3. Assert: GlobalSymbolRegistry does not currently contain an entry for `sym`.
4. Return `undefined`.

19.4.2.8 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 }.

19.4.2.9 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 }.

19.4.2.10 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 }.

19.4.2.11 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 }.

19.4.2.12 Symbol[ @@create ] ()
The @@create method of a Symbol object 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 Type(s) is Symbol, then let sym be s.
3. Else,
   a. If s does not have a [[SymbolData]] internal slot, then throw a TypeError exception.
   b. Let sym be the value of s’s [[SymbolData]] internal slot.
4. Return SymbolDescriptiveString(O).
19.4.3.3 SymbolDescriptiveString ( sym ) Abstract Operation

The abstract operation SymbolDescriptiveString is called with argument sym, the following steps are taken:

1. Assert: Type(sym) is Symbol.
2. Let desc be the value of sym’s [[Description]] attribute.
3. If desc is undefined, then let desc be the empty string.
4. Assert: Type(desc) is String.
5. Let result be the result of concatenating the strings “Symbol(", desc, ")”.
6. Return result.

19.4.3.4 Symbol.prototype.valueOf ()

The following steps are taken:

1. Let s be the this value.
2. If Type(s) is Symbol, then return s.
3. If s does not have a [[SymbolData]] internal slot, then throw a TypeError exception.
4. Return the value of s’s [[SymbolData]] internal slot.

19.4.3.5 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".

When the @@toPrimitive method is called with argument hint, the following steps are taken:

1. Let s be the this value.
2. If Type(s) is Symbol, then return s.
3. If Type(s) is not Object, then throw a TypeError exception.
4. If s does not have a [[SymbolData]] internal slot, then throw a TypeError exception.
5. Return the value of s’s [[SymbolData]] internal slot.

The value of the name property of this function is "Symbol.toPrimitive".

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

19.4.3.6 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 initializes 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 uninitialized [ErrorData] internal slot, it initializes 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 initialization.

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 initialize 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 the active 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 Construct(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 initialized 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, the code unit U+003A (COLON), the code unit U+0020 (SPACE), 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 initialized 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 the actual type of an operand is different than the expected type. See 8.6.2, 8.9.2, 8.10.5, 8.12.5, 8.12.7, 8.12.8, 8.12.9, 9.9, 9.10, 10.2.1, 10.2.1.1.3, 10.6, 11.2.2, 11.2.3, 11.4.1, 11.8.6, 11.8.7, 11.9.1.3, 13.2, 13.2.3, 15, 15.2.3.2, 15.2.3.3, 15.2.3.4, 15.2.3.5, 15.2.3.6, 15.2.3.7, 15.2.3.8, 15.2.3.9, 15.2.3.10, 15.2.3.11, 15.2.3.12, 15.2.3.13, 15.2.3.14, 15.2.3.15, 15.2.3.32, 15.3.3.3, 15.3.3.4, 15.3.3.5, 15.3.4, 15.3.4.3, 15.3.4.4, 15.3.4.5, 15.4.3.1, 15.4.3.11, 15.4.3.16, 15.4.3.17, 15.4.3.18, 15.4.3.19, 15.4.3.20, 15.4.3.21, 15.4.3.22, 15.4.5.1, 15.5.4.2, 15.5.4.3, 15.6.4.2, 15.6.4.3, 15.7.4, 15.7.4.2, 15.7.4.4, 15.9.5, 15.9.5.44, 15.10.4.1, 15.10.6, 15.11.4.4 and 15.12.3.

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.

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 initializes 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 uninitialized [[ErrorData]] internal slot, it initializes the `this` value using the argument value. This permits a `NativeError` to be used both as a factory method and to perform constructor instance initialization.

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 initialize 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 the active 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`, [the]
   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%", "%SyntaxErrorPrototype%", or "%TypeErrorPrototype%" 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 Construct(F, argumentsList).

If a NativeError constructor is implemented as an ECMAscript function object, its [[Construct]] internal method will perform the above steps.

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 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 initialized 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 initialized 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 uninitialized `[[NumberData]]` internal slot, it initializes the `this` value using the argument value. This permits `Number` to be used both to perform type conversion and to perform constructor instance initialization.
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 initialize 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 `Construct(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 `2.2204460492503130808472633361816 x 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`, `+∞`, or `−∞`, 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, +∞, or −∞, 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 NumberisNaN ( 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

NOTE The value of Number.MAX_SAFE_INTEGER is the largest integer n such that n and n + 1 are both exactly representable as a Number value.

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 × 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

NOTE The value of Number.MIN_SAFE_INTEGER is the smallest integer \(n\) such that \(n\) and \(n - 1\) are both exactly representable as a Number value.

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 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 initialized 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 initialized 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.constructor

The initial value of `Number.prototype.constructor` is the built-in `Number` constructor.

20.1.3.2 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(s).
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. Let s be the empty String.
8. If x < 0, then
   a. Let s be "-".
   b. Let x = –x.
9. If x = +∞, then
   a. Return the concatenation of the Strings s and "Infinity".
10. If f < 0 or f > 20, throw a RangeError exception.
11. If x = 0, then
    a. Let m be the String consisting of f+1 occurrences of the code unit 0x0030.
    b. Let e = 0.
12. Else x ≠ 0,
    a. If fractionDigits is not undefined, then
       i. Let e and n be integers such that \(10^f \leq n < 10^{f+1}\) and for which the exact mathematical value of 
          \(n \times 10^{-e} - x\) is as close to zero as possible. If there are two such sets of e and n, pick the 
          e and n for which \(n \times 10^{-e} - x\) is larger.
    b. Else fractionDigits is undefined, 
       i. Let e, n, and f be integers such that \(f \geq 0, 10^f \leq n < 10^{f+1}\), the number value for \(n \times 10^{-e}\) 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).
    13. If f ≠ 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.
    14. If e = 0, then
       a. Let c = "+\".
       b. Let d = "0\".
    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 toExponential for values of fractionDigits less than 0 or greater than 20. In this case toExponential would not necessarily throw RangeError for such values.

© Ecma International 2014
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 ≥ 0, 10^f ≤ n < 10^{f+1}, the number value for n \times 10^{-f} is x, and f is as small as possible. If there are multiple possibilities for n, choose the value of n for which n \times 10^{-f} is closest in value to x. If there are two such possible values of n, choose the one that is even.

20.1.3.3 Number.prototype.toFixed ( fractionDigits )

Note toFixed returns a String containing this Number value represented in decimal fixed-point notation with fractionDigits digits after the decimal point. If fractionDigits is undefined, 0 is assumed.

The following steps are performed:

1. Let x be thisNumberValue(this value).
2. ReturnIfAbrupt(x).
3. Let f be ToInteger(fractionDigits). (If fractionDigits is undefined, this step produces the value 0).
4. ReturnIfAbrupt(f).
5. If f < 0 or f > 20, throw a RangeError exception.
6. If x is NaN, return the String "NaN".
7. Let s be the empty String.
8. If x < 0, then
   a. Let s be "-".
   b. Let x = –x.
9. If x ≥ 10^{21}, then
   a. Let m = 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 = 0, then
       i. Let k be the number of elements in m.
       ii. If k ≤ f, then
           1. Let z be the string consisting of f+1–k occurrences of the code unit 0x0030.
           2. Let m be the concatenation of Strings z and m.
           3. Let k = f + 1.
       iii. Let a be the first k–f elements of m, and let b be the remaining f elements of m.
       iv. Let m be the concatenation of the three Strings a, ".", and b.
    11. Return the concatenation of the Strings s and m.

The length property of the toFixed method is 1.

If the toFixed method is called with more than one argument, then the behaviour is undefined (see clause 17).

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

NOTE The output of toFixed may be more precise than toString for some values because toString only prints enough significant digits to distinguish the number from adjacent number values. For example, (100000000000000128).toString() returns "100000000000000100", while (100000000000000128).toFixed(0) returns "100000000000000128".
20.1.3.4 Number.prototype.toLocaleString( [ reserved1 [, reserved2 ] ])

An ECMAScript implementation that includes the ECMA-402 Internationalization 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.

The length property of the toLocaleString method is 0.

20.1.3.5 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 (7.1.12) instead. Specifically, perform the following steps:

1. Let x be thisNumberValue(this value).
2. ReturnIfAbrupt(x).
3. If precision is undefined, return ToString(x).
4. Let p be ToInteger(precision).
5. ReturnIfAbrupt(p).
6. If x is NaN, return the String "NaN".
7. Let s be the empty String.
8. If x < 0, then
   a. Let s be code unit U+002D (HYPHEN-MINUS).
   b. Let x = -x.
9. If x = +∞, then
   a. Return the String that is the concatenation of 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 U+0030 (DIGIT ZERO).
   b. Let e = 0.
12. Else n = 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} - 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 × 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 a, code unit U+002E (FULL STOP), and b.
      iv. If e > 0, then
          1. Let c be code unit U+002B (PLUS SIGN).
v. Else $e < 0$.
   1. Let $c$ be code unit U+002D (HYPHEN-MINUS).
   2. Let $e = -e$.
   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 $s$, $m$, code unit U+0065 (LATIN SMALL LETTER E), $c$, and $d$.

13. If $e = p - 1$, then return the concatenation of the Strings $s$ and $m$.
14. If $e \geq 0$, then
   a. Let $m$ be the concatenation of the first $e+1$ elements of $m$, the code unit U+002E (FULL STOP), and the remaining $p-(e+1)$ elements of $m$.
15. Else $e < 0$,
   a. Let $m$ be the String formed by the concatenation of code unit U+0030 (DIGIT ZERO), code unit U+002E (FULL STOP), $-(e+1)$ occurrences of code unit U+0030 (DIGIT ZERO), and the String $m$.
16. Return the String that is the concatenation of $s$ and $m$.

The length property of the toPrecision method is 1.

If the toPrecision method is called with more than one argument, then the behaviour is undefined (see clause 17).

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

20.1.3.6 Number.prototype.toString ( [ radix ] )

NOTE: The optional radix should be an integer value in the inclusive range 2 to 36. If radix not present or is undefined the Number 10 is used as the value of radix.

The following steps are performed:
1. Let $x$ be thisNumberValue(this value).
2. ReturnIfAbrupt($x$).
3. If radix is not present, then let radixNumber be 10.
4. Else if radix is undefined, then let radixNumber be 10.
5. Else let radixNumber be ToInteger(radix).
6. ReturnIfAbrupt(radixNumber).
7. If radixNumber < 2 or radixNumber > 36, then throw a RangeError exception.
8. If radixNumber = 10, then return ToString($x$).
9. Return the String representation of this Number value using the radix specified by radixNumber.
   Letters a-z are used for digits with values 10 through 35. The precise algorithm is implementation-dependent; however the algorithm should be a generalization of that specified in 7.1.12.1.

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

20.1.3.7 Number.prototype.valueOf ( )

1. Let $x$ be thisNumberValue(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 [[NumberData]] internal slot. The [[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 [[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). The value returned by each function is a Number.

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 +∞, the result is +∞.

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 -π/2 to +π/2.

- If x is NaN, the result is NaN.
- If x is greater than 1, the result is NaN.
- If x is less than -1, the result is NaN.
- If x is +0, the result is +0.
- If x is -0, the result is -0.

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 +∞, the result is +∞.
• If $x = -\infty$, 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 $+0$.

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 $x$ or $y$ is NaN, the result is NaN.
- If $y > 0$ and $x$ is $+0$, the result is an implementation-dependent approximation to $+\pi/2$.
- If $y > 0$ and $x$ is $-0$, the result is an implementation-dependent approximation to $+\pi/2$.
- If $y$ is $+0$ and $x$ is $+0$, the result is $+0$.
- If $y$ is $+0$ and $x$ is $-0$, the result is $-0$.
- If $y$ is $-0$ and $x$ is $+0$, the result is $-0$.
- If $y$ is $-0$ and $x$ is $-0$, the result is an implementation-dependent approximation to $-\pi$.
- If $y$ is $+0$ and $x$ is $+0$, the result is an implementation-dependent approximation to $-\pi$.
- If $y > 0$ and $x$ is $+0$, the result is an implementation-dependent approximation to $-\pi/2$.
- If $y < 0$ and $x$ is $+0$, the result is an implementation-dependent approximation to $-\pi/2$.
- If $y > 0$ and $x$ is finite and $x$ is $+\infty$, the result is $+0$.
- If $y > 0$ and $y$ is finite and $x$ is $+\infty$, the result is an implementation-dependent approximation to $+\pi$.
- If $y > 0$ and $y$ is finite and $x$ is $+\infty$, the result is $+0$.
- If $y > 0$ and $y$ is finite and $x$ is $+\infty$, the result is an implementation-dependent approximation to $+\pi$.
- If $y > 0$ and $x$ is finite, the result is an implementation-dependent approximation to $+\pi/2$.
- If $y > 0$ and $x$ is finite, the result is an implementation-dependent approximation to $-\pi/2$.
- If $y > 0$ and $x$ is finite, the result is an implementation-dependent approximation to $+\pi/4$. 
• If \( y \) is \(+\infty\) and \( x \) is \(-\infty\), the result is an implementation-dependent approximation to \(+3\pi/4\).
• If \( y \) is \(-\infty\) and \( x \) is \(+\infty\), the result is an implementation-dependent approximation to \(-\pi/4\).

20.2.2.9 Math.cbrt \((x)\)

Returns an implementation-dependent approximation to the cube root of \( x \).

- If \( x \) is NaN, the result is NaN.
- If \( x \) is +0, the result is +0.
- If \( x \) is −0, the result is −0.
- If \( x \) is +\( \infty \), the result is +\( \infty \).
- If \( x \) is −\( \infty \), the result is −\( \infty \).

20.2.2.10 Math.ceil \((x)\)

Returns the smallest (closest to \(-\infty\)) Number value that is not less than \( x \) and is equal to a mathematical integer. If \( x \) is already an integer, the result is \( x \).

- If \( x \) is NaN, the result is NaN.
- If \( x \) is +0, the result is +0.
- If \( x \) is −0, the result is −0.
- If \( x \) is +\( \infty \), the result is +\( \infty \).
- If \( x \) is −\( \infty \), the result is −\( \infty \).

The value of \( \text{Math.ceil}(x) \) is the same as the value of \( -\text{Math.floor}(-x) \).

20.2.2.11 Math.clz32 \((x)\)

When Math.clz32 is called with one argument \( x \), the following steps are taken:
1. Let \( n \) be ToUint32 \((x)\).
2. ReturnIfAbrupt \((n)\).
3. Let \( p \) be the number of leading zero bits in the 32-bit binary representation of \( n \).
4. 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.

20.2.2.12 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 +\( \infty \), the result is NaN.
- If \( x \) is −\( \infty \), the result is NaN.

20.2.2.13 Math.cosh \((x)\)

Returns an implementation-dependent approximation to the hyperbolic cosine of \( x \).
• If \( x \) is NaN, the result is NaN.
• If \( x \) is +0, the result is 1.
• If \( x \) is −0, the result is 1.
• If \( x \) is +∞, the result is +∞.
• If \( x \) is −∞, the result is +∞.

NOTE The value of \( \cosh(x) \) is the same as \( (\exp(x) + \exp(-x))/2 \).

20.2.2.14 Math.exp ( x )

Returns an implementation-dependent approximation to the exponential function of \( x \) (\( e \) raised to the power of \( x \), where \( e \) is the base of the natural logarithms).

• If \( x \) is NaN, the result is NaN.
• If \( x \) is +0, the result is 1.
• If \( x \) is −0, the result is 1.
• If \( x \) is +∞, the result is +∞.
• If \( x \) is −∞, the result is +0.

20.2.2.15 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 +∞, the result is +∞.
• If \( x \) is −∞, the result is +0.

20.2.2.16 Math.floor ( x )

Returns the greatest (closest to +∞) Number value that is not greater than \( x \) and is equal to a mathematical integer. If \( x \) is already an integer, the result is \( x \).

• If \( x \) is NaN, the result is NaN.
• If \( x \) is +0, the result is +0.
• If \( x \) is −0, the result is −0.
• If \( x \) is +∞, the result is +∞.
• If \( x \) is −∞, the result is −∞.
• If \( x \) is greater than 0 but less than 1, the result is +0.

NOTE The value of \( \text{Math.floor}(x) \) is the same as the value of \( \text{-Math.ceil}(-x) \).

20.2.2.17 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, +∞, −∞, then return \( x \).
3. Let \( x_{32} \) be the result of converting \( x \) to a value in IEEE-754-2008 binary32 format using roundTiesToEven.
4. Let \( x64 \) be the result of converting \( x32 \) to a value in IEEE-754-2008 binary64 format.
5. Return the ECMAScript Number value corresponding to \( x64 \).

20.2.2.18 \texttt{Math.hypot (value1, value2, ...values)}

\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 length property of the \texttt{hypot} function is 2.

\textbf{NOTE:} Implementations should take care to avoid the loss of precision from overflows and underflows that are prone to occur in naive implementations when this function is called with more than two arguments.

20.2.2.19 \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 \( product \) be (\( a \times b \)) modulo 2\(^{32} \).
6. If \( product \geq 2^{31} \), return \( product \times 2^{32} \), otherwise return \( product \).

20.2.2.20 \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 +\( \infty \).

20.2.2.21 \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 +\( \infty \).
20.2.2.22  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 $+\infty$.

20.2.2.23  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 $+\infty$.

20.2.2.24  Math.max ( value1, value2, …, values )

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 (7.2.9) except that $+0$ is considered to be larger than $-0$.

The length property of the max method is 2.

20.2.2.25  Math.min ( value1, value2, …, values )

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 (7.2.9) except that $+0$ is considered to be larger than $-0$.

The length property of the min method is 2.

20.2.2.26  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 \(\text{abs}(x) > 1\) and y is +\(\infty\), the result is +\(\infty\).
• If \(\text{abs}(x) > 1\) and y is −\(\infty\), the result is +0.
• If \(\text{abs}(x) < 1\) and y is +\(\infty\), the result is +0.
• If \(\text{abs}(x) < 1\) and y is −\(\infty\), the result is +\(\infty\).
• If x is +\(\infty\) and y>0, the result is +\(\infty\).
• If x is +\(\infty\) and y<0, the result is +0.
• If x is −\(\infty\) and y>0 and y is an odd integer, the result is −\(\infty\).
• If x is −\(\infty\) and y>0 and y is not an odd integer, the result is +0.
• If x is −\(\infty\) and y<0 and y is an odd integer, the result is −\(\infty\).
• If x is −\(\infty\) and y<0 and y is not an odd integer, the result is +\(\infty\).

**20.2.2.27 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.

**20.2.2.28 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 +\(\infty\). 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 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 not always the same as the value of Math.floor(x+0.5). When x is −0 or is less than 0 but greater than or equal to −0.5, Math.round(x) returns −0, but Math.floor(x+0.5) returns
Math.round(x) may also differ from the value of Math.floor(x+0.5) because of internal rounding when computing x+0.5.

20.2.2.29 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.

20.2.2.30 Math.sin(x)

Returns an implementation-dependent approximation to the sine of x. The argument is expressed in radians.

- If x is NaN, the result is NaN.
- If x is +0, the result is +0.
- If x is -0, the result is -0.
- If x is +∞ or -∞, the result is NaN.

NOTE The value of sinh(x) is the same as (exp(x) - exp(-x))/2.

20.2.2.31 Math.sinh(x)

Returns an implementation-dependent approximation to the hyperbolic sine of x.

- If x is NaN, the result is NaN.
- If x is +0, the result is +0.
- If x is -0, the result is -0.
- If x is +∞, the result is +∞.
- If x is -∞, the result is -∞.

20.2.2.32 Math.sqrt(x)

Returns an implementation-dependent approximation to the square root of x.

- If x is NaN, the result is NaN.
- If x is less than 0, the result is NaN.
- If x is +0, the result is +0.
- If x is -0, the result is -0.
- If x is +∞, the result is +∞.

20.2.2.33 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 \( \text{NaN} \).

20.2.2.34 Math.tanh(\( x \))

Returns an implementation-dependent approximation to the hyperbolic tangent of \( x \).

- If \( x \) is \( \text{NaN} \), the result is \( \text{NaN} \).
- 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 \( (\exp(x) - \exp(-x))/(\exp(x) + \exp(-x)) \).

20.2.2.35 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 \( \text{NaN} \), the result is \( \text{NaN} \).
- If \( x \) is \(-0\), the result is \(-0\).
- If \( x \) is \(+0\), the result is \(+0\).
- If \( x \) is \(+\infty\), the result is \(+\infty\).
- If \( x \) is \(-\infty\), the result is \(-\infty\).
- If \( x \) is greater than 0 but less than 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 \( \text{NaN} \), the result will be \( \text{NaN} \).

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 \( \text{NaN} \), 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

\[
\text{DaysInYear}(y) = \begin{cases} 
365 & \text{if } (y \mod 4) \neq 0 \\
366 & \text{if } (y \mod 4) = 0 \text{ and } (y \mod 100) \neq 0 \\
365 & \text{if } (y \mod 100) = 0 \text{ and } (y \mod 400) \neq 0 \\
366 & \text{if } (y \mod 400) = 0 
\end{cases}
\]

All non-leap years have 365 days with the usual number of days per month and leap years have an extra day in February. The day number of the first day of year \( y \) is given by:

\[
\text{DayFromYear}(y) = 365 \times (y - 1970) + \text{floor}(\frac{(y - 1969)}{4}) - \text{floor}(\frac{(y - 1901)}{100}) + \text{floor}(\frac{(y - 1601)}{400})
\]

The time value of the start of a year is:

\[
\text{TimeFromYear}(y) = \text{msPerDay} \times \text{DayFromYear}(y)
\]

A time value determines a year by:

\[
\text{YearFromTime}(t) = \text{the largest integer } y \text{ (closest to positive infinity) such that } \text{TimeFromYear}(y) \leq t
\]

The leap-year function is 1 for a time within a leap year and otherwise is zero:

\[
\text{InLeapYear}(t) = \begin{cases} 
0 & \text{if } \text{DaysInYear(YearFromTime}(t)) = 365 \\
1 & \text{if } \text{DaysInYear(YearFromTime}(t)) = 366
\end{cases}
\]

20.3.1.4 Month Number

Months are identified by an integer in the range 0 to 11, inclusive. The mapping \( \text{MonthFromTime}(t) \) from a time value \( t \) to a month number is defined by:

\[
\text{MonthFromTime}(t) = \begin{cases} 
0 & \text{if } 0 \leq \text{DayWithinYear}(t) < 31 \\
1 & \text{if } 31 \leq \text{DayWithinYear}(t) < 59 + \text{InLeapYear}(t) \\
2 & \text{if } 59 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 90 + \text{InLeapYear}(t) \\
3 & \text{if } 90 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 120 + \text{InLeapYear}(t) \\
4 & \text{if } 120 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 151 + \text{InLeapYear}(t) \\
5 & \text{if } 151 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 181 + \text{InLeapYear}(t) \\
6 & \text{if } 181 + \text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 212 + \text{InLeapYear}(t)
\end{cases}
\]
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.

20.3.1.5 Date Number

A date number is identified by an integer in the range 1 through 31, inclusive. The mapping DateFromTime(t) from a time value t to a date number is defined by:

\[
\text{DateFromTime}(t) = \begin{cases} 
\text{DayWithinYear}(t)+1 & \text{if } \text{MonthFromTime}(t)=0 \\
\text{DayWithinYear}(t)-30 & \text{if } \text{MonthFromTime}(t)=1 \\
\text{DayWithinYear}(t)-59-\text{InLeapYear}(t) & \text{if } \text{MonthFromTime}(t)=2 \\
\text{DayWithinYear}(t)-90-\text{InLeapYear}(t) & \text{if } \text{MonthFromTime}(t)=3 \\
\text{DayWithinYear}(t)-120-\text{InLeapYear}(t) & \text{if } \text{MonthFromTime}(t)=4 \\
\text{DayWithinYear}(t)-150-\text{InLeapYear}(t) & \text{if } \text{MonthFromTime}(t)=5 \\
\text{DayWithinYear}(t)-180-\text{InLeapYear}(t) & \text{if } \text{MonthFromTime}(t)=6 \\
\text{DayWithinYear}(t)-211-\text{InLeapYear}(t) & \text{if } \text{MonthFromTime}(t)=7 \\
\text{DayWithinYear}(t)-242-\text{InLeapYear}(t) & \text{if } \text{MonthFromTime}(t)=8 \\
\text{DayWithinYear}(t)-272-\text{InLeapYear}(t) & \text{if } \text{MonthFromTime}(t)=9 \\
\text{DayWithinYear}(t)-303-\text{InLeapYear}(t) & \text{if } \text{MonthFromTime}(t)=10 \\
\text{DayWithinYear}(t)-333-\text{InLeapYear}(t) & \text{if } \text{MonthFromTime}(t)=11 
\end{cases}
\]

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 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:

- \( \text{HourFromTime}(t) = \text{floor}(t / \text{msPerHour}) \text{ modulo HoursPerDay} \)
- \( \text{MinFromTime}(t) = \text{floor}(t / \text{msPerMinute}) \text{ modulo MinutesPerHour} \)
- \( \text{SecFromTime}(t) = \text{floor}(t / \text{msPerSecond}) \text{ modulo SecondsPerMinute} \)
- \( \text{msFromTime}(t) = t \text{ modulo msPerSecond} \)

where

\[
\begin{align*}
\text{HoursPerDay} &= 24 \\
\text{MinutesPerHour} &= 60 \\
\text{SecondsPerMinute} &= 60 \\
\text{msPerSecond} &= 1000 \\
\text{msPerMinute} &= 60000 = \text{msPerSecond} \times \text{SecondsPerMinute} \\
\text{msPerHour} &= 3600000 = \text{msPerMinute} \times \text{MinutesPerHour}
\end{align*}
\]

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 \( t = h \times \text{msPerHour} + m \times \text{msPerMinute} + s \times \text{msPerSecond} + \text{milli} \), performing the arithmetic according to IEEE 754 rules (that is, as if using the ECMAScript operators \( \times \) 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 d 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.

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 × 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.

NOTE Examples of extended years:

- +000001-01T00:00:00Z 1 B.C.
- +001970-01T00:00:00Z 1970 A.D.
- +200909-12T00:00:00Z 2009 A.D.
- +287396-10-12T08:59:00.992Z 287396 A.D.

-283457-03-21T15:00:59.008Z 283458 B.C.
-000001-01T00:00:00Z 2 B.C.
+000000-01-01T00:00:00Z 1 B.C.
+000001-01T00:00:00Z 1 A.D.
+001970-01T00:00:00Z 1970 A.D.
+002009-12-15T00:00:00Z 2009 A.D.
+287396-10-12T08:59:00.992Z 287396 A.D.
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 uninitialized [[DateValue]] internal slot, Date initializes 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 initialization.

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 initialize 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 function 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(dt).
   g. If hours is supplied then let h be ToNumber(hours); else let h be 0.
   h. ReturnIfAbrupt(h).
   i. If minutes is supplied then let min be ToNumber(minutes); else let min be 0.
   j. ReturnIfAbrupt(min).
   k. If seconds is supplied then let s be ToNumber(seconds); else let s be 0.
   l. ReturnIfAbrupt(s).
   m. If ms is supplied then let milli be ToNumber(ms); else let milli be 0.
   n. ReturnIfAbrupt(milli).
   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. Let now be the Number that is the time value (UTC) identifying the current time.
   b. Return ToDateString(now).

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 function 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(dt).
   g. If hours is supplied then let h be ToNumber(hours); else let h be 0.
   h. ReturnIfAbrupt(h).
   i. If minutes is supplied then let min be ToNumber(minutes); else let min be 0.
   j. ReturnIfAbrupt(min).
   k. If seconds is supplied then let s be ToNumber(seconds); else let s be 0.
   l. ReturnIfAbrupt(s).
   m. If ms is supplied then let milli be ToNumber(ms); else let milli be 0.
   n. ReturnIfAbrupt(milli).
   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.
1. Let \( numberOfArgs \) be the number of arguments passed to this function call.
2. Assert: \( numberOfArgs = 1 \).
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. Let \( now \) be the Number that is the time value (UTC) identifying the current time.
   b. Return ToDateString(\( now \)).

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 function 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. Let \( now \) be the Number that is the time value (UTC) identifying the current time.
   b. Return ToDateString(\( now \)).

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 Construct(\( F, argumentsList \)).

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. Unrecognizable 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(\( \text{year} \)).
2. ReturnIfAbrupt(\( y \)).
3. Let \( m \) be ToNumber(\( \text{month} \)).
4. ReturnIfAbrupt(\( m \)).
5. If \( \text{date} \) is supplied then let \( dt \) be ToNumber(\( \text{date} \)); else let \( dt \) be 1.
6. ReturnIfAbrupt(\( dt \)).
7. If \( \text{hours} \) is supplied then let \( h \) be ToNumber(\( \text{hours} \)); else let \( h \) be 0.
8. ReturnIfAbrupt(\( h \)).
9. If \( \text{minutes} \) is supplied then let \( \text{min} \) be ToNumber(\( \text{minutes} \)); else let \( \text{min} \) be 0.
10. ReturnIfAbrupt(\( \text{min} \)).
11. If \( \text{seconds} \) is supplied then let \( s \) be ToNumber(\( \text{seconds} \)); else let \( s \) be 0.
12. ReturnIfAbrupt(\( s \)).
13. If \( \text{milliseconds} \) is supplied then let \( \text{milli} \) be ToNumber(\( \text{milliseconds} \)); else let \( \text{milli} \) be 0.
14. ReturnIf Abrupt(\( \text{milli} \)).
15. If \( y \) is not \( \text{NaN} \) and \( 0 \leq \text{ToInteger}(y) \leq 99 \), then let \( \text{yr} \) be \( 1900 + \text{ToInteger}(y) \); otherwise, let \( \text{yr} \) be \( y \).
16. Return TimeClip(MakeDate(MakeDay(\( \text{yr} \), \( m \), \( dt \)), MakeTime(\( h \), \( \text{min} \), \( s \), \( \text{milli} \))).

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 \( \text{obj} \) be OrdinaryCreateFromConstructor(\( F \), "\[\[\text{DatePrototype}\]\]", (\[\[\text{DateValue}\]\])).
2. 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** \[\[\text{DateValue}\]\] is initially assigned the value \text{undefined} as a flag to indicate that the instance has not yet been initialized 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 \[\[\text{DateValue}\]\] internal slot.

The value of the \[\[\text{Prototype}\]\] 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 \[\[\text{DateValue}\]\] internal slot that has been initialized to a time value.

The abstract operation thisTimeValue(value) performs the following steps:

1. If Type(value) is Object and value has a \[\[\text{DateValue}\]\] internal slot, then
   a. Let \( n \) be the value of value's \[\[\text{DateValue}\]\] 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. If the Type of the `this` value is not `Object`, a `TypeError` exception is thrown. 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 \texttt{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 \texttt{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 \texttt{Date.prototype.getTime ( )}

1. Return this time value.

20.3.4.11 \texttt{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 \(( t - \text{LocalTime}(t)) / \text{msPerMinute}\).

20.3.4.12 \texttt{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 \texttt{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 \texttt{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 \( \text{NaN} \), return \( \text{NaN} \).
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 \( \text{NaN} \), return \( \text{NaN} \).
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 \( \text{NaN} \), return \( \text{NaN} \).
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 \( \text{NaN} \), return \( \text{NaN} \).
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 \( \text{NaN} \), return \( \text{NaN} \).
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(\( \text{date} \)).
3. ReturnIfAbrupt(\( dt \)).
4. Let \( \text{newDate} \) be MakeDate(MakeDay(YearFromTime(\( t \)), MonthFromTime(\( t \)), \( dt \)), TimeWithinDay(\( t \))).
5. Let \( a \) be TimeClip(UTC(\( \text{newDate} \))).
6. Set the [[DateValue]] internal slot of this Date object to \( a \).
7. Return \( a \).

20.3.4.21 Date.prototype.setFullYear ( year [, month [, date ]] )

1. Let \( t \) be the result of LocalTime(this time value); but if this time value is \( \text{NaN} \), let \( t \) be \( +0 \).
2. Let \( y \) be ToNumber(\( \text{year} \)).
3. ReturnIfAbrupt(\( y \)).
4. If \( \text{month} \) is not specified, then let \( m \) be MonthFromTime(\( t \)); otherwise, let \( m \) be ToNumber(\( \text{month} \)).
5. ReturnIfAbrupt(m).
6. If date is not specified, then let dt be DateFromTime(t); otherwise, let dt be ToNumber(date).
7. ReturnIfAbrupt(dt).
8. Let newDate be MakeDate(MakeDay(y, m, dt), TimeWithinDay(t)).
9. Let a be TimeClip(UTC(newDate)).
10. Set the [[DateValue]] internal slot of this Date object to a.

The length property of the setFullYear method is 3.

NOTE  If month is not specified, this method behaves as if month were specified with the value getMonth(). If date is not specified, it behaves as if date were specified with the value getDate().

20.3.4.22 Date.prototype.setHours ( hour [, min [, sec [, ms ]]] )

1. Let t be the result of LocalTime(this time value).
2. Let h be ToNumber(hour).
3. ReturnIfAbrupt(h).
4. If min is not specified, then let m be MinFromTime(t); otherwise, let m be ToNumber(min).
5. ReturnIfAbrupt(m).
6. If sec is not specified, then let s be SecFromTime(t); otherwise, let s be ToNumber(sec).
7. ReturnIfAbrupt(s).
8. If ms is not specified, then let milli be msFromTime(t); otherwise, let milli be ToNumber(ms).
9. ReturnIfAbrupt(milli).
10. Let date be MakeDate(Day(t), MakeTime(h, m, s, milli)).
11. Let a be TimeClip(UTC(date)).
12. Set the [[DateValue]] internal slot of this Date object to a.

The length property of the setHours method is 4.

NOTE  If min is not specified, this method behaves as if min were specified with the value getMinutes(). If sec is not specified, it behaves as if sec were specified with the value getSeconds(). If ms is not specified, it behaves as if ms were specified with the value getMilliseconds().

20.3.4.23 Date.prototype.setMilliseconds ( ms )

1. Let t be the result of LocalTime(this time value).
2. Let ms be ToNumber(ms).
3. ReturnIfAbrupt(ms).
4. Let time be MakeTime(HourFromTime(t), MinFromTime(t), SecFromTime(t), ms).
5. Let a be TimeClip(UTC(MakeDate(Day(t), time))).
6. Set the [[DateValue]] internal slot of this Date object to a.
7. Return a.

20.3.4.24 Date.prototype.setMinutes ( min [, sec [, ms ]]] )

1. Let t be the result of LocalTime(this time value).
2. Let m be ToNumber(min).
3. ReturnIfAbrupt(m).
4. If sec is not specified, then let s be SecFromTime(t); otherwise, let s be ToNumber(sec).
5. ReturnIfAbrupt(s).
6. If `ms` is not specified, then let `milli` be `msFromTime(t)`; otherwise, let `milli` be `ToNumber(ms)`.
7. ReturnIfAbrupt(`milli`).
8. Let `date` be `MakeDate(Day(t), MakeTime(HourFromTime(t), m, s, `milli`))`.
9. Let `u` be `TimeClip(UTC(date))`.
10. Set the `[[DateValue]]` internal slot of this Date object to `u`.
11. Return `u`.

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

NOTE If `sec` is not specified, this method 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()`.

20.3.4.25 Date.prototype.setMonth ( month [, date ] )
1. Let `t` be the result of `LocalTime(this time value)`.
2. Let `m` be `ToNumber(month)`.
3. ReturnIfAbrupt(`m`).
4. If `date` is not specified, then let `dt` be `DateFromTime(t)`; otherwise, let `dt` be `ToNumber(date)`.
5. ReturnIfAbrupt(`dt`).
6. Let `newDate` be `MakeDate(MakeDay(YearFromTime(t), m, dt), TimeWithinDay(t))`.
7. Let `u` be `TimeClip(UTC(newDate))`.
8. Set the `[[DateValue]]` internal slot of this Date object to `u`.
9. Return `u`.

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

NOTE If `date` is not specified, this method behaves as if `date` were specified with the value `getDate()`.

20.3.4.26 Date.prototype.setSeconds ( sec [, ms ] )
1. Let `t` be the result of `LocalTime(this time value)`.
2. Let `s` be `ToNumber(sec)`.
3. ReturnIfAbrupt(`s`).
4. If `ms` is not specified, then let `milli` be `msFromTime(t)`; otherwise, let `milli` be `ToNumber(ms)`.
5. ReturnIfAbrupt(`milli`).
6. Let `date` be `MakeDate(Day(t), MakeTime(HourFromTime(t), MinFromTime(t), 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 `setSeconds` method is 2.

NOTE If `ms` is not specified, this method behaves as if `ms` were specified with the value `getMilliseconds()`.

20.3.4.27 Date.prototype.setTime ( time )
1. Let `t` be `ToNumber(time)`.
2. ReturnIfAbrupt(`time`).
3. Let `v` be `TimeClip(t)`.
4. Set the `[[DateValue]]` internal slot of this Date object to `v`.
5. 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. ReturnIfAbrupt(dt).
5. Let newDate be MakeDate(MakeDay(YearFromTime(t), MonthFromTime(t), dt), TimeWithinDay(t)).
6. Let v be TimeClip(newDate).
7. Set the [[DateValue]] internal slot of this Date object to v.
8. Return v.

20.3.4.29 Date.prototype.setUTCFullYear ( year [, month [, date ] ] )

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. ReturnIfAbrupt(y).
5. If month is not specified, then let m be MonthFromTime(t); otherwise, let m be ToNumber(month).
6. ReturnIfAbrupt(m).
7. If date is not specified, then let dt be DateFromTime(t); otherwise, let dt be ToNumber(date).
8. ReturnIfAbrupt(dt).
9. Let newDate be MakeDate(MakeDay(y, m, dt), TimeWithinDay(t)).
10. Let v be TimeClip(newDate).
11. Set the [[DateValue]] internal slot of this Date object to v.
12. Return v.

The length property of the setUTCFullYear method is 3.

NOTE If month is not specified, this method behaves as if month were specified with the value getUTCMonth(). If date is not specified, it behaves as if date were specified with the value getUTCDate().

20.3.4.30 Date.prototype.setUTCHours ( hour [, min [, sec [, ms ] ] ] )

1. Let t be this time value.
2. ReturnIfAbrupt(t).
3. Let h be ToNumber(hour).
4. ReturnIfAbrupt(h).
5. If min is not specified, then let m be MinFromTime(t); otherwise, let m be ToNumber(min).
6. ReturnIfAbrupt(m).
7. If sec is not specified, then let s be SecFromTime(t); otherwise, let s be ToNumber(sec).
8. ReturnIfAbrupt(s).
9. If ms is not specified, then let milli be msFromTime(t); otherwise, let milli be ToNumber(ms).
10. ReturnIfAbrupt(milli).
11. Let newDate be MakeDate(YearFromTime(t), MonthFromTime(t), Day(t), MakeTime(h, m, s, milli)).
12. Let v be TimeClip(newDate).
13. Set the [[DateValue]] internal slot of this Date object to v.

The length property of the setUTCHours method is 4.
NOTE If min is not specified, this method behaves as if min were specified with the value getUTCMinutes(). If sec is not specified, it behaves as if sec were specified with the value getUTCSeconds(). If ms is not specified, it behaves as if ms were specified with the value getUTCMilliseconds().

20.3.4.31 Date.prototype.setUTCMilliseconds (ms)
1. Let t be this time value.
2. ReturnIfAbrupt(t).
3. Let milli be ToNumber(ms).
4. ReturnIfAbrupt(milli).
5. Let time be MakeTime(HourFromTime(t), MinFromTime(t), SecFromTime(t), milli).
6. Let v be TimeClip(MakeDate(Day(t), time)).
7. Set the [[DateValue]] internal slot of this Date object to v.
8. Return v.

The length property of the setUTCMilliseconds method is 1.

NOTE If sec is not specified, this method behaves as if sec were specified with the value getUTCSeconds(). If ms is not specified, it function behaves as if ms were specified with the value return by getUTCMilliseconds().

20.3.4.32 Date.prototype.setUTCMinutes (min[, sec[, ms]])
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])
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.

NOTE If date is not specified, this method behaves as if date were specified with the value getUTCDate().

20.3.4.34 Date.prototype.setUTCSeconds (sec[, ms])
1. Let t be this time value.
2. ReturnIfAbrupt(t).
3. Let s be ToNumber(sec).
4. If \( ms \) is not specified, then let \( \text{milli} \) be \( \text{msFromTime}(t) \); otherwise, let \( \text{milli} \) be \( \text{ToNumber}(ms) \).

5. Let \( date \) be \( \text{MakeDate}(\text{Day}(t), \text{MakeTime}(\text{HourFromTime}(t), \text{MinFromTime}(t), s, \text{milli})) \).

6. Let \( v \) be \( \text{TimeClip}(date) \).

7. Set the \([\text{DateValue}]\) internal slot of this Date object to \( v \).

8. Return \( v \).

The \texttt{length} property of the \texttt{setUTCSeconds} method is 2.

NOTE If \( ms \) is not specified, this method behaves as if \( ms \) were specified with the value \( \text{getUTCMilliseconds()} \).

20.3.4.35 \texttt{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 \texttt{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 \texttt{RangeError} exception is thrown.

20.3.4.37 \texttt{Date.prototype.toJSON(key)}

This function provides a String representation of a Date object for use by \texttt{JSON.stringify} (24.3.2).

When the \texttt{toJSON} method is called with argument \( key \), the following steps are taken:

1. Let \( O \) be the result of calling \texttt{ToObject}, giving it the \texttt{this} value as its argument.
2. Let \( tv \) be \texttt{ToPrimitive}(\( O \), hint Number).
3. ReturnIfAbrupt(\( tv \)).
4. If \( \text{Type}(tv) \) is Number and \( tv \) is not finite, return \texttt{null}.
5. Return \texttt{Invoke}(\( O \), "toISOString").

NOTE 1 The argument is ignored.

NOTE 2 The \texttt{toJSON} function is intentionally generic; it does not require that its \texttt{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 \texttt{toISOString} method.

20.3.4.38 \texttt{Date.prototype.toLocaleDateString([reserved1[,reserved2]])}

An ECMAScript implementation that includes the ECMA-402 Internationalization API must implement the \texttt{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 \texttt{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 positions for anything else.

The length property of the toLocaleDateString method is 0.

20.3.4.39 Date.prototype.toLocaleDateString ([ reserved1 [, reserved2 ] ])

An ECMAScript implementation that includes the ECMA-402 Internationalization 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 positions for anything else.

The length property of the toLocaleDateString method is 0.

20.3.4.40 Date.prototype.toLocaleTimeString ([ reserved1 [, reserved2 ] ])

An ECMAScript implementation that includes the ECMA-402 Internationalization 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 positions for anything else.

The length property of the toLocaleTimeString method is 0.

20.3.4.41 Date.prototype.toString ()

The following steps are performed:

1. Let O be this Date object.
2. If O does not have a [[DateValue]] internal slot, then
   a. Let tv be NaN.
3. Else,
a. Let tv be this time value.
4. Return ToDateString(tv).

NOTE For any Date object d whose milliseconds amount is zero, the result of Date.parse(d.toString()) is equal to d.valueOf(). See 20.3.3.2.

20.3.4.41.1 Runtime Semantics: ToDateString(tv) Abstract Operation

1. Assert: Type(tv) is Number.
2. If tv is NaN, then return “Invalid Date”.
3. Return an implementation-dependent String value that represents tv as a date and time in the current time zone using a convenient, human-readable form.

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 “T” 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 object in that they treat “default” as being equivalent to “string”. All other built-in ECMAScript objects treat “default” as being equivalent to “number”.

When the @@toPrimitive method is called with argument hint, the following steps are taken:

1. Let O be the this 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 \{ \[[\text{Writable}]\]: false, \[[\text{Enumerable}]\]: false, \[[\text{Configurable}]\]: 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 \[[\text{DateValue}]\] internal slot. The \[[\text{DateValue}]\] 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 uninitialized \[[\text{StringData}]\] internal slot, it initializes the this value using the argument value. This permits String to be used both to perform type conversion and to perform constructor instance initialization.

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 initialize the \[[\text{StringData}]\] 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 Type(O) is Object and O has a \[[\text{StringData}]\] internal slot and the value of \[[\text{StringData}]\] is undefined, then
   a. Let initializing be true.
3. Else, let initializing be false.
4. If no arguments were passed to this function invocation, then let s be "".
5. Else,
   a. If initializing is false and Type(O) is Symbol, then return SymbolDescriptiveString(O).
   b. Let s be ToString(value).
6. ReturnIfAbrupt(s).
7. If initializing is true, then
   a. Let extensible be IsExtensible(O).
   b. ReturnIfAbrupt(extensible).
   c. If extensible is false, then throw a TypeError exception.
   d. Let length be the number of code unit elements in s.
   e. Let status be the result of DefinePropertyOrThrow(O, "length", [[Value]]: length, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false).
   f. ReturnIfAbrupt(status).
   g. Set the value of O’s \[[\text{StringData}]\] internal slot to s.
   h. Return O.
8. 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 initializes 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 `Construct(F, argumentsList)`.

If `String` is implemented as an ECMA Script 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 any number of arguments which form the rest parameter `codeUnits`. The following steps are taken:

1. Let `codeUnits` be a List containing the arguments passed to this function.
2. Let `length` be the number of elements in `codeUnits`.
3. Let `elements` be a new List.
4. Let `nextIndex` be 0.
5. Repeat while `nextIndex < length`
   
   a. Let `next` be `codeUnits[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 any number of arguments which form the rest parameter `codePoints`. The following steps are taken:

1. Let `codePoints` be a List containing the arguments passed to this function.
2. Let `length` be the number of elements in `codePoints`.
3. Let `elements` be a new List.
4. Let `nextIndex` be 0.
5. Repeat while `nextIndex < length`
   
   a. Let `next` be `codePoints[nextIndex]`.
   b. Let `nextCP` be `ToNumber(next)`.

Commented [AWB992]: Should we provide `fromCodeUnit` as an alias for this property and label `fromCharCode` as obsolete.
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-16Encoding (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 1.

21.1.2.3 String.prototype

The initial value of 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 List substitutions. The following steps are taken:

1. Let substitutions be a List consisting of all of the arguments passed to this function, starting with the second argument. If fewer than two arguments were passed, the List is empty.
2. Let numberOfSubstitutions be the number of elements in substitutions.
3. Let cooked be ToObject(callSite).
4. ReturnIfAbrupt(cooked).
5. Let rawValue be the result of Get(cooked, "raw").
6. Let raw be ToObject(rawValue).
7. ReturnIfAbrupt(raw).
8. Let len be the result of Get(raw, "length").
9. Let literalSegments be ToLength(len).
10. ReturnIfAbrupt(literalSegments).
11. If literalSegments < 0, then return the empty string.
12. Let stringElements be a new List.
13. Let nextIndex be 0.
14. 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 stringElements has no elements, the empty string is returned.
   g. If nextIndex < numberOfSubstitutions, then let next be substitutions[nextIndex].
   h. Else, let next be the empty String.
   i. Let nextSub be ToString(next).
   j. ReturnIfAbrupt(nextSub).
   k. Append in order the code unit elements of nextSub to the end of stringElements.
   l. 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 (12.3.7). 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 F performs the following steps:

1. Let F be the this value.
2. Let proto be the result of GetPrototypeFromConstructor( F, "%StringPrototype% ").
3. ReturnIfAbrupt( proto ).
4. Let obj be the result of calling StringCreate ( proto ).
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  [[StringData]] is initially assigned the value undefined as a flag to indicate that the instance has not yet been initialized by the String constructor. This flag value is never directly exposed to ECMAScript 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 initialized to a String value.

The abstract operation thisStringValue( value ) performs the following steps:

1. If Type( value ) is String, return value.
2. If Type( value ) is Object and value has a [[StringData]] internal slot, then
   a. Let s be the value of value's [[StringData]] internal slot.
   b. If s is not undefined, then return s.
3. Throw a TypeError exception.

The phrase "this String value" within the specification of a method refers to the result returned by calling the abstract operation thisStringValue with the this value of the method invocation 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 pos in the String value resulting from converting this object to a String. If there is no element at that position, the result is the empty String.

The result is a String value, not a String object.
If `pos` is a value of Number type that is an integer, then the result of `x.charAt(pos)` is equal to the result of `x.substring(pos, pos+1)`.

When the `charAt` method is called with one argument `pos`, the following steps are taken:

1. Let `O` be `RequireObjectCoercible(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 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.charCodeAt (pos)

**NOTE** Returns a Number (a nonnegative integer less than 2¹⁶) that is the code unit value of the string element at position `pos` in the String resulting from converting this object to a String. If there is no element at that position, the result is `NaN`.

When the `charCodeAt` method is called with one argument `pos`, the following steps are taken:

1. Let `O` be `RequireObjectCoercible(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 `charCodeAt` function is intentionally generic; it does not require that its `this` value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

### 21.1.3.3 String.prototype.codePointAt (pos)

**NOTE** Returns a nonnegative integer Number less than 1114112 (0x110000) that is the code point value of the UTF-16 encoded code point 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 `RequireObjectCoercible(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 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.

When the concat method is called with zero or more arguments the following steps are taken:

1. Let O be RequireObjectCoercible(this value).
2. Let S be ToString(O).
3. ReturnIfAbrupt(S).
4. Let args be a List whose elements are the arguments passed to this function.
5. Let R be S.
6. Repeat, while args is not empty
   a. Remove the first element from args and let next be the value of that element.
   b. Let nextString be ToString(next).
   c. ReturnIfAbrupt(nextString).
   d. Let R be the String value consisting of the string elements in the previous value of R followed by the string elements of nextString.
7. Return R.

The length property of the concat method is 1.

NOTE The concat function is intentionally generic; it does not require that its this value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

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 ] )

The contains method takes two arguments, searchString and position, and performs the following steps:

1. Let O be RequireObjectCoercible(this value).
2. Let S be ToString(O).
3. ReturnIfAbrupt(S).
4. If Type(searchString) is Object, then
   a. Let isRegExp be ToBoolean(Get(searchString, @@isRegExp)).
   b. ReturnIfAbrupt(isRegExp).
   c. If isRegExp is not true, then 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)`.  
11. Let `searchLen` be the number of elements in `searchStr`.  
12. 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 code unit at position `k+j` of `S` is the same as the code unit at position `j` of `searchStr`, return `true`; but if there is no such integer `k`, return `false`.  

The `length` property of the `contains` method is 1.  

**NOTE 1** If `searchString` appears as a substring of the result of converting this object to a String, at one or more positions that are greater than or equal to `position`, then return `true`; otherwise, returns `false`. If `position` is `undefined`, 0 is assumed, so as to search all of the String.  

**NOTE 2** Throwing an exception if the first argument is a RegExp is specified in order to allow future editions to define extensions that allow such argument values.  

**NOTE 3** 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:  
1. Let `O` be `RequireObjectCoercible(this value)`.  
2. Let `S` be `ToString(O)`.  
3. ReturnIfAbrupt(`S`).  
4. If `Type(searchString)` is Object, then  
   a. Let `isRegExp` be `ToBoolean(Get(searchString, @@isRegExp))`.  
   b. ReturnIfAbrupt(`isRegExp`).  
   c. If `isRegExp` is not `true`, then throw a `TypeError` exception.  
5. Let `searchStr` be `ToString(searchString)`.  
6. ReturnIfAbrupt(`searchStr`).  
7. Let `len` be the number of elements in `S`.  
8. If `endPosition` is `undefined`, let `pos` be `len`, else let `pos` be `ToInteger(endPosition)`.  
9. ReturnIfAbrupt(`pos`).  
10. Let `end` be `min(max(pos, 0), len)`.  
11. Let `searchLength` be the number of elements in `searchStr`.  
12. Let `start` be `end - searchLength`.  
13. If `start` is less than 0, return `false`.  
14. If the `searchLength` sequence of elements of `S` starting at `start` is the same as the full element sequence of `searchStr`, return `true`.  
15. Otherwise, return `false`.  

The `length` property of the `endsWith` method is 1.  

**NOTE 1** Returns `true` if the sequence of elements of `searchString` converted to a String is the same as the corresponding elements of this object (converted to a String) starting at `endPosition` – `length(this)`. Otherwise returns `false`.
NOTE 2 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 ])

NOTE 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 `RequireObjectCoercible(this value)`.  
2. Let `S` be `ToString(O)`.  
3. ReturnIfAbrupt(S).  
4. Let `searchStr` be `ToString(searchString)`.  
5. ReturnIfAbrupt(searchString).  
6. Let `pos` be `ToInteger(position)` (If `position` is `undefined`, this step produces the value `0`).  
7. ReturnIfAbrupt(pos).  
8. Let `len` be the number of elements in `S`.  
9. Let `start` be `min(max(pos, 0), len)`.  
10. Let `searchLen` be the number of elements in `searchStr`.  
11. Return the smallest possible integer `k` not smaller than `start` such that `k+searchLen` is not greater than `len`, and for all nonnegative integers `j` less than `searchLen`, the code unit at position `k+j` of `S` is the same as the code unit at position `j` of `searchStr`, but if there is no such integer `k`, then return the value `-1`.  

The `length` property of the `indexOf` method is `1`.

NOTE The `indexOf` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

21.1.3.9 String.prototype.lastIndexOf ( searchString [, position ])

NOTE 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 `RequireObjectCoercible(this value)`.  
2. Let `S` be `ToString(O)`.  
3. ReturnIfAbrupt(S).  
4. Let `searchStr` be `ToString(searchString)`.  
5. ReturnIfAbrupt(searchString).  
6. Let `numPos` be `ToNumber(position)` (If `position` is `undefined`, this step produces the value `NaN`).  
7. ReturnIfAbrupt(numPos).  
8. If `numPos` is `NaN`, let `pos` be `+∞`; otherwise, let `pos` be `ToInteger(numPos)`.  
9. Let `len` be the number of elements in `S`.  
10. Let `start` be `min(max(pos, 0), len)`.  

© Ecma International 2014
11. Let searchLen be the number of elements in searchStr.
12. Return the largest possible nonnegative integer k not larger than start such that k + searchLen is not
    greater than len, and for all nonnegative integers j less than searchLen, the code unit at position k+j
    of S is the same as the code unit at position j of searchStr; but if there is no such integer k, then
    return the value -1.

The length property of the lastIndexOf method is 1.

NOTE The lastIndexOf function is intentionally generic; it does not require that its this value be a String
    object. Therefore, it can be transferred to other kinds of objects for use as a method.

21.1.3.10 String.prototype.localeCompare ( that [, reserved1 [, reserved2 ] ] )

An ECMAScript implementation that includes the ECMA-402 Internationalization API must implement the
localeCompare method as specified in the ECMA-402 specification. If an ECMAScript implementation
does not include the ECMA-402 API the following specification of the localeCompare method is used.

When the localeCompare method is called with argument that, it returns a Number other than NaN that
represents the result of a locale-sensitive String comparison of the this value (converted to a String) with
that (converted to a String). The two Strings are S and That. The two Strings are compared in an
implementation-defined fashion. The result is intended to order String values in the sort order specified by
a host default locale, and will be negative, zero, or positive, depending on whether S comes before That in
the sort order, the Strings are equal, or S comes after That in the sort order, respectively.

Before perform the comparisons the following steps are performed to prepare the Strings:

1. Let O be RequireObjectCoercible(this value).
2. Let S be ToString(O).
3. ReturnIfAbrupt(S).
4. Let That be ToString(that).
5. ReturnIfAbrupt(That).

The meaning of the optional second and third parameters to this method are defined in the ECMA-402
specification; implementations that do not include ECMA-402 support must not assign any other
interpretation to those parameter positions.

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. This function
must treat Strings that are canonically equivalent according to the Unicode standard as identical and must
return 0 when comparing Strings that are considered canonically equivalent.

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. However, regardless of the host provided comparision capabilities, this function must
must treat Strings that are canonically equivalent according to the Unicode standard as identical. It is 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 `RequireObjectCoercible(this value).`
2. Let `S` be `ToString(O).`
3. ReturnIfAbrupt(S).
4. If `Type(regexp)` is Object and `HasProperty(regexp, @isRegExp)` is `true`, then let `rx` be `regexp`;
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 ])`

When the `normalize` method is called with one argument `form`, the following steps are taken:

1. Let `O` be `RequireObjectCoercible(this value).` 
2. Let `S` be `ToString(O).` 
3. ReturnIfAbrupt(S). 
4. If `form` is not provided or `form` is `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, Unicode Normalization 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 `RequireObjectCoercible(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 \( count \) times.

**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 `RequireObjectCoercible(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 Invoke(searchValue, "replace", (string, replaceValue))`.
5. Let `searchString` be `ToString(searchValue)`.
6. `ReturnIfAbrupt(searchString)`.
7. Let `functionalReplace` be `IsCallable(replaceValue)`.
8. If `functionalReplace` is `false`, then
   a. Let `replaceValue` be `ToString(replaceValue)`.
   b. `ReturnIfAbrupt(replaceValue)`.
9. 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`.
10. If `functionalReplace` 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)`.
11. Else,
    a. Let `captures` be an empty List.
    b. Let `replStr` be `GetReplaceSubstitution(matched, string, pos, captures, replaceValue)`.
12. Let `tailPos` be `pos + the number of code units in matched`.
13. 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.

**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, replacement)` 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 nonnegative integer.
7. Assert: captures is a possibly empty List of Strings.
8. Assert: Type(replacement) is String.
9. Let tailPos be position + matchLength.
10. Let m be the number of elements in captures.
11. Let result be a String value derived from replacement by copying code unit elements from replacement to result while performing replacements as specified in Table 41. These $ replacements are done left-to-right, and, once such a replacement is performed, the new replacement text is not subject to further replacements.
12. Return result.

Table 41 — 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>$</td>
<td>$</td>
</tr>
<tr>
<td>0x0024, 0x0026</td>
<td>$</td>
<td>matched</td>
</tr>
<tr>
<td>0x0024, 0x0060</td>
<td>$</td>
<td>If 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 point is at index position - 1.</td>
</tr>
<tr>
<td>0x0024, 0x0027</td>
<td>$</td>
<td>If tailPos &gt; stringLength, the replacement is the empty String. Otherwise the replacement is the substring of string that starts at index tailPos and continues to the end of string.</td>
</tr>
<tr>
<td>0x0024, N where 0x0031 ≤ N ≤ 0x0039</td>
<td>$n</td>
<td>The n\textsuperscript{th} element of captures, where n is a single digit in the range 1 to 9.</td>
</tr>
<tr>
<td>0x0024, N where 0x0030 ≤ N ≤ 0x0039</td>
<td>$nn</td>
<td>The mn\textsuperscript{th} element of captures, where mn is a two-digit decimal number in the range 01 to 99. If mn &gt; m and the mn\textsuperscript{th} element of captures is undefined, use the empty String instead. If mn &gt; m, the result is implementation-defined.</td>
</tr>
<tr>
<td>0x0024</td>
<td>$</td>
<td>in any context that does not match any of the above.</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 RequireObjectCoercible(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;
   b. Otherwise, let rx be String.prototype.
5. Else,
422

© Ecma International 2014

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 slice method takes two arguments, start and end, and returns a substring of the result of converting this object to a String, starting from element position start and running to, but not including, element position end (or through the end of the String if end is undefined). If start is negative, it is treated as sourceLength-start where sourceLength is the length of the String. If end is negative, it is treated as sourceLength-end where sourceLength is the length of the String. The result is a String value, not a String object. The following steps are taken:
1. Let O be RequireObjectCoercible(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. If intStart is negative, let from be max(len + intStart, 0); else let from be min(intStart, len).
8. If intEnd is negative, let to be max(len + intEnd, 0); else let to be min(intEnd, len).
9. Let span be max(to - from, 0).
10. Return a String value containing span consecutive elements from S beginning with the element at position from.

The length property of the slice method is 2.

NOTE The slice function is intentionally generic; it does not require that its this value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

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 separator; these occurrences are not part of any substring in the returned array, but serve to divide up the String value. The value of separator may be a String of any length or it may be a RegExp object.

When the split method is called, the following steps are taken:
1. Let O be RequireObjectCoercible(this value).
2. ReturnIfAbrupt(O).
3. If Type(separator) is Object and HasProperty(separator, @@isRegExp) is true, then
   a. Return the result of Invoke(separator, "split", (O, limit))
4. Let S be ToString(O).
5. ReturnIfAbrupt(S).
6. Let A be the result of the abstract operation ArrayCreate with argument 0.
7. Let lengthA be 0.
8. If limit is undefined, let lim = 2^31-1; else let lim = ToLength(limit).
9. Let s be the number of elements in S.
10. Let \( p = 0 \).
11. Let \( R \) be `toString(separator)`.
12. ReturnIfAbrupt(\( R \)).
13. If \( lim = 0 \), return \( A \).
14. If \( separator \) is `undefined`, then
   a. Call `createDataProperty(A, "$\zero"`, \( S \)).
   b. Assert: The above call will never result in an abrupt completion.
   c. Return \( A \).
15. If \( s = 0 \), then
   a. Let \( z \) be the result of `splitMatch(S, 0, \( R \))`.
   b. If \( z \) is `false`, return \( A \).
   c. Call `createDataProperty(A, "$\zero"`, \( S \)`).
   d. Assert: The above call will never result in an abrupt completion.
   e. Return \( A \).
16. Let \( q = p \).
17. Repeat, while \( q \neq s \)
   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 \( e \neq p \),
          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 \( A \).
          6. Let \( p = e \).
   d. 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 `["a","b"]`, while `"ab".split(/a+/)` evaluates to the array `["","b"]`.)

If the this object is (or converts to) the empty String, the result depends on whether \( separator \) can match the empty String. If it can, the result array contains no elements. Otherwise, the result array contains one element, which is the empty String.

If \( separator \) is a regular expression that contains capturing parentheses, then each time \( separator \) is matched the results (including any `undefined` results) of the capturing parentheses are spliced into the output array. For example, `"A<B>bold</B>and<CODE>coded</CODE>".split(/(<\(/\)?([^\<>]+)>)/)` evaluates to the array...
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 RequireObjectCoercible(this value).
2. Let $S$ be ToString($O$).
3. ReturnIfAbrupt($S$).
4. If Type(searchString) is Object, then
   a. Let isRegExp be ToBoolean(Get(searchString, @@isRegExp)).
   b. ReturnIfAbrupt(isRegExp).
   c. If isRegExp is not true, then 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).
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 if `end` is `undefined`). The result is a String value, not a String object.

If either argument is `NaN` or negative, it is replaced with zero; if either argument is larger than the length of the String, it is replaced with the length of the String.

If `start` is larger than `end`, they are swapped.

The following steps are taken:

1. Let `O` be `RequireObjectCoercible(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 `min(finalStart, finalEnd)`. 
10. Let `to` be `max(finalStart, finalEnd)`. 
11. Return a String whose length is `to - from`, containing code units from `S`, namely the code units with indices `from` through `to - 1`, in ascending order.

The length property of the `substring` method is `2`.

NOTE  The `substring` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

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 `RequireObjectCoercible(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-16Encoding (10.1.1) of c.  
8. Let L be a String whose elements are, in order, the elements of `cuList`.  
9. Return L.

The result must be derived according to the 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 `toUpperCase` and `toLowerCase` have context-sensitive behaviour, the functions are not symmetrical. In other words, s.toUpperCase().toLowerCase() is not necessarily equal to s.toLowerCase().

**NOTE 2** The `toLowerCase` function is intentionally generic; it does not require that its this value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

**21.1.3.23 String.prototype.toString ( )**

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

1. Let s be thisStringValue(this value).  
2. Return s.

**NOTE** For a String object, the `toString` method happens to return the same thing as the `valueOf` method.

**21.1.3.24 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 `String.prototype.toLowerCase`, except that code points are mapped to their `uppercase` equivalents as specified in the Unicode Character Database.

NOTE The `toUpperCase` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

### 21.1.3.25 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 `O` be `RequireObjectCoercible(this value)`.
2. Let `S` be `ToString(O)`.
3. Return `ReturnIfAbrupt(S)`.
4. Let `T` be a String value that is a copy of `S` with both leading and trailing white space removed. The definition of white space is the union of `WhiteSpace` and `LineTerminator`. When determining whether a Unicode 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 `T`.

NOTE The `trim` function is intentionally generic; it does not require that its `this` value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

### 21.1.3.26 String.prototype.valueOf ()

When the `valueOf` method is called, the following steps are taken:
1. Let `s` be `thisStringValue(this value)`.
2. Return `s`.

### 21.1.3.27 String.prototype[@@iterator] ()

When the `@@iterator` method is called it returns an Iterator object (25.1.1.2) 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 `RequireObjectCoercible(this value)`.
2. Let `S` be `ToString(O)`.
3. Return `ReturnIfAbrupt(S)`.
4. Return `CreateStringIterator(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 initialized, 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. Assert: Type(string) is String.
2. Let iterator be the result of ObjectCreate(%StringIteratorPrototype%, ([IteratedString], [StringIteratorNextIndex])).
3. Set iterator’s ([IteratedString]) internal slot to string.
4. Set iterator’s ([StringIteratorNextIndex]) internal slot to 0.
5. Return iterator.

21.1.5.2 %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 %IteratorPrototype% intrinsic object (25.1.2). 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 an String Iterator Instance (21.1.5.3), throw a TypeError exception.
4. Let s be the value of the [[IteratedString]] internal slot of O.
5. Let position be the value of the [[StringIteratorNextIndex]] internal slot of O.
6. Let len be the number of elements in s.
7. If position ≥ len, then
   a. Set the value of the [[IteratedString]] internal slot of O to undefined.
   b. Return CreateIterResultObject(undefined, true).
8. If first < 0xD800 or first > 0xDBFF or position+1 = len, then let resultString be the string consisting of the single code unit first.
10. If first < 0xD800 or first > 0xDBFF or position+1 = len, 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 `[[StringIteratorNextIndex]]` 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 44.

Table 42 — Internal Slots of String Iterator Instances

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[IteratedString]]</code></td>
<td>The String value whose elements are being iterated.</td>
</tr>
<tr>
<td><code>[[StringIteratorNextIndex]]</code></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**

```
Pattern::= Disjunction
Disjunction::= Alternative
Alternative::= Disjunction | Alternative
```

Formatted: French (Switzerland)
Alternative ::
  {empty}
  Alternative Term

Term ::
  Assertion
  Atom
  Atom Quantifier

Assertion ::
  ^
  \ b
  \ B
  (? = Disjunction )
  (? ! Disjunction )

Quantifier ::
  QuantifierPrefix
  QuantifierPrefix

QuantifierPrefix ::
  *
  +
  ?
  { DecimalDigits }
  { DecimalDigits , }
  { DecimalDigits , DecimalDigits }

Atom ::
  PatternCharacter
  \ AtomEscape
  CharacterClass
  ( Disjunction )
  ( ? : Disjunction )

SyntaxCharacter :: one of
  ^ $ \ . * + ? ( ) [ ] { } |

PatternCharacter ::
  SourceCharacter but not SyntaxCharacter

AtomEscape ::
  DecimalEscape
  CharacterEscape
  CharacterClassEscape
CharacterEscape\u203A
  ControlEscape
  c ControlLetter
  HexEscapeSequence
  RegExpUnicodeEscapeSequence\u203A
  IdentityEscape\u203A

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\u203A
  [^u] u LeadSurrogate \u TrailSurrogate
  {u} { HexDigits }

LeadSurrogate ::
  HexDigits [match only if the CV of HexDigits is in the inclusive range 0xD800 to 0xDBFF]

TrailSurrogate ::
  HexDigits [match only if the CV of HexDigits is in the inclusive range 0xDC00 to 0xDFFF]

IdentityEscape\u203A
  [^u] SyntaxCharacter
  {u} SourceCharacter but not IdentifierPart
  {u} <ZWJ>
  {u} <ZWNJ>

DecimalEscape ::
  DecimalIntegerLiteral [lookahead a DecimalDigit]

CharacterClassEscape :: one of
  d D s S w W

CharacterClass\u203A
  [lookahead a \+] ClassRanges\u203A
  [^ ClassRanges\u203A ]

ClassRanges\u203A
  [empty]
  NonemptyClassRanges\u203A

NonemptyClassRanges\u203A
  ClassAtom\u203A
  ClassAtom\u203A NonemptyClassRangesNoDash\u203A
  ClassAtom\u203A - ClassAtom\u203A ClassRanges\u203A
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 an "u". A BMP pattern matches against a String interpreted as consisting of a sequence of 16-bit values that are Unicode code points in the range of the Basic Multilingual Plane. 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 single 16-bit Unicode BMP code point. In the context of describing the behaviour of a Unicode pattern “character” means a UTF-16 encoded code point. In either context, “character value” means the numeric value of the code unit or code point.

The semantics of Pattern are defined as if a Pattern was a List of SourceCharacter values where each SourceCharacter corresponds to a Unicode code point. If a BMP pattern contains a non-BMP SourceCharacter the entire pattern is encoded using UTF-16 and the individual code units of that encoding are used as the elements of the List.

NOTE: For example, consider a pattern expressed in source code as the single non-BMP character U+1D11E (MUSICAL SYMBOL G CLEF). Interpreted as a Unicode pattern, it would be a single element (character) List consisting of the single code point 0x1D11E. However, interpreted as a BMP pattern, it is first UTF-16 encoded to produce a two element List consisting of the code units 0xD834 and 0xDD1E.

Patterns are passed to the RegExp constructor as ECMAScript string values in which non-BMP characters are UTF-16 encoded. For example, 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 0xDD1E. 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 point 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 unit or a code point, depending upon the kind of pattern involved. The notation `input[n]` means the `n`th character of `input`, where `n` can range between 0 (inclusive) and `InputLength` (exclusive).
- **InputLength** is the number of characters in `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 RegExp object's `[[OriginalFlags]]` internal slot contains "i" and otherwise is `false`.
- **Multiline** is `true` if the RegExp object's `[[OriginalFlags]]` internal slot contains "m" and otherwise is `false`.
- **Unicode** is `true` if the RegExp 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 references variables which are 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 \( \text{Pattern} \) :: \( \text{Disjunction} \) evaluates as follows:

1. Evaluate \( \text{Disjunction} \) 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 consisting of the sequence of code points of \( str \) interpreted as a UTF-16 encoded Unicode string. Otherwise, let \( Input \) be a List consisting of 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 characters contained in \( Input \). This variable will be used throughout the algorithms in 21.2.2.
   4. Let \( c \) be a Continuation that always returns its State argument as a successful MatchResult.
   5. Let \( cap \) be a List of \( N\text{capturingParens} \) undefined values, indexed 1 through \( N\text{capturingParens} \).
   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. \( \text{RegExp.prototype.exec} \) and other methods 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 \( \text{Disjunction} \) :: \( \text{Alternative} \) evaluates by evaluating \( \text{Alternative} \) to obtain a Matcher and returning that Matcher.

The production \( \text{Disjunction} \) :: \( \text{Alternative} \) \( | \) \( \text{Disjunction} \) evaluates as follows:

1. Evaluate \( \text{Alternative} \) to obtain a Matcher \( m_1 \).
2. Evaluate \( \text{Disjunction} \) to obtain a Matcher \( m_2 \).
3. Return an internal Matcher closure that takes two arguments, a State $x$ and a Continuation $c$, and performs the following:
   1. Call $m_1(x, c)$ and let $r$ be its result.
   2. If $r$ isn't failure, return $r$.
   3. Call $m_2(x, c)$ and return its result.

NOTE The | regular expression operator separates two alternatives. The pattern first tries to match the left Alternative (followed by the sequel of the regular expression); if it fails, it tries to match the right Disjunction (followed by the sequel of the regular expression). If the left Alternative, the right Disjunction, and the sequel all have choice points, all choices in the sequel are tried before moving on to the next choice in the left Alternative. If choices in the left Alternative are exhausted, the right Disjunction is tried instead of the left Alternative. Any capturing parentheses inside a portion of the pattern skipped by | produce undefined values instead of Strings. Thus, for example,

   `/a|ab/.exec("abc")`

   returns the result "a" and not "ab". Moreover,

   `/((a)|(ab))((c)|(bc))/.exec("abc")`

   returns the array

   `[
   "abc", "a", undefined, "bc", undefined, "bc"
   ]`

   and not

   `[
   "abc", "ab", undefined, "ab", c", c", undefined
   ]`

21.2.2.4 Alternative

The production $\text{Alternative} :: \{\text{empty}\}$ 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 $\text{Alternative} :: \text{Alternative Term}$ evaluates as follows:

1. Evaluate $\text{Alternative}$ to obtain a Matcher $m_1$.
2. Evaluate $\text{Term}$ to obtain a Matcher $m_2$.
3. Return an internal Matcher closure that takes two arguments, a State $x$ and a Continuation $c$, and performs the following:
   1. Create a Continuation $d$ that takes a State argument $y$ and returns the result of calling $m_2(y, c)$.
   2. Call $m_1(x, d)$ 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 $\text{Term} :: \text{Assertion}$ evaluates by returning an internal Matcher closure that takes two arguments, a State $x$ and a Continuation $c$, and performs the following:

1. Evaluate $\text{Assertion}$ 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 $\text{Term} :: \text{Atom}$ evaluates as follows:

1. Return the Matcher that is the result of evaluating $\text{Atom}$.
The production Term :: Atom Quantifier evaluates as follows:

1. Evaluate Atom to obtain a Matcher m.
2. Evaluate Quantifier to obtain the three results: an integer min, an integer (or \(\infty\)) max, and Boolean greedy.
3. If max is finite and less than min, then throw a SyntaxError exception.
4. Let parenIndex be the number of left capturing parentheses in the entire regular expression that occur to the left of this production expansion's Term. This is the total number of times the Atom :: ( Disjunction ) production is expanded prior to this production's Term plus the total number of Atom :: ( Disjunction ) productions enclosing this Term.
5. Let parenCount be the number of left capturing parentheses in the expansion of this production's Atom. This is the total number of Atom :: ( Disjunction ) productions enclosed by this production's Atom.
6. Return an internal Matcher closure that takes two arguments, a State x and a Continuation c, and performs the following:
   1. Call RepeatMatcher\((m, \text{min}, \text{max}, \text{greedy}, x, c, \text{parenIndex}, \text{parenCount})\) 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 \(\infty\)) max, a Boolean greedy, a State x, a Continuation c, an integer parenIndex, and an integer parenCount, and performs the following:

1. If max is zero, then call c(x) and return its result.
2. Create an internal Continuation closure d that takes one State argument y and performs the following:
   1. If min is zero and y's endIndex is equal to x's endIndex, then return failure.
   2. If min is zero then let min2 be zero; otherwise let min2 be min–1.
   3. If max is \(\infty\), then let max2 be \(\infty\); otherwise let max2 be max–1.
   4. Call RepeatMatcher\((m, \text{min2}, \text{max2}, \text{greedy}, y, c, \text{parenIndex}, \text{parenCount})\) and return its result.
3. Let cap be a fresh copy of x's captures List.
4. For every integer k that satisfies parenIndex ≤ k and k ≤ parenIndex-parenCount, set cap[k] to undefined.
5. Let e be x's endIndex.
6. Let xr be the State (e, cap).
7. If min is not zero, then call m(xr, d) and return its result.
8. If greedy is false, then
   a. Call c(x) and let z be its result.
   b. If z is not failure, return z.
   c. Call m(xr, d) and return its result.
9. Call m(xr, d) and let z be its result.
10. If z is not failure, return z.
11. Call c(x) and return its result.

NOTE 1 An Atom followed by a Quantifier is repeated the number of times specified by the Quantifier. A Quantifier can be non-greedy, in which case the Atom pattern is repeated as few times as possible while still matching the sequel, or it can be greedy, in which case the Atom pattern is repeated as many times as possible while still matching the sequel. The Atom pattern is repeated rather than the input 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-th) repetition of Atom are tried before moving on to the next choice in the next-to-last (n-1)° 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)° repetition of Atom and so on.

Compare

/a[a-z]{2,4} ./exec("abcdefghijklmnopqrstuvwxyz")
which returns “abcdefghi” with
/a[a-z]{2,4}? ./exec("abcdefghijklmnopqrstuvwxyz")
which returns “abc”.

Consider also

/(aa|aabac|ba|b|c)*/.exec("aabaac")
which, by the choice point ordering above, returns the array
["aaba", "ba"]
and not any of:
["aabaac", "aabaac"]
["aabaac", "c"]
The above ordering of choice points can be used to write a regular expression that calculates the greatest common divisor of two numbers (represented in unary notation). The following example calculates the gcd of 10 and 15:

"aaaaaaaaaa,aaaaaaaaaaaaaaa".replace(/^(a+)|1*\,\,1+$/,"$1")
which returns the gcd in unary notation “aaaaa”.

NOTE 3  Step 5 of the RepeatMatcher clears Atom’s captures each time Atom is repeated. We can see its behaviour in the regular expression

/(z)((a+)?(b+)?(c))*/.exec("zaacbbbcac")
which returns the array
["zaacbbbcac", "z", "ac", "a", undefined, "c"]
and not
["zaacbbbcac", "z", "ac", "a", "bbb", "c"]
because each iteration of the outermost * clears all captured Strings contained in the quantified Atom, which in this case includes capture Strings numbered 2, 3, 4, and 5.

NOTE 4  Step 1 of the RepeatMatcher’s d closure states that, once the minimum number of repetitions has been satisfied, any more expansions of Atom that match the empty 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\+/.exec("basaac")
which returns the array
["ba", ""]

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 :: \ B` evaluates by returning an internal `AssertionTester` closure that takes a State argument `x` and performs the following:
1. Let `e` be `x`'s `endIndex`.
2. Call `IsWordChar(e–1)` and let `a` be the Boolean result.
3. Call `IsWordChar(e)` and let `b` be the Boolean result.
4. If `a` is `true` and `b` is `false`, return `false`.
5. If `a` is `false` and `b` is `true`, return `false`.
6. Return `true`.

The production `Assertion :: ( ? = Disjunction )` evaluates as follows:
1. Evaluate `Disjunction` to obtain a Matcher `m`.
2. Return an internal Matcher closure that takes two arguments, a State `x` and a Continuation `c`, and performs the following steps:
   1. Let `d` be a Continuation that always returns its State argument as a successful MatchResult.
   2. Call `m(x, d)` and let `r` be its result.
   3. If `r` is `failure`, return `failure`.
   4. Let `y` be `x`'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.
   a b c d e f g h i j k l m n o p q r s t u v w x y z
   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
   0 1 2 3 4 5 6 7 8 9
4. Return false.

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 :: * evaluates as follows:
1. Return the two results 0 and $\infty$.

The production QuantifierPrefix :: + evaluates as follows:
1. Return the two results 1 and $\infty$.

The production QuantifierPrefix :: ? evaluates as follows:
1. Return the two results 0 and 1.

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 :: { DecimalDigits , DecimalDigits }` evaluates as follows:

1. Let \( i \) be the MV of the first `DecimalDigits`.
2. Let \( j \) be the MV of the second `DecimalDigits`.
3. Return the two results \( i \) and \( j \).

### 21.2.2.8 Atom

The production `Atom :: PatternCharacter` evaluates as follows:

1. Let \( ch \) be the character matched by `PatternCharacter`.
2. Let \( A \) be a one-element CharSet containing the character \( ch \).
3. Call `CharacterSetMatcher(A, false)` and return its Matcher result.

The production `Atom :: .` evaluates as follows:

1. Let \( A \) be the set of all characters except `LineTerminator`.
2. Call `CharacterSetMatcher(A, false)` and return its Matcher result.

The production `Atom :: \ AtomEscape` evaluates as follows:

1. Return the Matcher that is the result of evaluating `AtomEscape`.

The production `Atom :: CharacterClass` evaluates as follows:

1. Evaluate `CharacterClass` to obtain a CharSet \( A \) and a Boolean `invert`.
2. Call `CharacterSetMatcher(A, invert)` and return its Matcher result.

The production `Atom :: ( Disjunction )` evaluates as follows:

1. Evaluate `Disjunction` to obtain a Matcher \( m \).
2. Let \( parenIndex \) be the number of left capturing parentheses in the entire regular expression that occur to the left of this production expansion's initial left parenthesis. This is the total number of times the `Atom :: ( Disjunction )` production is expanded prior to this production's `Atom` plus the total number of `Atom :: ( Disjunction )` productions enclosing this `Atom`.
3. Return an internal Matcher closure that takes two arguments, a State \( x \) and a Continuation \( c \), and performs the following steps:
   1. Create an internal Continuation closure \( d \) that takes one State argument \( y \) and performs the following steps:
      1. Let \( cap \) be a fresh copy of \( y \)'s captures 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 :: ( ? : Disjunction )` evaluates as follows:

1. Return the Matcher that is the result of evaluating `Disjunction`. 
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 `cc` be the result of `Canonicalize(ch)`.
   5. If `invert` is `false`, then
      a. If there does not exist a member `a` of set `A` such that `Canonicalize(a)` is `cc`, return `failure`.
   6. Else `invert` is `true`, then
      a. If there exists a member `a` of set `A` such that `Canonicalize(a)` is `cc`, return `failure`.
   7. Let `cap` be `x`'s `captures` List.
   8. Let `y` be the State `(e+1, cap)`.
   9. 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 `Unicode` is `true`, then
   a. If the file `CaseFolding.txt` of the Unicode Character Database provides a simple or common case folding mapping for `ch`, then return the result of applying that mapping to `ch`.
   b. Else, return `ch`.
3. Else, then
   a. Assert: `ch` is a UTF-16 code unit.
   b. Let `s` be the ECMAScript String value consisting of the single code unit `ch`.
   c. Let `u` be the same result produced as if by performing the algorithm for `String.prototype.toUpperCase` using `s` as the `this` value.
   d. ReturnIfAbrupt(`u`).
   e. Assert: `u` is a String value.
   f. If `u` does not consist of a single code unit, then return `ch`.
   g. Let `cu` be `u`'s single code unit element.
   h. If `ch`'s code unit value ≥ 128 and `cu`'s code unit value < 128, then return `ch`.
   i. Return `cu`.

NOTE 1 Parentheses of the form `( Disjunction )` serve both to group the components of the `Disjunction` pattern together and to save the result of the match. The result can be used either in a backreference (`\` followed by a nonzero decimal number), referenced in a replace String, or returned as part of an array from the regular expression matching internal procedure. To inhibit the capturing behaviour of parentheses, use the form `(?: Disjunction )` instead.

NOTE 2 The form `(?! Disjunction )` specifies a zero-width positive lookahead. In order for it to succeed, the pattern inside `Disjunction` must match at the current position, but the current position is not advanced before matching the sequel. If `Disjunction` can match at the current position in several ways, only the first one is tried. Unlike other regular expression operators, there is no backtracking into a `(?!` form (this unusual behaviour is inherited from Perl). This only matters when the `Disjunction` contains capturing parentheses and the sequel of the pattern contains backreferences to those captures.
For example,

```
/(?!=(a+))/.exec("baaabac")
```
matches the empty String immediately after the first b and therefore returns the array:

```
["", "aaa"]
```

To illustrate the lack of backtracking into the lookahead, consider:

```
/(?!=(a+))a*b\1/.exec("baaabac")
```
This expression returns

```
["aba", "a"]
```
and not:

```
["aaaba", "a"]
```

NOTE 3 The form `(?! Disjunction)` specifies a zero-width negative lookahead. In order for it to succeed, the pattern inside Disjunction must fail to match at the current position. The current position is not advanced before matching the sequel. Disjunction can contain capturing parentheses, but backreferences to them only make sense from within Disjunction itself. Backreferences to these capturing parentheses from elsewhere in the pattern always return `undefined` because the negative lookahead must fail for the pattern to succeed. For example,

```
/\1(?!a(?!b)c)\2/.exec("baaabac")
```
looks for an a not immediately followed by some positive number n of a's, a b, another n a's (specified by the first `\2`) and a c. The second `\2` is outside the negative lookahead, so it matches against `undefined` and therefore always succeeds. The whole expression returns the array:

```
["baaabac", "ba", undefined, "abaac"]
```

NOTE 4 In case-insignificant matches when `Unicode` is `true`, 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+00D5) to “SS”. It may however map a code point outside the Basic Latin range to a character within, for example, “ß” (U+00D5) to “s”. Such characters are not mapped if `Unicode` is `false`. This prevents Unicode code points such as U+017F and U+212A from matching regular expressions such as `/[a-z]/i`, but they will match `/[a-z]u/i`. 

### 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
   a. Let `ch` be E’s character.
   b. Let `A` be a one-element CharSet containing the character `ch`.
   c. Call `CharacterSetMatcher(A, false)` and return its Matcher result.
3. Assert: E must be an integer.
4. Let n be that integer.
5. If n=0 or n^NcapturingParens then throw a `SyntaxError` exception.
6. Return an internal Matcher closure that takes two arguments, a State x and a Continuation c, and performs the following steps:
   1. Let `cap` be x’s `captures` List.
   2. Let `s` be `cap[n]`.
   3. If s is `undefined`, then call c(s) and return its result.
   4. Let e be s’s `endIndex`.
   5. Let len be s’s length.
   6. Let f be e+len.
   7. If f>InputLength, return `failure`.
8. If there exists an integer \( i \) between 0 (inclusive) and \( \text{len} \) (exclusive) such that \( \text{Canonicalize}(s[i]) \) is not the same character value as \( \text{Canonicalize}(\text{Input}[e+i]) \), then return failure.

9. Let \( y \) be the State \((f, \text{cap})\).

10. Call \( c(y) \) and return its result.

The production \( \text{AtomEscape} :: \text{CharacterEscape} \) evaluates as follows:

1. Evaluate \( \text{CharacterEscape} \) to obtain a character \( ch \).
2. Let \( A \) be a one-element CharSet containing the character \( ch \).
3. Call CharacterSetMatcher\((A, \text{false})\) and return its Matcher result.

The production \( \text{AtomEscape} :: \text{CharacterClassEscape} \) evaluates as follows:

1. Evaluate \( \text{CharacterClassEscape} \) to obtain a CharSet \( A \).
2. Call CharacterSetMatcher\((A, \text{false})\) and return its Matcher result.

NOTE An escape sequence of the form \( \\backslash \) followed by a nonzero decimal number \( n \) matches the result of the \( n \)th set of capturing parentheses (see 21.2.2.11). It is an error if the regular expression has fewer than \( n \) capturing parentheses. If the regular expression has \( n \) or more capturing parentheses but the \( n \)th one is \textit{undefined} because it has not captured anything, then the backreference always succeeds.

21.2.2.10 CharacterEscape

The production \( \text{CharacterEscape} :: \text{ControlEscape} \) evaluates by returning the character according to Table 43.

<table>
<thead>
<tr>
<th>ControlEscape</th>
<th>Character Value</th>
<th>Code Point</th>
<th>Unicode Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td>9</td>
<td>U+0009</td>
<td>CHARACTER TABULATION</td>
<td>&lt;HT&gt;</td>
</tr>
<tr>
<td>( n )</td>
<td>10</td>
<td>U+000A</td>
<td>LINE FEED</td>
<td>&lt;LF&gt;</td>
</tr>
<tr>
<td>( v )</td>
<td>11</td>
<td>U+000B</td>
<td>LINE TABULATION</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>( x )</td>
<td>13</td>
<td>U+000D</td>
<td>CARRIAGE RETURN</td>
<td>&lt;CR&gt;</td>
</tr>
</tbody>
</table>

The production \( \text{CharacterEscape} :: \text{c} \) \text{ControlLetter} evaluates as follows:

1. Let \( ch \) be the character matched by \( \text{ControlLetter} \).
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 \( \text{CharacterEscape} :: \text{HexEscapeSequence} \) evaluates as follows:

1. Return the character whose code is the CV of \( \text{HexEscapeSequence} \).

The production \( \text{CharacterEscape} :: \text{RegExpUnicodeEscapeSequence} \) evaluates as follows:

1. Return the result of evaluating \( \text{RegExpUnicodeEscapeSequence} \).

The production \( \text{CharacterEscape} :: \text{IdentityEscape} \) evaluates as follows:

1. Return the character matched by \( \text{IdentityEscape} \).
The production `RegExpUnicodeEscapeSequence :: u LeadSurrogate \u TrailSurrogate` evaluates as follows:
1. Let `lead` be the result of evaluating `LeadSurrogate`.
2. Let `trail` be the result of evaluating `TrailSurrogate`.
3. Let `cp` be `UTF16Decode(lead, trail)`.
4. Return the character whose character value is `cp`.

The production `RegExpUnicodeEscapeSequence :: u Hex4Digits` evaluates as follows:
1. Return the character whose code is the CV of `Hex4Digits`.

The production `RegExpUnicodeEscapeSequence :: u \{ HexDigits \}` evaluates as follows:
1. Return the character whose code is the MV of `HexDigits`.

The production `LeadSurrogate :: Hex4Digits` evaluates as follows:
1. Return the character whose code is the CV of `Hex4Digits`.

The production `TrailSurrogate :: Hex4Digits` evaluates as follows:
1. Return the character whose code is the CV of `Hex4Digits`.

21.2.2.11 DecimalEscape

The production `DecimalEscape :: DecimalIntegerLiteral` 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. \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 :: S` evaluates by returning the set of all characters not included in the set returned by `CharacterClassEscape :: s`.

The production `CharacterClassEscape :: w` evaluates by returning the set of characters containing the sixty-three characters:
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 as follows:

1. Return the CharSet that is the result of evaluating `ClassAtom`.

The production `NonemptyClassRanges :: ClassAtom NonemptyClassRangesNoDash` evaluates as follows:

1. Evaluate `ClassAtom` to obtain a CharSet `A`.
2. Evaluate `NonemptyClassRangesNoDash` to obtain a CharSet `B`.
3. Return the union of CharSets `A` and `B`.

The production `NonemptyClassRanges :: ClassAtom - ClassAtom ClassRanges` evaluates as follows:

1. Evaluate the first `ClassAtom` to obtain a CharSet `A`.
2. Evaluate the second `ClassAtom` to obtain a CharSet `B`.
3. Evaluate `ClassRanges` to obtain a CharSet `C`.
4. Call `CharacterRange(A, B)` and let `D` be the resulting CharSet.
5. Return the union of CharSets `D` and `C`.

### 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:

1. If `A` does not contain exactly one character or `B` does not contain exactly one character then throw a `SyntaxError` exception.
2. Let `a` be the one character in CharSet `A`.
3. Let `b` be the one character in CharSet `B`.
4. Let `i` be the character value of character `a`.
5. Let `j` be the character value of character `b`.
6. If \( i > j \) then throw a `SyntaxError` exception.
7. Return the set containing all characters numbered \( i \) through \( j \), inclusive.

21.2.2.16 **NonemptyClassRangesNoDash**

The production `NonemptyClassRangesNoDash` :: `ClassAtom` evaluates as follows:

1. Return the CharSet that is the result of evaluating `ClassAtom`.

The production `NonemptyClassRangesNoDash` :: `ClassAtomNoDash` `NonemptyClassRangesNoDash` evaluates as follows:

1. Evaluate `ClassAtomNoDash` to obtain a CharSet \( A \).
2. Evaluate `NonemptyClassRangesNoDash` to obtain a CharSet \( B \).
3. Return the union of CharSets \( A \) and \( B \).

The production `NonemptyClassRangesNoDash` :: `ClassAtomNoDash` `-` `ClassAtom` `ClassRanges` evaluates as follows:

1. Evaluate `ClassAtomNoDash` to obtain a CharSet \( A \).
2. Evaluate `ClassAtom` to obtain a CharSet \( B \).
3. Evaluate `ClassRanges` to obtain a CharSet \( C \).
4. Call `CharacterRange` \( A, B \) and let \( D \) be the resulting CharSet.
5. Return the union of CharSets \( D \) and \( C \).

**NOTE 1**  
`ClassRanges` can expand into single `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]/i` matches all upper and lower-case letters in the Unicode Basic Latin block as well as the symbols `,`, `.`, `-`, `^` and `_`.

**NOTE 3**  
A `-` character can be treated literally or it can denote a range. It is treated literally if it is the first or last character of `ClassRanges`, the beginning or end limit of a range specification, or immediately follows a range specification.

21.2.2.17 **ClassAtom**

The production `ClassAtom` :: `-` evaluates by returning the CharSet containing the one character `-`.

The production `ClassAtom` :: `ClassAtomNoDash` evaluates by evaluating `ClassAtomNoDash` to obtain a CharSet and returning that CharSet.

21.2.2.18 **ClassAtomNoDash**

The production `ClassAtomNoDash` :: `SourceCharacter` but not one of `\` or `)` or `-` evaluates as follows:

1. Return the CharSet containing the character matched by `SourceCharacter`.

The production `ClassAtomNoDash` :: `\` `ClassEscape` evaluates as follows:

1. Return the CharSet that is the result of evaluating `ClassEscape`.  

© Ecma International 2014
21.2.2.19 ClassEscape

The production **ClassEscape :: DecimalEscape** evaluates as follows:
1. Evaluate **DecimalEscape** to obtain an **EscapeValue** E.
2. If E is not a character then throw a **SyntaxError** exception.
3. Let ch be E’s character.
4. Return the one-element CharSet containing the character ch.

The production **ClassEscape :: b** evaluates as follows:
1. Return the CharSet containing the single character `<BS>` U+0008 (BACKSPACE).

The production **ClassEscape :: CharacterEscape** evaluates as follows:
1. Return the CharSet that is the result of evaluating **CharacterEscape**.

The production **ClassEscape :: CharacterClassEscape** evaluates as follows:
1. Return the CharSet that is the result of evaluating **CharacterClassEscape**.

NOTE A **ClassAtom** can use any of the escape sequences that are allowed in the rest of the regular expression except for \b, \B, and backreferences. Inside a **CharacterClass**, \b means the backspace character, while \B and backreferences raise errors. Using a backreference inside a **ClassAtom** causes an error.

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 initializes a new RegExp object. Thus, the function call RegExp(_) is equivalent to the object creation expression new RegExp(_) with the same arguments. However, if the this value passed in the call is an Object with a [[RegExpMatcher]] internal slot whose value is undefined, it initializes the this value using the argument values. This permits RegExp to be used both as factory method and to perform constructor instance initialization.

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 super call to the RegExp constructor to initialize subclass instances.

21.2.3.1 RegExp(pattern, flags)

The following steps are taken:
1. Let func be the active 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 a [[RegExpMatcher]] internal slot or Type(O) is Object and O has a [[RegExpMatcher]] internal slot and the value of [[RegExpMatcher]] is not undefined, then
   a. If Type(pattern) is Object and O has a [[RegExpMatcher]] internal slot and flags is undefined, then
      i. Return pattern;
   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 [[RegExpMatcher]] internal slot, then
a. If the value of pattern’s [[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 [[OriginalSource]] internal slot.
d. Let F be the value of pattern’s [[OriginalFlags]] internal slot.
5. Else,
a. Let P be pattern.
b. Let F be flags.
6. Return the result of the abstract operation RegExpInitialize 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 recognized by RegExp, any backslash \ code points 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 `Construct(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 [[RegExpMatcher]] is initially assigned the value `undefined` as a flag to indicate that the instance has not yet been initialized 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: RegExpInitialize Abstract Operation

When the abstract operation RegExpInitialize 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 code unit other than "g", "i", "m", "u", or "y" or if it contains the same code unit more than once, then throw a SyntaxError exception.
8. If F contains "u" then let BMP be false, else let BMP be true.
9. If BMP is true, then
   a. Parse P using the grammars in 21.2.1 and interpreting each of its 16-bit elements as an Unicode BMP code point. UTF-16 decoding is not applied to the elements. The goal symbol for the parse is Pattern. Throw a SyntaxError exception if P did not conform to the grammar or if any elements of P were not matched by the parse.
   b. Let patternCharacters be a List whose elements are the code unit elements of P.
10. Else
    a. Parse P using the grammars in 21.2.1 and interpreting P as UTF-16 encoded Unicode code points. The goal symbol for the parse is Pattern. Throw a SyntaxError exception if P did not conform to the grammar or if any elements of P were not matched by the parse.
    b. Let patternCharacters be a List whose elements are the code points of resulting from applying UTF-16 decoding to P’s sequence of elements.
11. Set the value of obj’s [[OriginalSource]] internal slot to P.
12. Set the value of obj’s [[OriginalFlags]] internal slot to F.
13. Set obj’s [[RegExpMatcher]] internal slot to the internal procedure that evaluates the above parse of P by applying the semantics provided in 21.2.2 using patternCharacters as the pattern’s List of SourceCharacter values and F as the flag parameters.
14. Let putStatus be the result of Put(obj, "lastIndex", 0, true).
15. 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 RegExpInitialize 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:

1. Let S be a String in the form of a Pattern (Pattern% 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 (Pattern%) if F contains "u") must behave identically to the internal procedure given by the constructed object’s [[RegExpMatcher]] internal slot. Multiple calls to this abstract operation using the same values for P and F must produce identical results.
2. The code points / 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 can be parsed (in an appropriate lexical context) as a RegularExpressionLiteral that behaves identically to the constructed regular expression. For example, if P is "/", then S could be "\\/" or "\u002F", among other possibilities, but not "/", because \ followed by F would be parsed as a
450 © Ecma International 2014

SingleLineComment rather than a RegularExpressionLiteral. If $P$ is the empty String, this
specification can be met by letting $S$ be "(?::)".
3. 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:

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 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 `ToString(string)`.
6. ReturnIfAbrupt(S).

21.2.5.2.1 Runtime Semantics: `RegExpExec(R, S)` Abstract Operation

The abstract operation `RegExpExec` with arguments R and S performs the following steps:

1. Assert: `Type(R)` is Object.
2. Assert: `Type(S)` is String.
3. Let exec be `Get(R, "exec")`.
4. ReturnIfAbrupt(exec).
5. If `IsCallable(exec)` is `true`, then
   a. Let result be the result of calling the [[Call]] internal method of exec with arguments R, and (S).
   b. ReturnIfAbrupt(result).
   c. If `Type(result)` is neither Object or Null, then throw a `TypeError` exception.
   d. Return(result).
6. If R does not have a [[RegExpMatcher]] internal slot, then throw a `TypeError` exception.
7. If the value of R's [[RegExpMatcher]] internal slot is `undefined`, then throw a `TypeError` exception.

NOTE: If a callable exec property is not found this algorithm falls back to attempting to use the built-in RegExp matching algorithm. This provides compatible behaviour for code written for prior editions where most built-in algorithms that use regular expressions did not perform a dynamic property lookup of exec.

21.2.5.2.2 Runtime Semantics: `RegExpBuiltinExec(R, S)` Abstract Operation

The abstract operation `RegExpBuiltinExec` with arguments R and S performs the following steps:

1. Assert: R is an initialized RegExp instance.
2. Assert: `Type(S)` is String.
3. Let length be the number of code units in S.
4. Let lastIndex be `Get(R, "lastIndex")`.
5. Let i be `ToInteger(lastIndex)`.
6. ReturnIfAbrupt(i).
7. Let global be `ToBoolean(Get(R, "global"))`.
8. ReturnIfAbrupt(global).
9. Let sticky be `ToBoolean(Get(R, "sticky"))`.
10. ReturnIfAbrupt(sticky).
11. If `global` is `false` and `sticky` is `false`, then let i = 0.
12. Let matcher be the value of R's [[RegExpMatcher]] internal slot.
13. Let flags be the value of R's [[OriginalFlags]] internal slot.
14. If `flags` contains "u" then let `fullUnicode` be `true`, else let `fullUnicode` be `false`.
15. Let matchSucceeded be `false`.
16. Repeat, while `matchSucceeded` is `false`
   a. If i < 0 or i > length, then
      i. Let putStatus be `Put(R, "lastIndex", 0, true)`.
      ii. ReturnIfAbrupt(putStatus).
      iii. Return null.
   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 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.
17. Let \( e \) be \( r \)'s endIndex value.
18. If fullUnicode is true, then
   a. \( e \) is an index into the Input character list, derived from \( S \), matched by matcher. Let \( e \) UTF be the smallest index into \( S \) that corresponds to the character at element \( e \) of Input. If \( e \) is greater than the length of Input, then \( e \) UTF is \( 1 + \) the number of code units in \( S \).
   b. Let \( e \) be \( e \) UTF.
19. If global is true or sticky is true, then
   a. Let putStatus be the result of Put(\( R \), "lastIndex", \( e \), true).
   b. ReturnIfAbrupt(putStatus).
20. Let \( n \) be the length of \( r \)'s captures List. (This is the same value as 21.2.2.1's NcapturingParens.)
21. Let \( A \) be the result of the abstract operation ArrayCreate(\( n + 1 \)).
22. Assert: The value of \( A \)'s "length" property is \( n + 1 \).
23. Let matchIndex be \( i \).
24. Assert: The following CreateDataProperty calls will not result in an abrupt completion.
25. Perform CreateDataProperty(\( A \), "input", \( S \)).
26. Let matchedSubstr be the matched substring (i.e. the portion of \( S \) between offset \( i \) inclusive and offset \( e \) exclusive).
27. Perform CreateDataProperty(\( A \), "0", matchedSubstr).
28. For each integer \( i \) such that \( i > 0 \) and \( i \leq n \)
   a. Let captured be \( i \)th element of \( r \)'s captures List.
   b. If captured is undefined, then let capturedValue be undefined.
   c. Else if fullUnicode is true,
      i. Assert: captured is a List of code points.
      ii. Let capturedValue be a string whose code points are the UTF-16Encoding (10.1.1) of the code points of captured.
   d. Else, fullUnicode is false,
      i. Assert: captured is a List of code units.
      ii. Let capturedValue be a string whose elements are the code units of captured.
   e. Perform CreateDataProperty(\( A \), ToString(\( i \) , capturedValue)).
30. 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 code unit "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 code unit "i", 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. Let S be ToString(string).
4. ReturnIfAbrupt(S).
5. Let global be ToBoolean(Get(rx, "global")).
6. ReturnIfAbrupt(global).
7. If global is not true, then
   a. Return the result of RegExpExec(rx, S).
8. Else global is true.
   a. Let putStatus be Put(rx, "lastIndex", 0, true).
   b. ReturnIfAbrupt(putStatus).
   c. Let A be ArrayCreate(0).
   d. Let previousLastIndex be 0.
   e. Let n be 0.
   f. Repeat,
      i. Let result be RegExpExec(rx, S).
      ii. ReturnIfAbrupt(result).
      iii. If result is null, then
           1. If n = 0, then return null.
           2. Else, return A.
      iv. Else result is not null.
         1. Let thisIndex be ToInteger(Get(rx, "lastIndex")).
         2. ReturnIfAbrupt(thisIndex).
         3. If thisIndex = previousLastIndex then
            a. Let putStatus be Put(rx, "lastIndex", thisIndex, true).
            b. ReturnIfAbrupt(putStatus).
            c. Set previousLastIndex to thisIndex+1.
         4. Else,
            a. Set previousLastIndex to thisIndex.
         5. Let matchStr be Get(result, "0").
         6. Let defineStatus be CreateDataPropertyOrThrow(A, ToString(n), matchStr).
         7. ReturnIfAbrupt(defineStatus).
         8. Increment n.
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 code unit "m", then return true.
7. Return false.

21.2.5.7 RegExp.prototype.replace (string, replaceValue)

When the replace method is called with arguments string and replaceValue 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. Let S be ToString(string).
4. ReturnIfAbrupt(S).
5. Let lengthS be the number of code unit elements in S.
7. If functionalReplace is false, then
   a. Let replaceValue be ToString(replaceValue).
   b. ReturnIfAbrupt(replaceValue).
8. Let global be ToBoolean(Get(rx, "global"))
9. ReturnIfAbrupt(global).
10. If global is true, then
    a. Let putStatus be Put(rx, "lastIndex", 0, true).
    b. ReturnIfAbrupt(putStatus).
11. Let previousLastIndex be 0.
12. Let results be a new empty List.
13. Let done be false.
14. Repeat, while done is false
   a. Let result be RegExpExec(rx, S).
   b. ReturnIfAbrupt(result).
   c. If result is null, then set done to true.
   d. Else result is not null,
      i. If global is false, then set done to true.
      ii. Else,
         1. Let thisIndex be ToInteger(Get(rx, "lastIndex")).
         2. ReturnIfAbrupt(thisIndex).
         3. If thisIndex = previousLastIndex then
            a. Let putStatus be Put(rx, "lastIndex", thisIndex+1, true).
            b. ReturnIfAbrupt(putStatus).
            c. Set previousLastIndex to thisIndex+1.
         4. Else,
            a. Set previousLastIndex to thisIndex.
            b. If result is not null, then append result to the end of results.
15. Let accumulatedResult be the empty String value.
16. Let nextSourcePosition be 0.
17. Repeat, for each result in results,
a. Let nCaptures be ToLength(Get(result, "length")).
b. ReturnIfAbrupt(nCaptures).
c. Let nCaptures be max(nCaptures - 1, 0).
d. Let matched be ToString(Get(result, "0")).
e. ReturnIfAbrupt(matched).
f. Let matchLength be the number of code units in matched.
g. Let position be ToInteger(Get(result, "index")).
h. ReturnIfAbrupt(position).
i. Let position be max(min(position, lengthS), 0).

j. Let n be 1.
k. Let captures be an empty List.
l. Repeat while n ≤ nCaptures
   i. Let capN be Get(result, ToString(n)).
      ii. If Type(capN) is not Undefined, then let capN be ToString(capN).
      iii. ReturnIfAbrupt(capN).
      iv. Append capN as the last element of captures.
   v. Let n be n+1

m. If functionalReplace is true, then
   i. Let replerArgs be the List (matched).
   ii. Append in list order the elements of captures to the end of the List replerArgs.
   iii. Append position and S as the last two elements of replerArgs.
   iv. Let replValue be the result of calling the [[Call]] internal method of replaceValue passing undefined as the this value and replerArgs as the argument list.
   v. Let replacement be ToString(replValue).

n. Else,
   i. Let replacement be GetReplaceSubstitution(matched, S, position, captures, replaceValue).

o. ReturnIfAbrupt(replacement).
p. If position ≥ nextSourcePosition, then
   i. NOTE position should not normally move backwards. If it does, it is in indication of a ill-behaving RegExp subclass or use of an access triggered side-effect to change the global flag or other characteristics of rx. In such cases, the corresponding substitution is ignored.
   ii. Let accumulatedResult be the String formed by concatenating the code units of the current value of accumulatedResult with the substring of S consisting of the code units from nextSourcePosition inclusive up to position exclusive and with the code units of replacement.
   iii. Let nextSourcePosition be position + matchLength.

18. If nextSourcePosition ≥ lengthS, then return accumulatedResult.
19. Return the String formed by concatenating the code units of accumulatedResult with the substring of S consisting of the code units from nextSourcePosition inclusive up through the final code unit of S inclusive.

21.2.5.8 RegExp.prototype.search ( string )

When the search 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. Let S be ToString(string).
4. ReturnIfAbrupt(S).
5. Let previousLastIndex be Get(rx, "lastIndex").
6. ReturnIfAbrupt(previousLastIndex).
7. Let status be Put(rx, "lastIndex", 0, true)
8. ReturnIfAbrupt(status)
9. Let result be RegExpExec(rx, S).
10. ReturnIfAbrupt(result).
11. Let status be Put(rx, "lastIndex", previousLastIndex, true)
12. ReturnIfAbrupt(status)
13. If result is null, return -1.
14. Return Get(result, "index").

NOTE The lastIndex and global properties of this RegExp object are ignored when performing the search. The lastIndex property is left unchanged.

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.

21.2.5.10 RegExp.prototype.split (string, limit)

NOTE 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 String, nor does it 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 the regular expression matches 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.) 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, /a*/.split("ab") evaluates to the array ["a", "b"]; while /a*/.split("ab") evaluates to the array ["a", "b"].)

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\>bold\</b\>and\<code\>coded\</code\>\")

evaluates to the array


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. Let `matcher` be the value of `rx`'s `[[RegExpMatcher]]` internal slot.
5. If `matcher` is `undefined`, then throw a `TypeError` exception.
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 `T` be a String value equal to the substring of `S` consisting of the elements at positions `p` (inclusive) through `q` (exclusive).
      ii. Assert: The following call will never result in an abrupt completion.
      iii. Call CreateDataProperty(`A`, `ToString(lengthA)`), `T`.
      iv. If `lengthA = lim`, return `A`.
      v. Let `p = q`.
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 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 code unit "y", then return true.
7. Return false.

21.2.5.12  RegExp.prototype.test(S)

The following steps are taken:

1. Let R be the this value.
2. If Type(R) is not Object, then throw a TypeError exception.
3. Let string be ToString(S).
4. ReturnIfAbrupt(string).
5. Let match be RegExpExec(R, string).
6. ReturnIfAbrupt(match).
7. 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 Type(R) is not Object, then throw a TypeError exception.
3. Let pattern be ToString(Get(R, "source")).
4. ReturnIfAbrupt(pattern).
5. Let result be the String value formed by concatenating "/", pattern, and "/".
6. Let global be ToBoolean(Get(R, "global")).
7. ReturnIfAbrupt(global).
8. If global is true, then append "g" as the last code unit of result.
9. Let ignoreCase be ToBoolean(Get(R, "ignoreCase")).
10. ReturnIfAbrupt(ignoreCase).
11. If ignoreCase is true, then append "i" as the last code unit of result.
12. Let multiline be ToBoolean(Get(R, "multiline")).
13. ReturnIfAbrupt(multiline).
14. If multiline is true, then append "m" as the last code unit of result.
15. Let unicode be ToBoolean(Get(R, "unicode")).
16. ReturnIfAbrupt(unicode).
17. If unicode is true, then append "u" as the last code unit of result.
18. Let sticky be ToBoolean(Get(R, "sticky")).
19. ReturnIfAbrupt(sticky).
20. If sticky is true, then append "y" as the last code unit of result.
21. 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 `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 code unit "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 }.

**NOTE**
The `@@isRegExp` property is used by `String.prototype` methods to identify objects that have the basic behaviour of regular expressions. The absence of a `@@isRegExp` property or the existence of such a property whose value is not `Boolean coerce to true` indicates that the object should is not intended to be used as a regular expression object.

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 9.4.1.4 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 \( \text{Get}(O, \text{"length"}) \).
2. For each integer \( i \) in the range \( 0 \leq i < \text{ToUint32}(len) \)
   a. Let \( elem \) be the result of calling the \([[\text{GetOwnProperty}]] \) internal method of \( O \) with argument \( \text{ToString}(i) \).
   b. If \( elem \) is \text{undefined}, return \text{true}.
3. Return \text{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 initializes 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 [[ArrayInitializationState]] internal slot whose value is undefined, it initializes the this value using the argument values. This permits Array to be used both as factory method and to perform constructor instance initialization.

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 intend to inherit the specified Array behaviour must include a super call to the Array constructor to initialize subclass instances.

The length property of the Array constructor function is 1.

22.1.1.1 Array ()

This description applies if and only if the Array constructor is called with no arguments.

1. Let numberOfArgs be the number of arguments passed to this function call.
2. Assert: numberOfArgs = 0.
3. Let \( O \) be the this value.
4. If Type(O) is Object and \( O \) has an [[ArrayInitializationState]] internal slot and the value of [[ArrayInitializationState]] is false, then
   a. Set the value of \( O \)'s [[ArrayInitializationState]] internal slot to true.
   b. Let array be \( O \).
5. Else,
   a. Let \( F \) be the active function.
   b. Let proto be GetPrototypeFromConstructor(\( F \), "%ArrayPrototype%").
   c. ReturnIfAbrupt(proto).
   d. Let array be ArrayCreate(0, proto).
6. ReturnIfAbrupt(array).
7. Let putStatus be Put(array, "%length", 0, true).
8. ReturnIfAbrupt(putStatus).

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 function call.
3. Let \( O \) be the this value.
4. If Type(O) is Object and O has an [[ArrayInitializationState]] internal slot and the value of
   [[ArrayInitializationState]] is false, then
   a. Set the value of O’s [[ArrayInitializationState]] internal slot to true.
   b. Let array be O.
5. Else,
   a. Let F be the active function.
   b. Let proto be GetPrototypeFromConstructor(F, "%ArrayPrototype%”).
   c. ReturnIfAbrupt(proto).
   d. Let array be ArrayCreate(0, proto).
6. ReturnIfAbrupt(array).
7. If Type(len) is not Number, then
   a. Let defineStatus be 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.
10. ReturnIfAbrupt(putStatus).
11. Return array.

22.1.1.3 Array (...items)

This description applies if and only if the Array constructor is called with 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 function call.
3. Let O be the this value.
4. If Type(O) is Object and O has an [[ArrayInitializationState]] internal slot and the value of
   [[ArrayInitializationState]] is false, then
   a. Set the value of O’s [[ArrayInitializationState]] internal slot to true.
   b. Let array be O.
5. Else,
   a. Let F be the active function.
   b. Let proto be GetPrototypeFromConstructor(F, "%ArrayPrototype%”).
   c. ReturnIfAbrupt(proto).
   d. Let array be ArrayCreate(numberOfArgs, 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 kth element of items.
   c. Let defineStatus be CreateDataPropertyOrThrow(array, Pk, itemK).
   d. ReturnIfAbrupt(defineStatus).
   e. Increase k by 1.
10. Let putStatus be Put(array, "length", numberOfArgs, true).
11. ReturnIfAbrupt(putStatus).
12. Return array.
22.1.4  **new Array ( ... argumentsList)**

When `Array` is called as part of a `new` expression, it initializes 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 `Construct (F, argumentsList)`.

If `Array` is implemented as an ECMA-Script 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 [ , thisArg ] ] )**

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)`. ReturnIfAbrupt(`items`).
3. If `mapfn` is `undefined`, then let `mapping` be `false`.
4. 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`.
   c. Let `mapping` be `true`.
5. Let `usingIterator` be `CheckIterable(items)`.
6. ReturnIfAbrupt(`usingIterator`).
7. If `usingIterator` is not `undefined`, 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 `ArrayCreate(0)`.
      c. ReturnIfAbrupt(`A`).
      d. Let `iterator` be `GetIterator(items, usingIterator)`.
      e. ReturnIfAbrupt(`iterator`).
      f. Let `k` be `0`.
      g. Repeat
         i. Let `Pk` be `ToString(k)`.
         ii. Let `next` be `IteratorStep(iterator)`.
         iii. ReturnIfAbrupt(`next`).
         iv. If `next` is `false`, then
            1. Let `putStatus` be `Put(A, "length", k, true)`.
            2. ReturnIfAbrupt(`putStatus`).
            3. Return `A`.
         v. Let `nextValue` be `IteratorValue(next)`.

Commented [AWB798]: It would be nice to have a more explicit way to create a collection with a pre-specified number of elements.
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 (nextValue, k) as argumentsList.
2. ReturnIfAbrupt(mappedValue).
viii. Else, let mappedValue be nextValue.
ix. Let defineStatus be CreateDataPropertyOrThrow(A, Pk, mappedValue).

x. ReturnIfAbrupt(defineStatus).
xi. Increase k by 1.

9. Assert: items is not an Iterator so assume it is an array-like object.
10. Let lenValue be Get(items, “length”).
11. Let len be ToLength(lenValue).
12. ReturnIfAbrupt(len).
13. If IsConstructor(C) is true, then
   a. Let A be the result of calling the [[Construct]] internal method of C with an argument list
   containing the single item len.

14. Else,
   a. Let A be ArrayCreate(len).
15. ReturnIfAbrupt(A).
16. Let k be 0.
17. Repeat, while k < len
   a. Let Pk be ToString(k).
   b. Let kValue be Get(items, Pk).
   c. ReturnIfAbrupt(kValue).
   d. If mapping is true, then
      i. Let mappedValue be the result of calling the [[Call]] internal method of mapfn with T as
thisArgument and (kValue, k) as argumentsList.
   ii. ReturnIfAbrupt(mappedValue).
   e. Else, let mappedValue be kValue.
   f. Let defineStatus be CreateDataPropertyOrThrow(A, Pk, mappedValue).
   g. ReturnIfAbrupt(defineStatus).
   h. Increase k by 1.
19. ReturnIfAbrupt(putStatus).
20. Return A.

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 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 argument arg, and performs the following:
1. If Type(arg) is not Object, return false.
2. If arg is an exotic Array object, then return true.
3. Return 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 \( \text{len} \) be the actual number of arguments passed to this function.
2. Let \( \text{items} \) be the List of arguments passed to this function.
3. Let \( C \) be the \text{this} value.
4. If IsConstructor\((C)\) is true, then
   a. Let \( A \) be the result of calling the [[Construct]] internal method of \( C \) with an argument list containing the single item \( \text{len} \).
5. Else,
   a. Let \( A \) be ArrayCreate\((\text{len})\).
6. ReturnIfAbrupt\((A)\).
7. Let \( k \) be 0.
8. Repeat, while \( k < \text{len} \)
   a. Let \( k\text{Value} \) be element \( k \) of \( \text{items} \).
   b. Let \( \text{Pk} \) be ToString\((k)\).
   c. Let \( \text{defineStatus} \) be CreateDataPropertyOrThrow\((A, \text{Pk}, k\text{Value}[[\text{value}]])\).
   d. ReturnIfAbrupt\((\text{defineStatus})\).
   e. Increase \( k \) by 1.
9. Let \( \text{putStatus} \) be Put\((A, \text{"length"}, \text{len}, \text{true})\).
10. ReturnIfAbrupt\((\text{putStatus})\).
11. Return \( A \).

The \text{length} property of the \text{of} method is 0.

NOTE 1  The \text{items} argument is assumed to be a well-formed rest argument value.

NOTE 2  The \text{of} function is an intentionally generic factory method; it does not require that its \text{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 \( \text{Array.prototype} \) 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 \text{this} value.
2. Let \( \text{proto} \) be GetPrototypeFromConstructor\((F, \text{"%ArrayPrototype%"})\).
3. ReturnIfAbrupt\((\text{proto})\).
4. Let \( \text{obj} \) be ArrayCreate\((\text{undefined}, \text{proto})\).
5. Return \( \text{obj} \).

The value of the \text{name} property of this function is "\[Symbol.create\]."

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

NOTE 1  Passing \text{undefined} as the first argument to ArrayCreate causes the [[ArrayInitializationState]] internal slot of the array to be initially assigned the value \text{false}. This is a flag used to indicate that the instance has not yet been initialized 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 ( ...arguments )

When the `concat` method is called with zero or more arguments, 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 SameValue(`thisRealm`, GetFunctionRealm(`C`)) is true, 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 ArrayCreate(0).
6. ReturnIfAbrupt(`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 IsConcatSpreadable(`E`).
   c. ReturnIfAbrupt(`spreadable`).
   d. If `spreadable` is true, then
      i. Let `k` be 0.
      ii. Let `lenVal` be 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 HasProperty(`E`, `P`).
         3. ReturnIfAbrupt(`exists`).
         4. If `exists` is true, then

a. Let subElement be Get(E, P).
b. ReturnIfAbrupt(subElement).
c. Let status be 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 CreateDataPropertyOrThrow (A, ToString(n), E).
   ii. ReturnIfAbrupt(status).
   iii. Increase n by 1.
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.

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 ])

The copyWithin method takes up to three arguments target, start and end.

NOTE 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 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 relativeStart be ToInteger(start).
16. ReturnIfAbrupt(relativeStart).
17. If relativeStart is negative, let from be max((len + relativeStart), 0); else let from be min(relativeStart, len).
18. If end is undefined, let relativeEnd be len; else let relativeEnd be ToInteger(end).
19. ReturnIfAbrupt(relativeEnd).
20. If from < to and to < from + count
   a. Let direction = -1.
   b. Let from = from + count - 1.
   c. Let to = to + count - 1.
21. Else,
   a. Let direction = 1.
22. Repeat, while count > 0
   a. Let fromKey be ToString(from).
   b. Let toKey be ToString(to).
   c. Let fromPresent be HasProperty(O, fromKey).
   d. ReturnIfAbrupt(fromPresent).
   e. If fromPresent is true, then
      i. Let fromVal be Get(O, fromKey).
      ii. ReturnIfAbrupt(fromVal).
      iii. Let putStatus be Put(O, toKey, fromVal, true).
      iv. ReturnIfAbrupt(putStatus).
   f. Else fromPresent is false,
      i. Let deleteStatus be DeletePropertyOrThrow(O, toKey).
      ii. ReturnIfAbrupt(deleteStatus).
23. 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.

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 CreateArrayIterator(O, "key+value").

22.1.3.5 Array.prototype.every (callbackfn [ , thisArg ])

NOTE 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 \( \text{callbackfn} \) returned \( \text{true} \) for all elements, \( \text{every} \) will return \( \text{true} \). \( \text{callbackfn} \) is called only for elements of the array which actually exist; it is not called for missing elements of the array.

If a \( \text{thisArg} \) parameter is provided, it will be used as the \( \text{this} \) value for each invocation of \( \text{callbackfn} \). If it is not provided, \( \text{undefined} \) is used instead.

\( \text{callbackfn} \) is called with three arguments: the value of the element, the index of the element, and the object being traversed.

\( \text{every} \) does not directly mutate the object on which it is called but the object may be mutated by the calls to \( \text{callbackfn} \).

The range of elements processed by \( \text{every} \) is set before the first call to \( \text{callbackfn} \). Elements which are appended to the array after the call to \( \text{every} \) begins will not be visited by \( \text{callbackfn} \). If existing elements of the array are changed, their value as passed to \( \text{callbackfn} \) will be the value at the time \( \text{every} \) visits them; elements that are deleted after the call to \( \text{every} \) begins and before being visited are not visited. \( \text{every} \) acts like the “for all” quantifier in mathematics. In particular, for an empty array, it returns \( \text{true} \).

When the \( \text{every} \) method is called with one or two arguments, the following steps are taken:

1. Let \( O \) be the result of calling \( \text{ToObject} \) passing the \( \text{this} \) value as the argument.
2. ReturnIfAbrupt\( (O) \).
3. Let \( \text{lenValue} \) be \( \text{Get}(O, \text{"length"}) \).
4. Let \( \text{len} \) be \( \text{ToLength}(\text{lenValue}) \).
5. ReturnIfAbrupt\( (\text{len}) \).
6. If \( \text{IsCallable}(\text{callbackfn}) \) is \( \text{false} \), throw a \( \text{TypeError} \) exception.
7. If \( \text{thisArg} \) was supplied, let \( T \) be \( \text{thisArg} \); else let \( T \) be \( \text{undefined} \).
8. Let \( k \) be \( 0 \).
9. Repeat, while \( k < \text{len} \):
   a. Let \( P_k \) be \( \text{ToString}(k) \).
   b. Let \( k\text{Present} \) be \( \text{HasProperty}(O, P_k) \).
   c. ReturnIfAbrupt\( (k\text{Present}) \).
   d. If \( k\text{Present} \) is \( \text{true} \), then
      i. Let \( k\text{Value} \) be \( \text{Get}(O, P_k) \).
      ii. ReturnIfAbrupt\( (k\text{Value}) \).
      iii. Let \( \text{testResult} \) be the result of calling the \([\text{\[Call\]}]\) internal method of \( \text{callbackfn} \) with \( T \) as \( \text{thisArgument} \) and a List containing \( k\text{Value} \), \( k \), and \( O \) as \( \text{argumentsList} \).
      iv. ReturnIfAbrupt\( (\text{testResult}) \).
   e. If \( \text{ToBoolean}(\text{testResult}) \) is \( \text{false} \), return \( \text{false} \).
   f. Increase \( k \) by \( 1 \).
10. Return \( \text{true} \).

The \( \text{length} \) property of the \( \text{every} \) method is \( 1 \).

NOTE The \( \text{every} \) 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.

22.1.3.6 Array.prototype.fill (value [, start [, end ]])

The \( \text{fill} \) method takes up to three arguments \( \text{value} \), \( \text{start} \) and \( \text{end} \).

NOTE The \( \text{start} \) and \( \text{end} \) arguments are optional with default values of \( 0 \) and the length of the \( \text{this} \) object. If \( \text{start} \) is negative, it is treated as \( \text{length}+\text{start} \) where \( \text{length} \) is the length of the array. If \( \text{end} \) is negative, it is treated as \( \text{length}+\text{end} \).

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 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 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.

22.1.3.7 Array.prototype.filter (callbackfn [, thisArg])

NOTE callbackfn should be a function that accepts three arguments and returns a value that is coercible to the Boolean value true or false. filter calls callbackfn once for each element in the array, in ascending order, and constructs a new array of all the values for which callbackfn returns true. callbackfn is called only for elements of the array which actually exist; it is not called for missing elements of the array.

If a thisArg parameter is provided, it will be used as the this value for each invocation of callbackfn. If it is not provided, undefined is used instead.

callbackfn is called with three arguments: the value of the element, the index of the element, and the object being traversed.

filter does not directly mutate the object on which it is called but the object may be mutated by the calls to callbackfn.

The range of elements processed by filter is set before the first call to callbackfn. Elements which are appended to the array after the call to filter begins will not be visited by callbackfn. If existing elements of the array are changed their value as passed to callbackfn will be the value at the time filter visits them; elements that are deleted after the call to filter begins and before being visited are not visited.

When the filter method is called with one or two arguments, the following steps are taken:

1. Let O be the result of calling ToObject passing the this value as the argument.
2. ReturnIfAbrupt(O).
3. Let lenValue be 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 SameValue(thisRealm, GetFunctionRealm(C)) is true, then
         1. Let A be the result of calling the [[Construct]] internal method of C with an argument list containing the single item O.
   10. If A is undefined, then
       a. Let A be ArrayCreate(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 HasProperty(O, Pk).
           c. ReturnIfAbrupt(kPresent).
           d. If kPresent is true, then
              i. Let kValue be Get(O, Pk).
              ii. ReturnIfAbrupt(kValue).
              iii. Let selected be the result of calling the [[Call]] internal method of callbackfn with T as thisArgument and a List containing kValue, k, and O as argumentsList.
              iv. ReturnIfAbrupt(selected).
              v. If ToBoolean(selected) is true, then
                 1. Let status be CreateDataPropertyOrThrow (A, ToString(to), kValue).
                 2. ReturnIfAbrupt(status).
                 3. Increase to by 1.
           e. 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.

22.1.3.8 Array.prototype.find ( predicate [, thisArg ] )

NOTE 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.

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 `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 `kValue` be `Get(O, Pk)`.
   c. ReturnIfAbrupt(`kValue`).
   d. 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.
   e. ReturnIfAbrupt(`testResult`).
   f. If `ToBoolean(testResult)` is `true`, return `kValue`.
   g. 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.

22.1.3.9 `Array.prototype.findIndex` ( `predicate [, thisArg ]` )

NOTE `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`.

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 `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 kValue be Get(O, Pk).
   c. ReturnIfAbrupt(kValue).
   d. 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 arguments.
   e. ReturnIfAbrupt(testResult).
   f. If ToBoolean(testResult) is true, return k.
   g. 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.

22.1.3.10 Array.prototype.forEach (callbackfn [, thisArg ])

NOTE  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 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 HasProperty(O, Pk).
   c. ReturnIfAbrupt(kPresent).
   d. If kPresent is true, then
      i. Let kValue be Get(O, Pk).
      ii. ReturnIfAbrupt(kValue).
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.

22.1.3.11 Array.prototype.indexOf (searchElement, fromIndex)

NOTE `indexOf` compares `searchElement` to the elements of the array, in ascending order, using the Strict Equality Comparison algorithm (7.2.11), 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 `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`.
   b. If `k < 0`, then let `k` be 0.
12. Repeat, while `k < len`
   a. Let `kPresent` be `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.
22.1.3.12 Array.prototype.join (separator)

NOTE 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-element 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 Get(O, ToString(k)).
   c. If element is undefined or null, then let next be the empty String; otherwise, let next be ToString(element).
   d. ReturnIfAbrupt(next).
   e. Let R be a String value produced by concatenating S and next.
   f. Increase k by 1.
14. Return R.

The length property of the join method is 1.

NOTE The join function is intentionally generic; it does not require that its this value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method.

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 CreateArrayIterator(O, "key").

22.1.3.14 Array.prototype.lastIndexOf (searchElement [, fromIndex])

NOTE lastIndexOf compares searchElement to the elements of the array in descending order using the Strict Equality Comparison algorithm (7.2.11), 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 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 HasProperty(O, ToString(k)).
   b. ReturnIfAbrupt(kPresent).
   c. If kPresent is true, then
      i. Let elementK be 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.

22.1.3.15 Array.prototype.map (callbackfn [, thisArg ])

NOTE 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 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 SameValue(thisRealm, GetFunctionRealm(C)) is true, 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
   a. Let A be ArrayCreate(len).
11. ReturnIfAbrupt(A).
12. Let k be 0.
13. Repeat, while k < len
   a. Let Pk be ToString(k).
   b. Let kPresent be HasProperty(O, Pk).
   c. ReturnIfAbrupt(kPresent).
   d. If kPresent is true, then
      i. Let kValue be Get(O, Pk).
      ii. ReturnIfAbrupt(kValue).
      iii. Let mappedValue be the result of calling the [[Call]] internal method of callbackfn with T as thisArgument and a List containing kValue, k, and O as argumentsList.
      iv. ReturnIfAbrupt(mappedValue).
      v. Let status be CreateDataPropertyOrThrow(A, Pk, mappedValue).
      vi. ReturnIfAbrupt(status).
   e. Increase k by 1.
14. 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.

22.1.3.16 Array.prototype.pop ()

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

When the pop method is called the following steps are taken:
1. Let O be the result of calling ToObject passing the this value as the argument.
2. ReturnIfAbrupt(O).
3. Let lenVal be Get(O, "length").
4. Let len be ToLength(lenVal).
5. ReturnIfAbrupt(len).
6. If len is zero,
   a. Let putStatus be 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 Get(O, indx).
d. ReturnIfAbrupt(element).
e. Let deleteStatus be DeletePropertyOrThrow(O, indx).
f. ReturnIfAbrupt(deleteStatus).
g. Let putStatus be 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.

22.1.3.17 Array.prototype.push (...items)

NOTE 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 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 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 Put(O, ToString(n), E, true).
   c. ReturnIfAbrupt(putStatus).
   d. Increase n by 1.
8. Let putStatus be 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.

22.1.3.18 Array.prototype.reduce ( callbackfn [, initialValue ])

NOTE 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 `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`.
   b. If `kPresent` is `false`, throw a `TypeError` exception.
    c. If `kPresent` is `false`, throw a `TypeError` exception.
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 `HasProperty(O, Pk)`.
          iii. ReturnIfAbrupt(`kPresent`).
              iv. If `kPresent` is `true`, then
                  1. Let `accumulator` be `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 `HasProperty(O, Pk)`.
    c. ReturnIfAbrupt(`kPresent`).
    d. If `kPresent` is `true`, then
       i. Let `kValue` be `Get(O, Pk)`.
       ii. ReturnIfAbrupt(`kValue`).
       iii. Let `accumulator` be the result of calling the `[[Call]]` internal method of `callbackfn` with `undefined` as this Argument 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.

22.1.3.19 Array.prototype.reduceRight (callbackfn[, initialValue])

NOTE `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 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 HasProperty(O, Pk).
      iii. ReturnIfAbrupt(kPresent).
      iv. If kPresent is true, then
          i. Let accumulator be Get(O, Pk).
          ii. ReturnIfAbrupt(accumulator).
          iii. 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 HasProperty(O, Pk).
   c. ReturnIfAbrupt(kPresent).
   d. If kPresent is true, then
      i. Let kValue be Get(O, Pk).
      ii. ReturnIfAbrupt(kValue).
      iii. Let accumulator be the result of calling the [[Call]] internal method of callbackfn with undefined as thisArgument and a List containing accumulator, kValue, k, and O as argumentsList.
      iv. ReturnIfAbrupt(accumulator).
   e. Decrease k by 1.
12. Return accumulator.

The length property of the reduceRight method is 1.
NOTE  The `reduceRight` function is intentionally generic; it does not require that its this value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

22.1.3.20  Array.prototype.reverse ( )

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

When the reverse method is called the following steps are taken:

1. Let O be the result of calling ToObject passing the this value as the argument.
2. ReturnIfAbrupt(O).
3. Let lenVal be Get(O, "length").
4. Let len be ToLength(lenVal).
5. ReturnIfAbrupt(len).
6. Let middle be floor(len/2).
7. Let lower be 0.
8. Repeat, while lower ≠ middle
   a. Let upper be len– lower – 1.
   b. Let upperP be ToString(upper).
   c. Let lowerP be ToString(lower).
   d. Let lowerExists be HasProperty(O, lowerP).
   e. ReturnIfAbrupt(lowerExists).
   f. If lowerExists is true, then
      i. Let lowerValue be Get(O, lowerP).
      ii. ReturnIfAbrupt(lowerValue).
   g. Let upperExists be HasProperty(O, upperP).
   h. ReturnIfAbrupt(upperExists).
   i. If upperExists is true, then
      i. Let upperValue be Get(O, upperP).
      ii. ReturnIfAbrupt(upperValue).
   j. If lowerExists is true and upperExists is true, then
      i. Let putStatus be Put(O, lowerP, upperValue, true).
      ii. ReturnIfAbrupt(putStatus).
      iii. Let putStatus be Put(O, upperP, lowerValue, true).
      iv. ReturnIfAbrupt(putStatus).
   k. Else if lowerExists is false and upperExists is true, then
      i. Let putStatus be Put(O, lowerP, upperValue, true).
      ii. ReturnIfAbrupt(putStatus).
      iii. Let deleteStatus be DeletePropertyOrThrow (O, upperP).
      iv. ReturnIfAbrupt(deleteStatus).
   l. Else if lowerExists is true and upperExists is false, then
      i. Let deleteStatus be DeletePropertyOrThrow (O, lowerP).
      ii. ReturnIfAbrupt(deleteStatus).
      iii. Let putStatus be Put(O, upperP, lowerValue, true).
      iv. ReturnIfAbrupt(putStatus).
   m. Else both lowerExists and upperExists are false,
      i. No action is required.
   n. Increase lower by 1.
9. Return O.

NOTE  The reverse function is intentionally generic; it does not require that its this value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method.
22.1.3.21 Array.prototype.shift()

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

When the shift method is called the following steps are taken:

1. Let O be the result of calling ToObject passing the this value as the argument.
2. ReturnIfAbrupt(O).
3. Let lenVal be Get(O, "length").
4. Let len be ToLength(lenVal).
5. ReturnIfAbrupt(len).
6. If len is zero, then
   a. Let putStatus be Put(O, "length", 0, true).
   b. ReturnIfAbrupt(putStatus).
   c. Return undefined.
7. Let first be 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 HasProperty(O, from).
    d. ReturnIfAbrupt(fromPresent).
    e. If fromPresent is true, then
       i. Let fromVal be Get(O, from).
       ii. ReturnIfAbrupt(putVal).
       iii. Let putStatus be Put(O, to, fromVal, true).
       iv. ReturnIfAbrupt(putStatus).
    f. Else fromPresent is false,
       i. Let deleteStatus be DeletePropertyOrThrow(O, to).
       ii. ReturnIfAbrupt(deleteStatus).
    g. Increase k by 1.
11. Let deleteStatus be DeletePropertyOrThrow(O, ToString(len–1)).
12. ReturnIfAbrupt(deleteStatus).
13. Let putStatus be Put(O, "length", len–1, true).
14. ReturnIfAbrupt(putStatus).
15. Return first.

NOTE The shift function is intentionally generic; it does not require that its this value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

22.1.3.22 Array.prototype.slice(start, end)

NOTE 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 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 `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 `SameValue(thisRealm, GetFunctionRealm(C))` is `true`, then  
         i. 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 `ArrayCreate(count)`.  
   b. ReturnIfAbrupt(A).  
16. Let `n` be 0.  
17. Repeat, while `k < final`  
   a. Let `Pk` be `ToString(k)`.  
   b. Let `kPresent` be `HasProperty(O, Pk)`.  
   c. ReturnIfAbrupt(kPresent).  
   d. If `kPresent` is `true`, then  
      i. Let `kValue` be `Get(O, Pk)`.  
      ii. ReturnIfAbrupt(kValue).  
      iii. Let `status` be `CreateDataPropertyOrThrow(A, ToString(n), kValue)`.  
      iv. ReturnIfAbrupt(status).  
   e. Increase `k` by 1.  
   f. Increase `n` by 1.  
19. Let `putStatus` be `Put(A, "length", n, true)`.  
20. ReturnIfAbrupt(putStatus).  

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.

22.1.3.23 `Array.prototype.some(callbackfn[, thisArg])`

NOTE `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.

`s某些` 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 `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 `HasProperty(O, Pk)`.
   c. ReturnIfAbrupt(`kPresent`).
   d. If `kPresent` is `true`, then:
      i. Let `kValue` be `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.

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 `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 `Get(obj, "length")`.
3. Let `len` be `ToLength(lenValue)`.
4. ReturnIfAbrupt(len).

The sort order is the ordering of the array index property values of obj after completion of this function. 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 sort order is implementation-defined. The sort order is also implementation-defined if comparefn is undefined and SortCompare (22.1.3.24.1) does not act as a consistent comparison function.

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 sort order is implementation-defined:

- \( \text{obj is sparse (22.1)} \)
- \( 0 \leq j < \text{len} \)
- The result of HasProperty(proto, ToString(j)) is true.

The sort order is also implementation defined if \( \text{obj is sparse} \) and any of the following conditions are true:

- The result of the predicate IsExtensible(obj) is false.
- Any array index property of \( \text{obj} \) whose name is a nonnegative integer less than \( \text{len} \) is a data property whose [[Configurable]] attribute is false.

The sort order is also implementation defined if any of the following conditions are true:

- If \( \text{obj is an exotic object (including Proxy exotic objects) whose behaviour for [[Get]], [[Set]], [[Delete]], and [[HasOwnProperty]] is different from the ordinary object behaviour for these internal methods.} \)
- If any array index property of \( \text{obj} \) whose name is a nonnegative integer less than \( \text{len} \) is an accessor property or is a data property whose [[Writable]] attribute is false.

The following steps are taken:

1. Perform an implementation-dependent sequence of calls to the [[Get]], [[Set]], and [[HasOwnProperty]] internal methods of comparefn, to the DeletePropertyOrThrow abstract operation with \( \text{obj as the first argument, and to SortCompare (described below), such that:} \)
   - The property key argument for each call to [[Get]], [[Set]], [[HasOwnProperty]], or DeletePropertyOrThrow is the string representation of a nonnegative integer less than \( \text{len} \).
   - The arguments for calls to SortCompare are either values returned by a previous calls to the [[Get]] internal method or undefined if [[HasOwnProperty]] was used to determine that the property of \( \text{obj} \) that would otherwise be accessed using [[Get]] does not exist.
   - If \( \text{obj is not sparse} \) 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.

Unless the sort order is specified above to be implementation-defined, the returned object must have the following two characteristics:

- 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 \( \text{old}[j] \) existed, then \( \text{new}[\pi(j)] \) is exactly the same value as \( \text{old}[j] \). But if property \( \text{old}[j] \) did not exist, then \( \text{new}[\pi(j)] \) does not exist.
• Then for all nonnegative integers \( j \) and \( k \), each less than \( len \), if \( \operatorname{SortCompare}(\text{old}[j], \text{old}[k]) < 0 \) (see \( \operatorname{SortCompare} \) below), then \( \text{new}[\text{old}[j]] < \text{new}[\text{old}[k]] \).

Here the notation \( \text{old}[j] \) is used to refer to the hypothetical result of calling the \([\text{Get}]\) internal method of \( \text{obj} \) with argument \( j \) before this function is executed, and the notation \( \text{new}[j] \) to refer to the hypothetical result of calling the \([\text{Get}]\) internal method of \( \text{obj} \) with argument \( j \) after this function has been executed.

A function \( \text{comparefn} \) is a consistent comparison function for a set of values \( S \) if all of the requirements below are met for all values \( a \), \( b \), and \( c \) (possibly the same value) in the set \( S \). The notation \( a \prec_S b \) means \( \text{comparefn}(a, b) < 0 \); \( a \prec_S b \) means \( \text{comparefn}(a, b) = 0 \) (of either sign); and \( a \succ_S b \) means \( \text{comparefn}(a, b) > 0 \).

• Calling \( \text{comparefn}(a, b) \) always returns the same value \( v \) when given a specific pair of values \( a \) and \( b \) as its two arguments. Furthermore, \( \text{Type}(v) \) is \text{Number}, and \( v \) is not \text{NaN}. Note that this implies that exactly one of \( a \prec_S b \), \( a \equiv_S b \), and \( a \succ_S b \) will be true for a given pair of \( a \) and \( b \).

• Calling \( \text{comparefn}(a, b) \) does not modify \( \text{obj} \) or any object on \( \text{obj} \)’s prototype chain.

• \( a \equiv_S a \) (reflexivity)
• If \( a \equiv_S b \), then \( b \equiv_S a \) (symmetry)
• If \( a \prec_S b \) and \( b \equiv_S c \), then \( a \prec_S c \) (transitivity of \( \prec_S \))
• If \( a \prec_S b \) and \( b \succ_S c \), then \( a \prec_S c \) (transitivity of \( \prec_S \))
• If \( a \succ_S b \) and \( b \succ_S c \), then \( a \succ_S c \) (transitivity of \( \succ_S \))

### NOTE 1
The above conditions are necessary and sufficient to ensure that \( \text{comparefn} \) divides the set \( S \) into equivalence classes and that these equivalence classes are totally ordered.

### NOTE 2
The \text{sort} 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.

#### 22.1.3.24.1 Runtime Semantics: \text{SortCompare} Abstract Operation

When the \text{SortCompare} abstract operation is called with two arguments \( x \) and \( y \), the following steps are taken:

1. If \( x \) and \( y \) are both \text{undefined}, return +0.
2. If \( x \) is \text{undefined}, return 1.
3. If \( y \) is \text{undefined}, return −1.
4. If the argument \text{comparefn} is not \text{undefined}, then
   a. If \text{IsCallable}(\text{comparefn}) is \text{false}, throw a \text{TypeError} exception.
   b. Let \( v \) be the result of calling the \([\text{Call}]\) internal method of \text{comparefn} passing \text{undefined} as \text{thisArgument} and with a \text{List} containing the values of \( x \) and \( y \) as the \text{argumentsList}.
   c. \text{ReturnIfAbrupt}(v).
   d. If \( v \) is \text{NaN}, then return +0.
   e. Return \( v \).
5. Let \text{xString} be ToString(\( x \)).
6. ReturnIfAbrupt(\text{xString}).
7. Let \text{yString} be ToString(\( y \)).
8. ReturnIfAbrupt(\text{yString}).
9. If \text{xString} < \text{yString}, return −1.
10. If \text{xString} > \text{yString}, return 1.
11. Return +0.

### NOTE 1
Because non-existent property values always compare greater than \text{undefined} property values, and \text{undefined} always compares greater than any other value, \text{undefined} property values always sort to the end of the result, followed by non-existent property values.
NOTE 2 Method calls performed by the ToString abstract operations in steps 5 and 7 have the potential to cause SortCompare to not behave as a consistent comparison function.

22.1.3.25 Array.prototype.splice (start, deleteCount, ...items)

NOTE When the splice method is called with two or more arguments start, deleteCount and zero or more items, the deleteCount elements of the array starting at integer index start are replaced by the arguments items. 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 the number of actual arguments is 0, then
   a. Let actualDeleteCount be 0.
10. Else if the number of actual arguments is 1, 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 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 SameValue(thisRealm, GetFunctionRealm(C)) is true, then
           i. 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 ArrayCreate(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 HasProperty(O, from).
    c. ReturnIfAbrupt(fromPresent).
    d. If fromPresent is true, then
       i. Let fromValue be Get(O, from).
       ii. ReturnIfAbrupt(fromValue).
       iii. Let status be CreateDataPropertyOrThrow(A, ToString(k), fromValue).
       iv. ReturnIfAbrupt(status).
    e. Increment k by 1.
19. ReturnIfAbrupt(putStatus).

Commented [AWB7105]: It would be nice to have a more explicit way to create a collection with a pre-specified number of elements.
20. Let items be a List whose elements are, in left to right order, the portion of the actual argument list starting with the third argument. The list will be empty fewer than three arguments were passed.

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 HasProperty(O, from).
      iv. ReturnIfAbrupt(fromPresent).
      v. If fromPresent is true, then
         1. Let fromValue be Get(O, from).
         2. ReturnIfAbrupt(fromValue).
         3. Let putStatus be Put(O, to, fromValue, true).
         4. ReturnIfAbrupt(putStatus).
     vi. Else fromPresent is false,
         1. Let deleteStatus be DeletePropertyOrThrow(O, to).
         2. ReturnIfAbrupt(deleteStatus).
     vii. Increase k by 1.
   c. Let k be len.
   d. Repeat, while k > (len - actualDeleteCount + itemCount)
      i. Let deleteStatus be DeletePropertyOrThrow(O, ToString(k - 1)).
      ii. ReturnIfAbrupt(deleteStatus).
      iii. Decrease k by 1.
23. Else if itemCount > actualDeleteCount, then
   a. Let k be (len – actualDeleteCount).
   b. Repeat, while k > actualStart
      i. Let from be ToString(k + actualDeleteCount - 1).
      ii. Let to be ToString(k + itemCount - 1).
      iii. Let fromPresent be HasProperty(O, from).
      iv. ReturnIfAbrupt(fromPresent).
      v. If fromPresent is true, then
         1. Let fromValue be Get(O, from).
         2. ReturnIfAbrupt(fromValue).
         3. Let putStatus be Put(O, to, fromValue, true).
         4. ReturnIfAbrupt(putStatus).
      vi. Else fromPresent is false,
         1. Let deleteStatus be DeletePropertyOrThrow(O, to).
         2. ReturnIfAbrupt(deleteStatus).
     vii. Decrease k by 1.
24. Let k be actualStart.
25. 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 Put(O, ToString(k), E, true).
   c. ReturnIfAbrupt(putStatus).
   d. Increase k by 1.
27. ReturnIfAbrupt(putStatus).
28. Return A.

The length property of the splice method is 2.
NOTE 1. The explicit setting of the length property of the result Array in step 18 is necessary to ensure that its value is correct in situations where its trailing elements are not present.

NOTE 2. The splice function is intentionally generic; it does not require that its this value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

22.1.3.26 Array.prototype.toLocaleString ( [ reserved1 [, reserved2 ] ] )

An ECMAScript implementation that includes the ECMA-402 Internationalization API must implement the Array.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.

The meanings 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 positions for anything else.

The following steps are taken:

1. Let array be the result of calling ToObject passing the this value as the argument.
2. ReturnIfAbrupt(array).
3. Let arrayLen be Get(array, "length").
4. Let len be ToLength(arrayLen).
5. ReturnIfAbrupt(len).
6. Let separator be the String value for the list-separator String appropriate for the host environment’s current locale (this is derived in an implementation-defined way).
7. If len is zero, return the empty String.
8. Let firstElement be Get(array, "0").
9. ReturnIfAbrupt(firstElement).
10. If firstElement is undefined or null, then
a. Let R be the empty String.
11. Else
a. Let R be Invoke(firstElement, "toLocaleString").
b. Let R be ToString(R).
c. ReturnIfAbrupt(R).
12. Let k be 1.
13. Repeat, while k < len
a. Let S be a String value produced by concatenating R and separator.
b. Let nextElement be Get(array, ToString(k)).
c. ReturnIfAbrupt(nextElement).
d. If nextElement is undefined or null, then
   i. Let R be the empty String.
   Else
      i. Let R be 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 elements of the array are converted to Strings using their toLocaleString methods, and these Strings are then concatenated, separated by occurrences of a separator String that has been derived in an implementation-defined locale-specific way. The result of calling this function is intended to be analogous to the result of toString, except that the result of this function is intended to be locale-specific.

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.

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 on the this value.
2. ReturnIfAbrupt(array).
3. Let func be Get(array, "join").
4. ReturnIfAbrupt(func).
5. If IsCallable(func) is false, then let func be the intrinsic function %ObjProto_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.

22.1.3.28 Array.prototype.unshift ( ...items )

NOTE 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 Get(O, "length").
4. Let len be ToLength(lenVal).
5. ReturnIfAbrupt(len).
6. Let argCount be the number of actual arguments.
7. If argCount > 0, then
   a. Let k be len.
   b. Repeat, while k > 0,
      i. Let from be ToString(k–1).
      ii. Let to be ToString(k+argCount –1).
      iii. Let fromPresent be 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 Put(O, to, fromValue, true).
         4. ReturnIfAbrupt(putStatus).
      vi. Else fromPresent is false,
         1. Let deleteStatus be DeletePropertyOrThrow(O, to).
         2. ReturnIfAbrupt(deleteStatus).
      vii. Decrease k by 1.
   c. Let j be 0.
d. Let items be a List whose elements are, in left to right order, the arguments that were passed to
this function invocation.
e. Repeat, while items is not empty
i. Remove the first element from items and let E be the value of that element.
ii. Let putStatus be Put(O, ToString(j), E, true).
iii. ReturnIfAbrupt(putStatus).
iv. Increase j by 1.
8. Let putStatus be Put(O, "length", len+argCount, true).
9. ReturnIfAbrupt(putStatus).
10. Return len+argCount.

The length property of the unshift method is 1.

NOTE The unshift function is intentionally generic; it does not require that its this value be an Array object.
Therefore it can be transferred to other kinds of objects for use as a method.

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 CreateArrayIterator(O, "value").

This function is the %ArrayProto_values% intrinsic object.

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 ObjectCreate(%ObjectPrototype%).
2. Perform CreateDataProperty(blackList, "copyWithin", true).
3. Perform CreateDataProperty(blackList, "entries", true).
4. Perform CreateDataProperty(blackList, "fill", true).
5. Perform CreateDataProperty(blackList, "find", true).
6. Perform CreateDataProperty(blackList, "findIndex", true).
7. Perform CreateDataProperty(blackList, "keys", true).
8. Perform CreateDataProperty(blackList, "values", true).
9. Assert: Each of the above calls will return true.
10. Return blackList.

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

NOTE The own property names of this object 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 [[ArrayInitializationState]] 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. Assert: Type(array) is Object.
2. Let iterator be ObjectCreate(%ArrayIteratorPrototype%, ([IteratedObject], [[ArrayIteratorNextIndex]], [[ArrayIterationKind]])).
3. Set iterator’s [[IteratedObject]] internal slot to array.
4. Set iterator’s [[ArrayIteratorNextIndex]] internal slot to 0.
5. Set iterator’s [[ArrayIterationKind]] internal slot to kind.
6. 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 %IteratorPrototype% intrinsic object (25.1.2). 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 \([\text{IteratedObject}]\) internal slot of \( O \).
5. If \( a \) is \textbf{undefined}, then return \text{CreateIterResultObject}(\text{undefined}, \text{true}).
6. Let \( index \) be the value of the \([\text{ArrayIteratorNextIndex}]\) internal slot of \( O \).
7. Let \( itemKind \) be the value of the \([\text{ArrayIterationKind}]\) internal slot of \( O \).
8. Let \( lenValue \) be \text{Get}(\( a \), \text{"length"}).
9. Let \( len \) be \text{ToLength}(\( lenValue \)).
10. ReturnIfAbrupt(\( len \)).
11. If \( index \geq len \), then
   a. Set the value of the \([\text{IteratedObject}]\) internal slot of \( O \) to \text{undefined}.
   b. Return \text{CreateIterResultObject}(\text{undefined}, \text{true}).
12. Set the value of the \([\text{ArrayIteratorNextIndex}]\) internal slot of \( O \) to \( index+1 \).
13. If \( itemKind \) is \"key\", then let \( result \) be \( index \).
14. Else,
   a. Let \( elementKey \) be \text{ToString}(\( index \)).
   b. Let \( elementValue \) be \text{Get}(\( a \), \( elementKey \)).
   c. ReturnIfAbrupt(\( elementValue \)).
15. If \( itemKind \) is \"value\", then let \( result \) be \( elementValue \).
16. Else,
   a. Assert \( itemKind \) is \"key+value\".
   b. Let \( result \) be \text{ArrayCreate}(2).
   c. Assert: \( result \) is a new, well-formed Array object so the following operations will never fail.
   d. \text{Call CreateDataProperty}(\( result \), \"0\", \( index \)).
   e. \text{Call CreateDataProperty}(\( result \), \"1\", \( elementValue \)).
17. Return \text{CreateIterResultObject}(\( result \), \text{false}).

22.1.5.2.2 \%ArrayIteratorPrototype% [@@iterator\]()

The following steps are taken:

1. Return the \text{this} value.

The value of the \text{name} property of this function is \"\[\text{Symbol.iterator}\]\".

22.1.5.2.3 %ArrayIteratorPrototype% [@@toStringTag]

The initial value of the \[\text{toStringTag}\] 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 44.

<table>
<thead>
<tr>
<th>Table 44 — Internal Slots of Array Iterator Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal Slot</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>[IteratedObject]</td>
</tr>
<tr>
<td>[ArrayIteratorNextIndex]</td>
</tr>
<tr>
<td>[ArrayIterationKind]</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 45, for each of the nine supported element types. Each constructor in Table 45 has 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>ToFloat32</td>
<td>32-bit IEEE floating point</td>
<td>Float</td>
</tr>
<tr>
<td>Float64Array</td>
<td>Float64</td>
<td>8</td>
<td>ToFloat64</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 initializes 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 initialize 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.
8. Let constructorName be the string value of O’s [[TypedArrayName]] internal slot.
9. Let elementType be the string value of the Element Type value in Table 45 for constructorName.
10. Let numberLength be ToNumber(length).
11. Let elementLength be ToLength(numberLength).
12. ReturnIfAbrupt(elementLength).
13. If SameValueZero(numberLength, elementLength) is false, then throw a RangeError exception.
14. Let data be AllocateArrayBuffer(%ArrayBuffer%).
15. ReturnIfAbrupt(data).
16. If the value of O’s [[ViewedArrayBuffer]] internal slot is not undefined, then throw a TypeError exception.
17. If elementSize be the Element Size value in Table 45 for constructorName.
18. Let byteLength be elementType * elementLength.
19. Let status be SetArrayBufferData(data, byteLength).
20. ReturnIfAbrupt(status).
21. Set O’s [[ViewedArrayBuffer]] internal slot to data.
22. Set O’s [[ByteLength]] internal slot to byteLength.
23. Set O’s [[ByteOffset]] internal slot to 0.
24. Set O’s [[ArrayLength]] internal slot to elementLength.
25. 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 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 \texttt{undefined}, then throw a \texttt{TypeError} exception.
8. Let srcData be the value of srcArray’s [[ViewedArrayBuffer]] internal slot.
9. If srcData is \texttt{undefined}, then throw a \texttt{TypeError} exception.
10. If IsDetachedBuffer(srcData) is \texttt{true}, then throw a \texttt{TypeError} exception.
11. Let constructorName be the string value of O’s [[TypedArrayName]] internal slot.
12. Let elementType be the string value of the Element Type value in Table 45 for constructorName.
13. Let elementLength be the value of srcArray’s [[ArrayLength]] internal slot.
14. Let srcName be the string value of srcArray’s [[TypedArrayName]] internal slot.
15. Let srcType be the Element Size value in Table 45 for srcName.
16. Let srcElementSize be the Element Size value in Table 45 for srcName.
17. Let srcByteOffset be the value of srcArray’s [[ByteOffset]] internal slot.
18. Let byteLength be the Element Size value in Table 45 for constructorName.
19. Let bufferConstructor be Get(srcData, “constructor”).
20. If SameValue(elementType, srcType), then
   a. Let data be CloneArrayBuffer(srcData, srcByteOffset).
   b. ReturnIfAbrupt(data).
   c. Else, do the following:
      a. Let data be AllocateArrayBuffer(bufferConstructor, byteLength).
      b. ReturnIfAbrupt(data).
      c. If IsDetachedBuffer(srcData) is \texttt{true}, then throw a \texttt{TypeError} exception.
      d. Let status be SetArrayBufferData(data, srcByteOffset, elementSize, elementType, value).
      e. If SameValue(elementType, srcType), then
         a. Let data be CloneArrayBuffer(srcData, srcByteOffset).
         b. ReturnIfAbrupt(data).
         c. If IsDetachedBuffer(srcData) is \texttt{true}, then throw a \texttt{TypeError} exception.
         d. Let status be SetArrayBufferData(data, byteLength).
         e. ReturnIfAbrupt(status).
         f. Let srcType be the Element Size value in Table 45 for constructorName.
         g. Let srcElementSize be the Element Size value in Table 45 for srcName.
         h. Let srcByteOffset be srcByteOffset.
         i. Let targetByteIndex be 0.
         j. If count be elementLength, then
            a. Let value be GetValueFromBuffer(srcData, srcByteIndex, srcType).
            b. Let status be SetValueInBuffer(data, targetByteIndex, elementType, value).
            i. Set srcByteIndex to srcByteIndex + srcElementSize.
            j. Set targetByteIndex to targetByteIndex + elementSize.
            k. Decrement count by 1.
         l. Repeat, while count > 0.
21. If the value of O’s [[ViewedArrayBuffer]] internal slot is not \texttt{undefined}, then throw a \texttt{TypeError} exception.
22. Assert: O has not been reentrantly initialized.
23. Set O’s [[ViewedArrayBuffer]] internal slot 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. Set O’s [[TypedArrayName]] internal slot to constructorName.
28. Return O.

22.2.1.3 \texttt{%TypedArray% ( object )}

This description applies when the \texttt{%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.

\texttt{%TypedArray%} called with argument \texttt{object} performs the following steps:
1. Assert: Type(\texttt{object}) is Object and \texttt{object} does not have either a [[TypedArrayName]] or an [[ArrayBufferData]] internal slot.
2. Let O be the this value.
3. If Type(O) is not Object or 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 undefined, then throw a TypeError exception.
5. Assert: O has a [[ViewedArrayBuffer]] internal slot.
6. If the value of O’s [[ViewedArrayBuffer]] internal slot is not undefined, then throw a TypeError exception.
7. Return TypedArrayFrom(undefined, O, object, undefined, undefined).

22.2.1.4 %TypedArray% ( buffer [, byteOffset [, length ] ] )

This description applies when the %TypedArray% function is called with at least one argument and the Type of the first argument is Object and that object has an [[ArrayBufferData]] internal slot.

%TypedArray% called with arguments buffer, byteOffset, and length performs the following steps:

1. Assert: Type(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 IsDetachedBuffer(buffer) is true, then throw a TypeError exception.
5. If Type(O) is not Object or if O does not have a [[TypedArrayName]] internal slot, then throw a TypeError exception.
6. If the value of O’s [[TypedArrayName]] internal slot is undefined, then throw a TypeError exception.
7. Assert: O has a [[ViewedArrayBuffer]] internal slot.
8. If the value of O’s [[ViewedArrayBuffer]] internal slot is not undefined, then throw a TypeError exception.
9. Let constructorName be the string value of O’s [[TypedArrayName]] internal slot.
10. Let elementType be the string value of the Element Type value in Table 45 for constructorName.
11. Let elementSize be the Number value of the Element Size value in Table 45 for constructorName.
12. Let offset be ToInteger(byteOffset).
13. ReturnIfAbrupt(offset).
14. If offset < 0, then throw a RangeError exception.
15. If offset modulo elementSize ≠ 0, then throw a RangeError exception.
16. Let bufferByteLength be the value of buffer’s [[ArrayBufferByteLength]] internal slot.
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]] internal slot 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.

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 [, thisArg ]])

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 was supplied, let f be mapfn; otherwise let f be undefined.
6. If f is not undefined, then
   a. If IsCallable(f) is false, then throw a TypeError exception.
7. If thisArg was supplied, let t be thisArg; else let t be undefined.
8. Return TypedArrayFrom(C, undefined, items, f, t).

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 Typed Array 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 [[Set]] 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.

22.2.2.1.1 Runtime Semantics: TypedArrayFrom ( constructor, target, items, mapfn, thisArg )

When the TypedArrayFrom abstract operation is called with arguments constructor, target, items, mapfn, and thisArg, the following steps are taken:

1. Let C be constructor.
2. Assert: one of constructor and target is undefined.
3. Assert: If constructor is not undefined, then IsConstructor(C) is true.
4. Assert: target is either undefined or an Object that has been validated by the %TypedArray% constructor as described in 22.2.1.3
5. Assert: Type(items) is Object.
6. Assert: Type(mapfn) is either a callable Object or Undefined.
7. If mapfn is undefined, then let mapping be false.
8. else
   a. Let T be thisArg.
   b. Let mapping be true
9. Let usingIterator be CheckIterable(items).
10. ReturnIfAbrupt(usingIterator).
11. If `usingIterator` is not `undefined`, then
   a. Let `iterator` be `GetIterator(items, usingIterator)`.`
   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 `IteratorStep(iterator)`.
      ii. `ReturnIfAbrupt(next)`.
   f. If `next` is not `false`, then
      i. Let `nextValue` be `IteratorValue(next)`.
      ii. `ReturnIfAbrupt(nextValue)`.
      iii. Append `nextValue` to the end of the List `values`.
   g. Let `len` be the number of elements in `values`.
   h. Let `targetObj` be `TypedArrayAllocOrInit(C, target, len)`.
   i. `ReturnIfAbrupt(targetObj)`.
   j. Let `k` be 0.
   k. 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 `(kValue, k) as argumentsList`.
          2. `ReturnIfAbrupt(mappedValue)`.
      iv. Else, let `mappedValue` be `kValue`.
      v. Let `putStatus` be `Put(targetObj, Pk, mappedValue, true)`.
      vi. `ReturnIfAbrupt(putStatus)`.
      vii. Increase `k` by 1.
   l. `Assert: values` is now an empty List.
   m. `Return targetObj`.

12. `Assert: items` is not an Iterator so assume it is an array-like object.
13. Let `lenValue` be `Get(items, "length")`.
14. Let `len` be `ToLength(lenValue)`.
15. `ReturnIfAbrupt(len)`.
16. Let `targetObj` be `TypedArrayAllocOrInit(C, target, len)`.
17. `ReturnIfAbrupt(targetObj)`.
18. Let `k` be 0.
19. Repeat, while `k` < `len`
   a. Let `Pk` be `ToString(k)`.
   b. Let `kValue` be `Get(items, Pk)`.
   c. `ReturnIfAbrupt(kValue)`.
   d. If `mapping` is `true`, then
      i. Let `mappedValue` be the result of calling the `[[Call]]` internal method of `mapfn` with `T` as `thisArgument` and `(kValue, k) as argumentsList`.
      ii. `ReturnIfAbrupt(mappedValue)`.
   e. Else, let `mappedValue` be `kValue`.
   f. Let `putStatus` be `Put(targetObj, Pk, mappedValue, true)`.
   g. `ReturnIfAbrupt(putStatus)`.
   h. Increase `k` by 1.
20. `Return targetObj`.
22.2.2.1.2 Runtime Semantics: TypedArrayAllocOrInit(constructor, target, length)

When the TypedArrayAllocOrInit abstract operation is called with arguments constructor, target, and length, the following steps are taken:

1. Assert: one of constructor and target is undefined.
2. Assert: If constructor is not undefined, then IsConstructor(constructor) is true.
3. Assert: target is either undefined or an Object that has been validated by the %TypedArray% constructor as described in 22.2.1.3. However, side-effects of subsequent operations may have initialized target’s [[ViewedArrayBuffer]].
4. Assert: Type(length) is Number.
5. If target is undefined, then
   a. Let targetObj be the result of calling the [[Construct]] internal method of constructor with argument (length).
   b. ReturnIfAbrupt(targetObj).
6. Else,
   a. Let targetObj be target.
   b. Let constructorName be the string value of targetObj’s [[TypedArrayName]] internal slot.
   c. Let elementType be the string value of the Element Type value in Table 45 for constructorName.
   d. Let data be AllocateArrayBuffer(%ArrayBuffer%)
   e. ReturnIfAbrupt(data).
   f. Let ElementSize be the Element Size value in Table 45 for constructorName.
   g. Let byteLength be ElementSize × length.
   h. Let status be SetArrayBufferData(data, byteLength)
   i. ReturnIfAbrupt(status).
   j. Note: Side-effects of preceding steps may have already initialized targetObj.
   k. If the value of targetObj’s [[ViewedArrayBuffer]] internal slot is not undefined, then throw a TypeError exception.
   l. Set targetObj’s [[ViewedArrayBuffer]] internal slot to data.
   m. Set targetObj’s [[ByteLength]] internal slot to byteLength.
   n. Set targetObj’s [[ByteOffset]] internal slot to 0.
   o. Set targetObj’s [[ArrayLength]] internal slot to length.
7. Return targetObj.

22.2.2.%TypedArray%.of(...items)

When the of method is called with any number of arguments, the following steps are taken:

1. Let len be the actual number of arguments passed to this function.
2. Let items be the List of arguments passed to this function.
3. Let C be the this value.
4. If IsConstructor(C) is true, then
   a. Let newObj be the result of calling the [[Construct]] internal method of C with argument (len).
   b. ReturnIfAbrupt(newObj).
5. Else,
   a. Throw a TypeError exception.
6. Let k be 0.
7. Repeat, while k < len
   a. Let kValue be element k of items.
   b. Let Pk be ToString(k).
   c. Let status be Put(newObj, Pk, kValue., [[value]], true).
   d. ReturnIfAbrupt(status).
8. Return `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 initializes a length property that is initialized 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. Let `proto` be `GetPrototypeFromConstructor(F, "%TypedArrayPrototype%")`.
3. ReturnIfAbrupt(`proto`).
4. Let `obj` be `IntegerIndexedObjectCreate(proto, ([[ViewedArrayBuffer]], [[TypedArrayName]], [[ByteLength]], [[ByteOffset]], [[ArrayLength]])).`
5. Assert: The `[[ViewedArrayBuffer]]` internal slot of `obj` is `undefined`.
6. Assert: The `[[TypedArrayName]]` internal slot of `obj` is `undefined`.
7. Set the `[[ByteLength]]` internal slot of `obj` to 0.
8. Set the `[[ByteOffset]]` internal slot of `obj` to 0.
9. Set the `[[ArrayLength]]` internal slot of `obj` to 0.
10. 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 any other of the internal slots that are specific to `TypedArray` instance objects.

#### 22.2.3.1 `%TypedArray%.prototype.buffer`Get

`%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.

Commented [AWB13113]: buffer needs to be an accessor both to comply with WebIDL requirements and to support the Kronos neutering strawman requirements.
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 buffer be the value of O's [[ViewedArrayBuffer]] internal slot.
5. If buffer is undefined, then throw a TypeError exception.
6. If IsDetachedBuffer(buffer) is true, then return 0.
7. Let size be the value of O's [[ByteLength]] internal slot.
8. 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. If IsDetachedBuffer(buffer) is true, then return 0.
7. Let offset be the value of O's [[ByteOffset]] internal slot.
8. 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 ])

%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. A TypeError exception is also immediately thrown if the this value is detached or if its [[ViewedArrayBuffer]] is undefined.

The length property of the copyWithin method is 2.
22.2.3.6 %TypedArray%.prototype.entries()

The following steps are taken:
1. If Type(O) is not Object, throw a TypeError exception.
2. If O does not have a [[ViewedArrayBuffer]] internal slot throw a TypeError exception.
3. Let buffer be the value of O’s [[ViewedArrayBuffer]] internal slot.
4. If buffer is undefined, then throw a TypeError exception.
5. If IsDetachedBuffer(buffer) is true, throw a TypeError exception.
6. Return CreateArrayIterator(O, "key+value").

22.2.3.7 %TypedArray%.prototype.every(callbackfn[, thisArg])

%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 and must take into account the possibility that calls to callbackfn may cause the this value to become detached.

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[, end]])

%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])

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 buffer be the value of O’s [[ViewedArrayBuffer]] internal slot.
5. If buffer is undefined, then throw a TypeError exception.
6. If `IsDetachedBuffer(buffer)` is `true`, throw a `TypeError` exception.
7. Let `len` be the value of `O`'s `[[ArrayLength]]` internal slot.
8. If `IsCallable(callbackFn)` is `false`, throw a `TypeError` exception.
9. If `thisArg` was supplied, let `T` be `thisArg`; else let `T` be `undefined`.
10. Let `C` be `Get(O, "constructor")`.
11. ReturnIfAbrupt(C).
12. If `IsConstructor(C)` is `false`, then
    a. Throw a `TypeError` exception.
13. Let `kept` be a new empty List.
14. Let `k` be 0.
15. Let `captured` be 0.
16. Repeat, while `k < len`
    a. Let `Pk` be `ToString(k)`.
    b. Let `kValue` be `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.
17. Let `A` be the result of calling the `[[Construct]]` internal method of `C` with argument `(captured)`.
18. ReturnIfAbrupt(A).
19. Let `n` be 0.
20. For each element `e` of `kept`
    a. Let `status` be `Put(A, ToString(n), e, true)`.
    b. ReturnIfAbrupt(status).
    c. Increment `n` by 1.

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.

22.2.10 `%TypedArray%.prototype.find (predicate [ , thisArg ] )

 `%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. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm and must take into account the possibility that calls to `predicate` may cause the `this` value to become detached.

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. A `TypeError` exception is also immediately thrown if the `this` value is detached or if its `[[ViewedArrayBuffer]]` is `undefined`.

The `length` property of the `find` method is 1.
22.2.3.11 %TypedArray%.prototype.findIndex ( predicate [, thisArg ])

%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. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm and must take into account the possibility that calls to predicate may cause the this value to become detached.

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. A TypeError exception is also immediately thrown if the this value is detached or if its [[ViewedArrayBuffer]] is.

The length property of the findIndex method is 1.

22.2.3.12 %TypedArray%.prototype.forEach ( callbackfn [, thisArg ])

%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 and must take into account the possibility that calls to callbackfn may cause the this value to become detached.

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. A TypeError exception is also immediately thrown if the this value is detached or if its [[ViewedArrayBuffer]] is.

The length property of the forEach method is 1.

22.2.3.13 %TypedArray%.prototype.indexOf ( searchElement [, fromIndex ])

%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. A TypeError exception is also immediately thrown if the this value is detached or if its [[ViewedArrayBuffer]] is undefined.

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. A TypeError exception is also immediately thrown if the this value is detached or if its [[ViewedArrayBuffer]] is undefined.

22.2.3.15 %TypedArray%.prototype.keys()

The following steps are taken:
1. If Type(O) is not Object, throw a TypeError exception.
2. If O does not have a [[ViewedArrayBuffer]] internal slot throw a TypeError exception.
3. Let buffer be the value of O’s [[ViewedArrayBuffer]] internal slot.
4. If buffer is undefined, then throw a TypeError exception.
5. If IsDetachedBuffer(buffer) is true, throw a TypeError exception.
6. Return CreateArrayIterator(O, "key").

22.2.3.16 %TypedArray%.prototype.lastIndexOf (searchElement[, fromIndex])

%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. A TypeError exception is also immediately thrown if the this value is detached or if its [[ViewedArrayBuffer]] is undefined.

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. If IsDetachedBuffer(buffer) is true, then return 0.
8. Let length be the value of O’s [[ArrayLength]] internal slot.
9. 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.
%TypedArray%.prototype.map ( callbackfn [ , thisArg ] )

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 buffer be the value of O’s [[ViewedArrayBuffer]] internal slot.
5. If buffer is undefined, then throw a TypeError exception.
6. Let len be the value of O’s [[ArrayLength]] internal slot.
7. If IsCallable(callbackfn) is false, throw a TypeError exception.
8. If thisArg was supplied, let T be thisArg; else let T be undefined.
9. Let C be Get(O, “constructor”).
10. ReturnIfAbrupt(C).
11. 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).
12. Else, throw a TypeError exception.
13. Let k be 0.
14. Repeat, while k < len
   a. Let Pk be ToString(k).
   b. Let kValue be 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).
16. If IsDetachedBuffer(buffer) is true, throw a TypeError exception.

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.

%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 and must take into account the possibility that calls to callbackfn may cause the this value to become detached.
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. A TypeError exception is also immediately thrown if the this value is detached or if its [[ViewedArrayBuffer]] is undefined.

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 and must take into account the possibility that calls to callbackfn may cause the this value to become detached.

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. A TypeError exception is also immediately thrown if the this value is detached or if its [[ViewedArrayBuffer]] is undefined.

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. A TypeError exception is also immediately thrown if the this value is detached or if its [[ViewedArrayBuffer]] is undefined.

22.2.3.22 %TypedArray%.prototype.set(array [ , offset ])

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 a [[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. If IsDetachedBuffer(targetBuffer) is true, then throw a TypeError exception.
9. Let targetLength be the value of target's [[ArrayLength]] internal slot.
10. Let targetOffset be ToInteger (offset)
11. ReturnIfAbrupt(targetOffset).
12. If targetOffset < 0, then throw a **RangeError** exception.
13. Let targetName be the string value of target’s [[TypedArrayName]] internal slot.
14. Let targetElementSize be the Number value of the Element Size value specified in Table 45 for targetName.
15. Let targetType be the string value of the Element Type value in Table 45 for targetName.
16. Let targetByteOffset be the value of target’s [[ByteOffset]] internal slot.
17. Let src be ToObject(array).
18. ReturnIfAbrupt(src).
19. Let srcLen be Get(src, "length").
20. Let numberLength be ToNumber(srcLen).
21. Let srcLength be ToInteger(numberLength).
22. ReturnIfAbrupt(srcLength).
23. If numberLength ≠ srcLength or srcLength < 0, then throw a **TypeError** exception.
24. If srcLength + targetOffset > targetLength, then throw a **RangeError** exception.
25. Let targetByteIndex be targetOffset × targetElementSize + targetByteOffset.
26. Let k be 0.
27. Let limit be targetByteIndex + targetElementSize × min(srcLength, targetLength – targetOffset).
28. Repeat, while targetByteIndex < limit
   a. Let Pk be ToString(k).
   b. Let kValue be Get(src, Pk).
   c. Let kNumber be ToNumber(kValue).
   d. ReturnIfAbrupt(kNumber).
   e. If IsDetachedBuffer(targetBuffer) is true, then throw a **TypeError** exception.
   f. Perform SetValueInBuffer(targetBuffer, targetByteIndex, targetType, kNumber).
   g. Set k to k + 1.
   h. Set targetByteIndex to targetByteIndex + targetElementSize.
29. Return undefined.

22.2.3.23 %TypedArray%.prototype.set(typedArray [ , offset ] )

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 a [[ViewedArrayBuffer]] internal slot.
6. Let targetOffset be ToInteger (offset)
7. ReturnIfAbrupt(targetOffset).
8. If targetOffset < 0, then throw a **RangeError** exception.
9. Let targetBuffer be the value of target’s [[ViewedArrayBuffer]] internal slot.
10. If targetBuffer is undefined, then throw a **TypeError** exception.
11. If IsDetachedBuffer(targetBuffer) is true, then throw a **TypeError** exception.
12. Let targetLength be the value of target’s [[ArrayLength]] internal slot.
13. Let srcBuffer be the value of typedArray’s [[ViewedArrayBuffer]] internal slot.
14. If srcBuffer is undefined, then throw a **TypeError** exception.
15. If IsDetachedBuffer(srcBuffer) is true, then throw a **TypeError** exception.
16. Let targetName be the string value of target’s [[TypedArrayName]] internal slot.
17. Let targetType be the string value of the Element Type value in Table 45 for targetName.
18. Let targetElementSize be the Number value of the Element Size value specified in Table 45 for targetName.
19. Let targetByteOffset be the value of target’s [[ByteOffset]] internal slot.
20. Let srcName be the string value of typedArray’s [[TypedArrayName]] internal slot.
21. Let srcType be the string value of the Element Type value in Table 45 for srcName.
22. Let srcElementSize be the Number value of the Element Size value specified in Table 45 for srcName.
23. Let srcLength be the value of typedArray’s [[ArrayLength]] internal slot.
24. Let srcByteOffset be the value of typedArray’s [[ByteOffset]] internal slot.
25. If srcLength + targetOffset > targetLength, then throw a RangeError exception.
26. If SameValue(srcBuffer, targetBuffer) is true, then
   a. Let srcBuffer be CloneArrayBuffer(srcBuffer, srcByteOffset).
   b. ReturnIfAbrupt(srcBuffer).
   c. If IsDetachedBuffer(targetBuffer) is true, then throw a TypeError exception.
   d. Let srcByteIndex be 0.
27. Else, let srcByteIndex be srcByteOffset.
28. Let targetByteIndex be offset × targetElementSize + targetByteOffset.
29. Let limit be targetByteIndex + targetElementSize × min(srcLength, targetLength – targetOffset).
30. Repeat, while targetByteIndex < limit
   a. Let value be GetValueFromBuffer(srcBuffer, srcByteIndex, srcType).
   b. Let status be SetValueInBuffer(targetBuffer, targetByteIndex, targetType, value).
   c. Set srcByteIndex to srcByteIndex + srcElementSize.
   d. Set targetByteIndex to targetByteIndex + targetElementSize.
31. 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 21.1.3.22. 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 buffer be the value of O’s [[ViewedArrayBuffer]] internal slot.
5. If buffer is undefined, then throw a TypeError exception.
6. If IsDetachedBuffer(buffer) is true, throw a TypeError exception.
7. Note Side-effects of subsequent operations may still detach buffer, but that will be detected by any accesses that are made to the content of buffer.
8. Let len be the value of O’s [[ArrayLength]] internal slot.
9. Let relativeStart be ToInteger(start).
10. ReturnIfAbrupt(relativeStart).
11. If relativeStart is negative, let k be max(len + relativeStart, 0); else let k 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 max(final – k, 0).
16. Let C be Get(O, “constructor”).
17. ReturnIfAbrupt(C).
18. 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).
19. Else,
   a. Throw a TypeError exception.
20. Let n be 0.
21. Repeat, while k < final
   a. Let Pk be ToString(k).
   b. Let kValue be Get(O, Pk).
   c. ReturnIfAbrupt(kValue).
   d. Let status be Put(A, ToString(n), kValue, true).
   e. ReturnIfAbrupt(status).
   f. Increase k by 1.
   g. Increase n by 1.
22. 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 slice method is 2.

22.2.3.25 %TypedArray%.prototype.some ( callbackfn [ , thisArg ])

%TypedArray%.prototype.some is a distinct function that implements the same algorithm as Array.prototype.some as defined in 22.1.3.23 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 and must take into account the possibility that calls to callbackfn may cause the this value to become detached.

This function is not generic. If the this value is not a object with a [[TypedArrayName]] internal slot, a TypeError exception is immediately thrown if the this value is detached or if its [[ViewedArrayBuffer]] is undefined.

The length property of the 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 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 this value is an object that has a fixed length and whose integer indexed properties are not sparse. The only internal methods of the this object that the algorithm may call are [[Get]] and [[Set]].

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 it is called.

Upon entry, the following steps are performed to initialize evaluation of the sort function. These steps are used instead of the entry steps in 22.1.3.24:

1. Let obj be the this value as the argument.
2. If obj does not have a [[TypedArrayName]] internal slot, then throw a TypeError exception.
3. Let buffer be the value of obj's [[ViewedArrayBuffer]] internal slot.
4. If `buffer` is `undefined`, then throw a `TypeError` exception.
5. If `IsDetachedBuffer(buffer)` is `true`, then throw a `TypeError` exception.
6. 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 `x` and `y`, the following steps are taken:

1. Assert: Both `Type(x)` and `Type(y)` is `Number`.
2. If the argument `comparefn` is not `undefined`, then
   a. If `IsCallable(comparefn)` is `false`, throw a `TypeError` exception.
   b. Let `v` be 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`.
   c. ReturnIfAbrupt(v).
   d. If `IsDetachedBuffer(buffer)` is `true`, then throw a `TypeError` exception.
   e. If `v` is `NaN`, then return `+0`.
   f. Return `v`.
3. If `x` and `y` are both `NaN`, return `+0`.
4. If `x` is `NaN`, return `1`.
5. If `y` is `NaN`, return `−1`.
6. If `x < y`, return `−1`.
7. If `x > y`, return `1`.
8. Return `+0`.

NOTE 1 Because `NaN` always compares greater than any other value, `NaN` property values always sort to the end of the result when `comparefn` is not provided.

22.2.3.27 `%TypedArray%.prototype.subarray([begin[, end]])`

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 a `[[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`. 

© Ecma International 2014

511
18. Let `newLength` be `endIndex - beginIndex`.
19. Let `constructorName` be the string value of `O`’s `[[TypedArrayName]]` internal slot.
20. Let `elementType` be the string value of the `Element Type` value in Table 45 for `constructorName`.
21. Let `elementSize` be the `Number` value of the `Element Size` value specified in Table 45 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 `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 ([ reserved1 [ , reserved2 ] ])`

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 following steps are taken:
1. If `Type(O)` is not `Object`, throw a `TypeError` exception.
2. If `O` does not have a `[[ViewedArrayBuffer]]` internal slot throw a `TypeError` exception.
3. Let `buffer` be the value of `O`’s `[[ViewedArrayBuffer]]` internal slot.
4. If `buffer` is `undefined`, then throw a `TypeError` exception.
5. If `IsDetachedBuffer(buffer)` is `true`, throw a `TypeError` exception.
6. Return CreateArrayIterator(`O`, "value").

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. If the value of `O`’s `[[TypedArrayName]]` internal slot is `undefined`, throw a `TypeError` exception.
6. Assert: `name` is a `String` value.
7. Return `name`. 

512 © Ecma International 2014
This property has the attributes \{ [[Enumerable]]: false, [[Configurable]]: true \}.

The initial value of the \texttt{name} property of this function is \texttt{"get \[Symbol.toStringTag]\"}.

22.2.4 The \texttt{TypedArray} Constructors

Each of these \texttt{TypedArray} constructor objects has the structure described below, differing only in the name used as the constructor name instead of \texttt{TypedArray}, in Table 45.

When a \texttt{TypedArray} constructor is called as a function rather than as a constructor, it initializes a new \texttt{TypedArray} object. The \texttt{this} value passed in the call must be an Object with a [[TypedArrayName]] internal slot and a [[ViewedArrayBuffer]] internal slot whose value is \texttt{undefined}. The constructor function initializes the \texttt{this} value using the argument values.

The \texttt{TypedArray} constructors are designed to be subclassable. They may be used as the value of an \texttt{extends} clause of a class declaration. Subclass constructors that intended to inherit the specified \texttt{TypedArray} behaviour must include a \texttt{super} call to the \texttt{TypedArray} constructor to initialize subclass instances.

22.2.4.1 \texttt{TypedArray(... argumentsList)}

A \texttt{TypedArray} constructor with a list of arguments \texttt{argumentsList} performs the following steps:

1. Let \texttt{O} be the \texttt{this} value.
2. If \text{Type}(\texttt{O}) is not \texttt{Object}, then throw a \texttt{TypeError} exception.
3. If \texttt{O} does not have a [[TypedArrayName]] internal slot, then throw a \texttt{TypeError} exception.
4. If the value of \texttt{O}'s [[TypedArrayName]] internal slot is not \texttt{undefined}, then throw a \texttt{TypeError} exception.
5. Set \texttt{O}'s [[TypedArrayName]] internal slot to the String value from the constructor name column in the row of Table 45 corresponding to this constructor.
6. Let \texttt{F} be the active function object.
7. Let \texttt{realmF} be \texttt{GetFunctionRealm}(\texttt{F}).
8. Let \texttt{super be realmF.\[intrinsic\].\[\%TypedArray\]}.
9. Return the result of calling the [[Call]] internal method of \texttt{super} with \texttt{O} and \texttt{argumentsList} as arguments.

22.2.4.2 \texttt{new TypedArray(... argumentsList)}

A \texttt{TypedArray} constructor called as part of a new expression performs the following steps:

1. Let \texttt{F} be the \texttt{TypedArray} function object on which the \texttt{new} operator was applied.
2. Let \texttt{argumentsList} be the \texttt{argumentsList} argument of the [[Construct]] internal method that was invoked by the \texttt{new} operator.
3. Return \texttt{Construct}(\texttt{F}, \texttt{argumentsList}).

22.2.5 Properties of the \texttt{TypedArray} Constructors

The value of the [[Prototype]] internal slot of each \texttt{TypedArray} constructor is the \%TypedArray\% intrinsic object (22.2.1).

Each \texttt{TypedArray} constructor has a \texttt{name} property whose value is the String value of the constructor name specified for it in Table 45.
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 45 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 45 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 ofTypedArray 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 ECMA-Script language values. A distinct key value may only occur in one key/value pair within the Map’s collection. Distinct key values are discriminated using the SameValueZero comparison algorithm.

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 initializes its this value with the internal state necessary to support the Map.prototype built-in 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 ])

When the Map function is called with optional argument 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 adder be the result of Get(map, "set").
   b. ReturnIfAbrupt(adder).
   c. If IsCallable(adder) is false, throw a TypeError exception.
   d. Let iter be the result of GetIterator(ToObject(iterable)).
   e. ReturnIfAbrupt(iter).
8. If the value of map's [[MapData]] internal slot is not undefined, then throw a TypeError exception.
9. Assert: map has not been reentrantly initialized.
10. Set map's [[MapData]] internal slot to a new empty List.
11. If iter is undefined, then return map.
12. Repeat
    a. Let next be the result of IteratorStep(iter).
    b. ReturnIfAbrupt(next).
    c. If next is false, then return 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 a Map key and whose second element is the value to associate with that key.

Commented [AWB12120]: Note that using a method call for inserting pairs during initialization provides allows subclasses to be more expressive.
23.1.1.2 \texttt{new Map(... argumentsList)}

When \texttt{Map} is called as part of a \texttt{new} expression it is a constructor: it initializes a newly created object.

\texttt{Map} called as part of a new expression with argument list \texttt{argumentsList} performs the following steps:

1. Let \( F \) be the Map function object on which the \texttt{new} operator was applied.
2. Let \texttt{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{Construct}(\( F \), \texttt{argumentsList}).

If \texttt{Map} is implemented as an ECMAscript function object, its \texttt{[[Construct]]} internal method will perform the above steps.

23.1.2 Properties of the Map Constructor

The value of the \texttt{[[Prototype]]} internal slot of the Map constructor is the Function prototype object (19.2.3).

Besides the \texttt{length} property (whose value is 1), the Map constructor has the following properties:

23.1.2.1 \texttt{Map.prototype}

The initial value of \texttt{Map.prototype} is the Map prototype object (23.1.3).

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

23.1.2.2 \texttt{Map[ @@create ]()}

The \texttt{@@create} method of a Map function object \( F \) performs the following steps:

1. Let \( F \) be the \texttt{this} value.
2. Let \( \texttt{obj} \) be the result of calling \texttt{OrdinaryCreateFromConstructor}(\( F \), "\texttt{MapPrototype}" , ([\texttt{[MapData]}]) ).
3. Return \( \texttt{obj} \).

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} }.

23.1.3 Properties of the Map Prototype Object

The value of the \texttt{[[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 \texttt{[[MapData]]} internal slot.

23.1.3.1 \texttt{Map.prototype.clear()}

The following steps are taken:

1. Let \( M \) be the \texttt{this} value.
2. If \texttt{Type}(\( M \)) is not \texttt{Object}, then throw a \texttt{TypeError} exception.
3. If \( M \) does not have a \texttt{[[MapData]]} internal slot throw a \texttt{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. 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. If p.([[key]]) is not empty and SameValueZero(p.([[key]]), key) is true, then
      i. Set p.([[key]]) to empty.
      ii. Set p.([[value]]) to empty.
      iii. Return true.
7. 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. Return the result of calling the CreateMapIterator abstract operation with arguments M and "key+value".

23.1.3.5 Map.prototype.forEach (callbackfn [, thisArg])

NOTE 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. Repeat for each Record `[[key]], [[value]]` p that is an element of `entries`,
   a. If `p.[[key]]` is not empty and `SameValueZero(p.[[key]], key)` is `true`, then return `p.[[value]]`.
7. 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. Repeat for each Record `[[key]], [[value]]` p that is an element of `entries`,
   a. If `p.[[key]]` is not empty and `SameValueZero(p.[[key]], key)` is `true`, then return `true`.
7. 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. Repeat for each Record {{[key]}, [[value]]} p that is an element of entries:
   a. If p.[[key]] is not empty and SameValueZero(p.[[key]], key) is true, then
      i. Set p.[[value]] to value.
      ii. Return M.
7. If key is −0, then let key be +0.
8. Let p be the Record {{[key]}; key, [[value]]; value}.
9. 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.

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. If the value of \( map \)'s [[MapData]] internal slot is undefined, then throw a TypeError exception.
4. Let \( \text{iterator} \) be the result of ObjectCreate(%MapIteratorPrototype%, \([[[\text{Map}]],[[\text{MapNextIndex}]],[[[\text{MapIterationKind}]]]\]).
5. Set \( \text{iterator} \)'s [[Map]] internal slot to \( map \).
6. Set \( \text{iterator} \)'s [[MapNextIndex]] internal slot to 0.
7. Set \( \text{iterator} \)'s [[MapIterationKind]] internal slot to \( kind \).
8. Return \( \text{iterator} \).

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 %IteratorPrototype% intrinsic object (25.1.2). 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(\( \text{undefined} \), true)
8. Assert: \( m \) has a [[MapData]] internal slot and \( m \) has been initialized 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 \( \text{entries} \). The number of elements must be redetermined each time this method is evaluated.
   a. Let \( e \) be the Record \([[[\text{key}]], [[[\text{value}]])\} that is the value of \( \text{entries}[\text{index}] \).
   b. Set \( index \) to \( index+1 \);
   c. Set the [[MapNextIndex]] internal slot of \( O \) to \( index \).
   d. If \( e.[[\text{key}]] \) is not empty, then
If `itemKind` is "key" then, let `result` be `e.[[key]]`.

ii. Else if `itemKind` is "value" then, let `result` be `e.[[value]]`.

iii. Else,  
      1. Assert: `itemKind` is "key+value".
      2. Let `result` be the result of 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 `CreateIterResultObject(result, false)`.

11. Set the [[Map]] internal slot of `O` to `undefined`.
12. Return `CreateIterResultObject(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 46.

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Map]]</td>
<td>The Map object that is being iterated.</td>
</tr>
<tr>
<td>[[MapNextIndex]]</td>
<td>The integer index of the next Map data element to be examined by this iterator.</td>
</tr>
<tr>
<td>[[MapIterationKind]]</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 the `SameValueZero` comparison algorithm.

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 initializes its this value with the internal state necessary to support the Set.prototype built-in 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 ])

When the Set 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 [[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 not present, let iterable be undefined.
6. If iterable is either undefined or null, then let iter be undefined.
7. Else,
   a. Let adder be the result of Get(set, ”add”).
   b. ReturnIfAbrupt(adder).
   c. If IsCallable(adder) is false, throw a TypeError Exception.
   d. Let iter be the result of GetIterator(ToObject(iterable)).
   e. ReturnIfAbrupt(iter).
8. If the value of set’s [[SetData]] internal slot is not undefined, then throw a TypeError exception.
9. Assert: set has not been reentrantly initialized.
10. Set set’s [[SetData]] internal slot to a new empty List.
11. If iter is undefined, then return set.
12. 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. 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.
   f. ReturnIfAbrupt(status).

NOTE Using a method call for inserting values during initialization enables subclasses to that redefine add to still make a super call to the inherited constructor.

23.2.1.2 new Set ( ...argumentsList )

When Set is called as part of a new expression it is a constructor: it initializes 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 Construct(F, argumentsList).
If Set is implemented as an ECMAScript function object, its [[Construct]] internal method will perform the above steps.

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 1), 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]])).
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]] internal slot.

23.2.3.1 this.add(e) 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. Repeat for each e that is an element of entries,
   a. If e is not empty and SameValueZero(e, value) is true, then
      i. Return S.
7. If value is −0, then let value be +0.
8. Append value as the last element of entries.
9. Return S.

Commented [AWB14121]: Because the @@create method is essential to the integrity of this “class” definition, just like the prototype property, it seems appropriate to freeze it in the same manner.
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. Repeat for each e that is an element of entries,
   a. If e is not empty and SameValueZero(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.
    ii. Return false.
7. 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 ] )

NOTE 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.
If `callbackfn` is an Arrow Function, this was lexically bound when the function was created so `thisArg` will have no effect.

`callbackfn` is called with three arguments: the first two arguments are a value contained in the Set. The same value of passed for both arguments. The Set object being traversed is passed as the third argument.

The `callbackfn` is called with three arguments to be consistent with the callback 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`.

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. Repeat for each `e` that is an element of `entries`,
   a. If `e` is not empty and `SameValueZero(e, value)` is `true`, then return `true`.
7. 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 initialization by the Set constructor, Set instances also have a [[SetData]] 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 \( \text{iterator} \) be the result of \( \text{ObjectCreate}(%\text{SetIteratorPrototype}%, \{[[\text{IteratedSet}]],
[[\text{SetNextIndex}]], [[\text{SetIterationKind}]]\}) \).
5. Set \( \text{iterator} \)'s [[IteratedSet]] internal slot to \( \text{set} \).
6. Set \( \text{iterator} \)'s [[SetNextIndex]] internal slot to 0.
7. Set \( \text{iterator} \)'s [[SetIterationKind]] internal slot to \( \text{kind} \).
8. Return \( \text{iterator} \).

### 23.2.5.2 The %\text{SetIteratorPrototype}% Object

All Set Iterator Objects inherit properties from the %\text{SetIteratorPrototype}% intrinsic object. The %\text{SetIteratorPrototype}% intrinsic object is an ordinary object and its [[Prototype]] internal slot is the %\text{SetIteratorPrototype}% intrinsic object (25.1.2). In addition, %\text{SetIteratorPrototype}% has the following properties:

#### 23.2.5.2.1 %\text{SetIteratorPrototype}%.next()

1. Let \( O \) be the \( \text{this} \) value.
2. If Type(\( O \)) is not Object, throw a \( \text{TypeError} \) exception.
3. If \( O \) does not have all of the internal slots of a Set Iterator Instance (23.2.5.3), throw a \( \text{TypeError} \) 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. If \( s \) is \( \text{undefined} \), then return CreateIterResultObject(\( \text{undefined} \), \( \text{true} \)).
8. Assert: \( s \) has a [[SetData]] internal slot and \( s \) has been initialized so the value of [[SetData]] is not \( \text{undefined} \).
9. Let \( \text{entries} \) be the List that is the value of the [[SetData]] internal slot of \( s \).
10. Repeat while \( index \) is less than the total number of elements of \( \text{entries} \). The number of elements must be redetermined each time this method is evaluated.
    a. Let \( e \) be \( \text{entries}[\text{index}] \).
    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 \( \text{result} \) be the result of performing ArrayCreate(2).
            2. Assert: \( \text{result} \) is a new, well-formed Array object so the following operations will never fail.
            3. Call CreateDataProperty(\( \text{result} \), "0", \( e \)).
            4. Call CreateDataProperty(\( \text{result} \), "1", \( e \)).
            5. Return CreateIterResultObject(\( \text{result} \), \( \text{false} \)).
        ii. Return CreateIterResultObject(\( e \), \( \text{false} \)).
    11. Set the [[IteratedSet]] internal slot of \( O \) to \( \text{undefined} \).
    12. Return CreateIterResultObject(\( \text{undefined} \), \( \text{true} \)).

#### 23.2.5.2.2 %\text{SetIteratorPrototype}% @@iterator()

The following steps are taken:

1. Return the \( \text{this} \) value.

The value of the name property of this function is "[Symbol.iterator]".
23.2.5.2.3 %SetIteratorPrototype%[@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 %SetIteratorPrototype% intrinsic object. Set Iterator instances are initially created with the internal slots specified in Table 47.

Table 47 — Internal Slots of Set Iterator Instances

<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 iterator.</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 a 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 initializes its this value with the internal state necessary to support the WeakMap.prototype built-in 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 ])

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 map does not have a [[WeakMapData]] internal slot, then throw a TypeError exception.
4. If map'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 adder be the result of Get(map, "set").
   b. ReturnIfAbrupt(adder).
   c. If IsCallable(adder) is false, throw a TypeError Exception.
   d. Let iter be the result of GetIterator(ToObject(iterable)).
   e. ReturnIfAbrupt(iter).
7. Else,
   a. Let next be the result of IteratorStep(iter).
   b. ReturnIfAbrupt(next).
   c. If next is false, then return 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. ReturnIfAbrupt(k).
   i. Let v be the result of Get(nextValue, "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).

Commented [AWB12123]: Note that using a method call for inserting pairs during initialization provides allows subclasses to be more expressive.
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 initializes 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 Construct(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 1), 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.
2. Let obj be the result of calling OrdinaryCreateFromConstructor(F, "%WeakMapPrototype%", ( [[WeakMapData]] )).
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 }.

23.3.3 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]]) is not empty and SameValue(p.([[key]]), key) is true, 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]]) is not empty and SameValue(p.([[key]]), key) is true, 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]] is not empty and SameValue(p.[[key]], key) is true, 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]] is not empty and SameValue(p.[[key]], key) is true, 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 initializes its this value with the internal state necessary to support the WeakSet.prototype built-in 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 ])

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 it be undefined.
7. Else,
   a. Let adder be the result of Get(set, "add").
   b. ReturnIfAbrupt(adder).
   c. If IsCallable(adder) is false, throw a TypeError Exception.
   d. Let iter be the result of GetIterator(ToObject(iterable)).
   e. ReturnIfAbrupt(iter).
8. If the value of set's [[WeakSetData]] internal slot is not undefined, then throw a TypeError exception.
9. Assert: set has not been reentrantly initialized.
10. Set set's [[WeakSetData]] internal slot to a new empty List.
11. If iter is undefined, then return set.
12. 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).
23.4.1.2  new WeakSet ( ...argumentsList)

When WeakSet is called as part of a new expression it is a constructor: it initializes 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 Construct(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 1), 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,
   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 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,
   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 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.
   b. 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 initialization 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 uninitialized ArrayBuffer object. It performs the following steps:

1. Let obj be OrdinaryCreateFromConstructor(constructor, "%ArrayBufferPrototype%",( [[ArrayBufferData]], [[ArrayBufferByteLength]]) ).
2. ReturnIfAbrupt(obj).
3. Set the [[ArrayBufferData]] internal slot of obj to null.
4. Set the [[ArrayBufferByteLength]] internal slot of obj to 0.
5. Return obj.

24.1.1.2 IsDetachedBuffer( arrayBuffer )

The abstract operation IsDetachedBuffer with argument arrayBuffer performs the following steps:

1. Assert: Type(arrayBuffer) is Object and it has [[ArrayBufferData]] internal slot.
2. If arrayBuffer’s [[ArrayBufferData]] internal slot is null, then return true.
3. Return false.

24.1.1.3 DetachArrayBuffer( arrayBuffer )

The abstract operation DetachArrayBuffer with argument arrayBuffer performs the following steps:

1. Assert: Type(arrayBuffer) is Object and it has [[ArrayBufferData]] and [[ArrayBufferByteLength]] internal slots.
2. Set arrayBuffer’s [[ArrayBufferData]] internal slot to null.
3. Set arrayBuffer’s [[ArrayBufferByteLength]] internal slot to 0.
4. Return NormalCompletion(null).

NOTE Detaching an ArrayBuffer instance disassociates the Data Block used as its backing store from the instance and sets the byte length of the buffer to 0. No operations defined by this specification uses the
24.1.1.4 SetArrayBufferData( arrayBuffer, bytes )

The abstract operation SetArrayBufferData with arguments `arrayBuffer` and `bytes` is used to initialize 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 CreateByteDataBlock(`bytes`).
5. ReturnIfAbrupt(`block`).
6. Set `arrayBuffer`'s `[[ArrayBufferData]]` internal slot to `block`.
7. Set `arrayBuffer`'s `[[ArrayBufferByteLength]]` internal slot to `bytes`.
8. Return `arrayBuffer`.

24.1.1.5 CloneArrayBuffer( srcBuffer, srcByteOffset )

The abstract operation CloneArrayBuffer takes two parameters, an ArrayBuffer `srcBuffer` and 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, then throw a `TypeError` exception.
4. If IsDetachedBuffer(`srcBuffer`) is true, then throw a `TypeError` exception.
5. Let `srcLength` be the value of `srcBuffer`'s `[[ArrayBufferByteLength]]` internal slot.
6. Let `bufferConstructor` be Get(`srcBuffer`, "constructor").
7. ReturnIfAbrupt(`bufferConstructor`).
9. Let `cloneLength` be `srcLength` - `srcByteOffset`.
10. If `bufferConstructor` is undefined, then let `bufferConstructor` be `%ArrayBuffer%`.
11. Let `targetBuffer` be AllocateArrayBuffer(`bufferConstructor`).
12. NOTE: Side-effects of the above steps may have detached `srcBuffer`.
13. If IsDetachedBuffer(`srcBuffer`) is true, then throw a `TypeError` exception.
14. Let `srcBlock` be the value of `srcBuffer`'s `[[ArrayBufferData]]` internal slot.
15. Let `status` be SetArrayBufferData(`targetBuffer`, `cloneLength`).
16. ReturnIfAbrupt(`status`).
17. Let `targetBlock` be the value of `targetBuffer`'s `[[ArrayBufferData]]` internal slot.
18. Perform CopyDataBlockBytes(`targetBlock`, 0, `srcBlock`, `srcByteOffset`, `cloneLength`).

24.1.1.6 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`. This operation performs the following steps:

1. Assert: `arrayBuffer` has been initialized.
2. Assert: IsDetachedBuffer(`arrayBuffer`) is false.
3. Assert: There are sufficient bytes in `arrayBuffer` starting at `byteIndex` to represent a value of `type`.
4. Assert: `byteIndex` is a positive integer.
5. Let `block` be `arrayBuffer`'s `[[ArrayBufferData]]` internal slot.
6. Let `elementSize` be the Number value of the Element Size value specified in Table 45 for Element Type type.
7. Let `rawValue` be a List of `elementSize` containing, in order, the `elementSize` bytes starting at `byteIndex` of block.
8. If `isLittleEndian` is not present, 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.
9. If `isLittleEndian` is `false`, reverse the order of the elements of `rawValue`.
10. 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`.
11. 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 corresponds to `value`.
12. If the first code unit 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.
13. 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` x 8.
14. Return the Number value that corresponds to `intValue`.

24.1.1.7 `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`. This operation performs the following steps:

1. **Assert:** `arrayBuffer` has been initialized.
2. **Assert:** `isDetachedBuffer(arrayBuffer)` is `false`.
3. **Assert:** There are sufficient bytes in `arrayBuffer` starting at `byteIndex` to represent a value of `type`.
4. **Assert:** `byteIndex` is a positive integer.
5. **Assert:** `Type(value)` is `Number`.
6. Let `block` be `arrayBuffer`’s [[ArrayBufferData]] internal slot.
7. If `block` is `undefined`, then throw a `TypeError` exception.
8. Let `elementSize` be the Number value of the Element Size specified in Table 45 for Element Type `type`.
9. If `isLittleEndian` is not present, 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.
10. If `type` is “Float32”, then
    a. Set `rawValue` to a List containing the 4 bytes that are the result of converting `value` to IEEE-856-2008 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. If `value` is `NaN`, `rawValue` may be set to any implementation chosen non-signaling `NaN` encoding.
11. Else, if `type` is “Float64”, then
a. Set `rawValue` to a List containing the 8 bytes that are the IEEE-868-2008 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.

12. Else, 
   a. Let `n` be the Number value of the Element Size specified in Table 45 for Element Type `type`.
   b. Let `convOp` be the abstract operation named in the Conversion Operation column in Table 45 for Element Type `type`.
   c. Let `intValue` be the result of calling `convOp` with `value` as its argument.
   d. If `intValue` ≥ 0, then 
      i. 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.
   e. Else, 
      i. 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.

13. Store the individual bytes of `rawBytes` in order starting at position `byteIndex` of `block`.

14. 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 initializes 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 initialize 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 throw a `TypeError` exception.
3. Let `numberLength` 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. If the value of `O`’s `[[ArrayBufferData]]` internal slot is not `undefined`, then throw a `TypeError` exception.
8. 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 Construct(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 an [[ArrayBufferData]] or [[ArrayBufferByteLength]] internal slot.

#### 24.1.4.1 get 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, then throw a TypeError exception.
5. If IsDetachedBuffer(O) is true, then throw a TypeError exception.
6. Let length be the value of O’s [[ArrayBufferByteLength]] internal slot.
7. 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, then throw a TypeError exception.
5. If IsDetachedBuffer(O) is true, then throw a TypeError exception.
6. Let len be the value of O’s [[ArrayBufferByteLength]] internal slot.
7. Let relativeStart be ToInteger(start).
8. ReturnIfAbrupt(relativeStart).
9. If relativeStart is negative, let first be max((len + relativeStart),0); else let first be min(relativeStart, len).
10. If end is undefined, let relativeEnd be len; else let relativeEnd be ToInteger(end).
11. ReturnIfAbrupt(relativeEnd).
12. If relativeEnd is negative, let final be max((len + relativeEnd),0); else let final be min(relativeEnd, len).
13. Let newLen be max(final - first,0).
14. Let ctor be the result of Get(O, "constructor").
15. ReturnIfAbrupt(ctor).
16. If IsConstructor(ctor) is false, then throw a TypeError exception.
17. Let new be the result of calling the [[Construct]] internal method of ctor with a new List containing the single element newLen.
18. ReturnIfAbrupt(new).
19. If new does not have an [[ArrayBufferData]] internal slot throw a TypeError exception.
20. If the value of new’s [[ArrayBufferData]] internal slot is undefined, then throw a TypeError exception.
21. If IsDetachedBuffer(new) is true, then throw a TypeError exception.
22. If SameValue(new, O) is true, then throw a TypeError exception.
23. If the value of new’s [[ArrayBufferByteLength]] internal slot < newLen, then throw a TypeError exception.
24. NOTE: Side effects of the above steps may have detached O.
25. If IsDetachedBuffer(O) is true, then throw a TypeError exception.
26. Let fromBuf be the value of O’s [[ArrayBufferData]] internal slot.
27. Let toBuf be the value of new’s [[ArrayBufferData]] internal slot.
28. Perform CopyDataBlockBytes(toBuf, 0, fromBuf, first, newLen).
29. Return new.

24.1.4.4 ArrayBuffer.prototype[ @@toStringTag ]

The initial value of the @@toStringTag property is the string value "ArrayBuffer".

25. [AWB26125]: Note the Khronos spec. said that neutered arraybuffers have a byteLength of 0 abut the June 2014 TC39 meeting decide that accessing it should be an error.
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 detached 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 numberIndex be ToNumber(requestIndex)
4. Let getIndex be ToInteger(numberIndex).
5. ReturnIfAbrupt(getIndex).
6. If numberIndex ≠ getIndex or getIndex < 0, then throw a RangeError exception.
7. Let isLittleEndian be ToBoolean(isLittleEndian).
8. ReturnIfAbrupt(isLittleEndian).
9. Let buffer be the value of view’s [[ViewedArrayBuffer]] internal slot.
10. If buffer is undefined, then throw a TypeError exception.
11. If IsDetachedBuffer(buffer) is true, then throw a TypeError exception.
12. Let viewOffset be the value of view’s [[ByteOffset]] internal slot.
13. Let viewSize be the value of view’s [[ByteLength]] internal slot.
14. Let elementSize be the Number value of the Element Size value specified in Table 45 for Element Type type.
15. If getIndex + elementSize > viewSize, then throw a RangeError exception.
16. Let bufferIndex be getIndex + viewOffset.
17. Return the result of GetValueFromBuffer(buffer, bufferIndex, type, 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 numberIndex be ToNumber(requestIndex)
4. Let getIndex be ToInteger(numberIndex).
5. ReturnIfAbrupt(getIndex).
6. If numberIndex ≠ getIndex or getIndex < 0, then throw a RangeError exception.
7. Let isLittleEndian be ToBoolean(isLittleEndian).
8. ReturnIfAbrupt(isLittleEndian).
9. Let buffer be the value of view’s [[ViewedArrayBuffer]] internal slot.
10. If buffer is undefined, then throw a TypeError exception.

11. If IsDetachedBuffer(buffer) is true, then throw a TypeError exception.
12. Let viewOffset be the value of view’s [[ByteOffset]] internal slot.
13. Let viewSize be the value of view’s [[ByteLength]] internal slot.
14. Let elementsSize be the Number value of the Element Size value specified in Table 45 for Element Type.
15. If getIndex + elementsSize > viewSize, then throw a RangeError exception.
16. Let bufferIndex be getIndex + viewOffset.
17. Return the result of SetValueInBuffer(buffer, bufferIndex, type, value, 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 initializes 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 intended to inherit the specified DataView behaviour must include a super call to the DataView constructor to initialize subclass instances.

24.2.2.1 DataView (buffer [ , byteOffset [ , byteLength ] ])

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.
7. If the value of buffer’s [[ArrayBufferData]] internal slot is undefined, then throw a TypeError exception.
8. Let numberOffset be ToNumber(byteOffset).
9. Let offset be ToInteger(numberOffset).
10. ReturnIfAbrupt(offset).
11. If numberOffset ≠ offset or offset < 0, then throw a RangeError exception.
12. Let bufferByteLength be the value of buffer’s [[ArrayBufferByteLength]] internal slot.
13. If offset > bufferByteLength, then throw a RangeError exception.
14. If byteLength is undefined, then
   a. Let viewByteLength be bufferByteLength – offset.
15. Else,
   a. Let numberLength be ToInteger(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.
16. If the value of O's `[[ViewedArrayBuffer]]` internal slot is not undefined, then throw a `TypeError` exception.
17. Set O's `[[ViewedArrayBuffer]]` internal slot to `buffer`.
18. Set O's `[[ByteLength]]` internal slot to `viewByteLength`.
19. Set O's `[[ByteOffset]]` internal slot to `offset`.
20. Return O.

24.2.2.2 `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 `Construct(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. If IsDetachedBuffer(buffer) is true, then throw a TypeError exception.
7. Let size be the value of O’s [[ByteLength]] internal slot.
8. 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. If IsDetachedBuffer(buffer) is true, then throw a TypeError exception.
7. Let offset be the value of O’s [[ByteOffset]] internal slot.
8. 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])

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 littleEndian is not present, then let littleEndian be false.
3. Return the result of GetViewValue(v, byteOffset, littleEndian, "Float32").

24.2.4.6 DataView.prototype.getFloat64 (byteOffset [, littleEndian])

When the getFloat64 method is called with argument byteOffset 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 GetViewValue(v, byteOffset, littleEndian, "Float64").

24.2.4.7 DataView.prototype.getInt8 (byteOffset)

When the getInt8 method is called with argument byteOffset the following steps are taken:

1. Let v be the this value.
2. Return the result of GetViewValue(v, byteOffset, true, "Int8").

24.2.4.8 DataView.prototype.getInt16 (byteOffset [, littleEndian])

When the getInt16 method is called with argument byteOffset 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 GetViewValue(v, byteOffset, littleEndian, "Int16").

24.2.4.9 DataView.prototype.getInt32 (byteOffset [, littleEndian])

When the getInt32 method is called with argument byteOffset 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 undefined.
3. Return the result of GetViewValue(v, byteOffset, littleEndian, "Int32").

24.2.4.10 DataView.prototype.getUint8 (byteOffset)

When the getUint8 method is called with argument byteOffset the following steps are taken:

1. Let v be the this value.
2. Return the result of GetViewValue(v, byteOffset, true, "Uint8").

24.2.4.11 DataView.prototype.getUint16 (byteOffset [, littleEndian])

When the getUint16 method is called with argument byteOffset and optional argument littleEndian the following steps are taken:
1. Let \( v \) be the \texttt{this} value.
2. If \texttt{littleEndian} is not present, then let \texttt{littleEndian} be \texttt{false}.
3. Return the result of GetViewValue(\( v, \text{byteOffset}, \text{littleEndian}, \text{"Uint16"} \)).

24.2.4.12 \texttt{DataView.prototype.getInt32 (byteOffset [, littleEndian])}

When the \texttt{getInt32} method is called with argument \texttt{byteOffset} and optional argument \texttt{littleEndian} the following steps are taken:

1. Let \( v \) be the \texttt{this} value.
2. If \texttt{littleEndian} is not present, then let \texttt{littleEndian} be \texttt{false}.
3. Return the result of GetViewValue(\( v, \text{byteOffset}, \text{littleEndian}, \text{"Uint32"} \)).

24.2.4.13 \texttt{DataView.prototype.setFloat32 (byteOffset, value [, littleEndian])}

When the \texttt{setFloat32} method is called with arguments \texttt{byteOffset} and \texttt{value} and optional argument \texttt{littleEndian} the following steps are taken:

1. Let \( v \) be the \texttt{this} value.
2. If \texttt{littleEndian} is not present, then let \texttt{littleEndian} be \texttt{false}.
3. Return the result of SetViewValue(\( v, \text{byteOffset}, \text{littleEndian}, \text{"Float32"}, \text{value} \)).

24.2.4.14 \texttt{DataView.prototype.setFloat64 (byteOffset, value [, littleEndian])}

When the \texttt{setFloat64} method is called with arguments \texttt{byteOffset} and \texttt{value} and optional argument \texttt{littleEndian} the following steps are taken:

1. Let \( v \) be the \texttt{this} value.
2. If \texttt{littleEndian} is not present, then let \texttt{littleEndian} be \texttt{false}.
3. Return the result of SetViewValue(\( v, \text{byteOffset}, \text{littleEndian}, \text{"Float64"}, \text{value} \)).

24.2.4.15 \texttt{DataView.prototype.setInt8 (byteOffset, value)}

When the \texttt{setInt8} method is called with arguments \texttt{byteOffset} and \texttt{value} the following steps are taken:

1. Let \( v \) be the \texttt{this} value.
2. Return the result of SetViewValue(\( v, \text{byteOffset}, \text{true}, \text{"Int8"}, \text{value} \)).

24.2.4.16 \texttt{DataView.prototype.setInt16 (byteOffset, value [, littleEndian])}

When the \texttt{setInt16} method is called with arguments \texttt{byteOffset} and \texttt{value} and optional argument \texttt{littleEndian} the following steps are taken:

1. Let \( v \) be the \texttt{this} value.
2. If \texttt{littleEndian} is not present, then let \texttt{littleEndian} be \texttt{false}.
3. Return the result of SetViewValue(\( v, \text{byteOffset}, \text{littleEndian}, \text{"Int16"}, \text{value} \)).

24.2.4.17 \texttt{DataView.prototype.setInt32 (byteOffset, value [, littleEndian])}

When the \texttt{setInt32} method is called with arguments \texttt{byteOffset} and \texttt{value} and optional argument \texttt{littleEndian} the following steps are taken:

1. Let \( v \) be the \texttt{this} value.
2. If \texttt{littleEndian} is not present, then let \texttt{littleEndian} be \texttt{false}.
3. Return the result of SetViewValue(v, byteOffset, littleEndian, "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 SetViewValue(v, byteOffset, true, "Uint8", value).

24.2.4.19 DataView.prototype.setUint16 (byteOffset, value [, littleEndian])

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])

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 \[\text{[[Call]}\] 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 Initializers and Object Initializers. 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 what it received, the structure is not modified. If it returns `undefined` then the property is deleted from the result.

1. Let `JText` be `ToString(text)`.
2. ReturnIfAbrupt(`JText`).
3. Parse `JText` interpreted as UTF-16 encoded Unicode points as a JSON text as specified in ECMA-404. Throw a `SyntaxError` exception if `JText` is not a valid JSON text as defined in that specification.
4. Let `scriptText` be the result of concatenating `( "", `JText`, and " ");`
5. Let `completion` be the result of parsing and evaluating `scriptText` as if it was the source text of an ECMAScript `Script`, but using the alternative definition of `DoubleStringCharacter` provided below. The extended PropertyDefinitionEvaluation semantics defined in B.3.1 must not be used during the evaluation.
6. Let `unfiltered` be `completion.[[value]]`.
7. Assert: `unfiltered` will be either a primitive value or an object that is defined by either an `ArrayLiteral` or an `ObjectLiteral`.
8. If `IsCallable(reviver)` is `true`, then
   a. Let `root` be `ObjectCreate(%ObjectPrototype%)`.
   b. Let `status` be the result of `CreateDataProperty(root, the empty String, unfiltered)`.
   c. Assert: `status` is `true`.
   d. Return the result of calling the abstract operation `Walk`, passing `root` and the empty String. The abstract operation `Walk` is described below.
9. Else
   a. Return `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 `scriptText` in step 5:

\[
\text{DoubleStringCharacter ::}
\begin{align*}
\text{SourceCharacter} & \text{ but not one of } " \text{ or } \\text{ or } \text{U+0000 through U+001F} \\
\text{\textbackslash EscapeSequence} & \\
\end{align*}
\]

- The CV of `DoubleStringCharacter :: SourceCharacter but not one of " or \ or U+0000 through U+001F` is the UTF-16Encoding (10.1.1) of the code point value of `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 `JText` conforms to that subset. When `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 *holder* object and the String *name* of a property in that object. Walk uses the value of *reviver* that was originally passed to the above parse function.

1. Let *val* be the result of Get(*holder, name*).
2. ReturnIfAbrupt(*val*).
3. If *val* is an object, then
   a. If *val* is an exotic Array object then
      i. Set *I* to 0.
      ii. Let *len* be the result of Get(*val, "length").
      iii. Assert: *len* is not an abrupt completion and its value is a positive integer.
      iv. Repeat while *I* < *len*,
         1. Let *newElement* be the result of calling the abstract operation Walk, passing *val* and ToString(*I*).
         2. If *newElement* is *undefined*, then
            a. Let *status* be the result of calling the [[Delete]] internal method of *val* with ToString(*I*) as the argument.
            b. NOTE This algorithm intentionally does not throw an exception if status is false.
         3. ReturnIfAbrupt(*status*).
         4. Add 1 to *I*.
   b. Else
      i. Let *keys* be EnumerableOwnNames(*val*).
      ii. For each String *P* in *keys* do,
         1. Let *newElement* be the result of calling the abstract operation Walk, passing *val* and *P*.
         2. If *newElement* is *undefined*, then
            a. Let *status* be the result of calling the [[DefineOwnProperty]] internal method of *val* with arguments ToString(*I*) andPropertyDescriptor{[[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 [[NumberData]] internal slot then,
      i. Let space be ToNumber(space).
   b. Else if space has a [[StringData]] internal slot then,
      i. Let space be ToString(space).
6. If Type(space) is Number
   a. Let space be min(10, ToInteger(space)).
   b. Set gap to a String containing space occurrences of code unit 0x0020 (SPACE). This will be the empty String if space is less than 1.
7. Else if Type(space) is String
   a. If the number of elements in space is 10 or less, set gap to space otherwise set gap to a String consisting of the first 10 elements of space.
8. Else
   a. Set gap to the empty String.
9. Let wrapper be ObjectCreate(%ObjectPrototype%).
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:

```
  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 code units " and \ are escaped with \ prefixes. Control characters code units are replaced with escape sequences \\uHHH, or with the shorter forms, \b (BACKSPACE), \t (FORM FEED), \f (LINE FEED), \r (CARRIAGE RETURN), \n (CHARACTER TABULATION).

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 `toJSON` be the result of Get(value, "toJSON").
   b. If IsCallable(toJSON) is true
      i. Let `value` be the result of calling the [[Call]] internal method of `toJSON` passing `value` as `thisArgument` and a List containing `key` as argumentsList.
      ii. ReturnIfAbrupt(value).
4. If ReplacerFunction is not undefined, then
   a. Let `value` be the result of calling the [[Call]] internal method of ReplacerFunction passing `holder` as the this value and with an argument list consisting of `key` and `value`.
   b. ReturnIfAbrupt(value).
5. If Type(value) is Object then,
   a. If value has 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 code units within it.

1. Let `product` be code unit U+0022 (QUOTATION MARK).
2. For each code unit `C` in `value`
   a. If `C` is U+0022 or U+005C (REVERSE SOLIDUS)
      i. Let `product` be the concatenation of `product` and code unit U+005C.
      ii. Let `product` be the concatenation of `product` and `C`.
   b. Else if `C` is backspace, formfeed, newline, carriage return, or tab
      i. Let `product` be the concatenation of `product` and code unit U+005C (REVERSE SOLIDUS).
      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 U+0020 (SPACE)
      i. Let `product` be the concatenation of `product` and code unit U+005C (REVERSE SOLIDUS).
      ii. 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 Latin letters.
      iii. 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 U+0022 (QUOTATION MARK).
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`.
6. Else
   a. Let `K` be `EnumerableOwnNames(value)`.
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 \(U+0020\) (SPACE).
   iv. Let \(member\) be the concatenation of \(member\) and \(strP\).
   v. Append \(member\) to \(partial\).

9. If \(partial\) is empty, then
   a. Let \(final\) be "{}".

10. Else
   a. If \(gap\) is the empty string
      i. Let \(properties\) be a String formed by concatenating all the element Strings of \(partial\) with
         each adjacent pair of Strings separated with code unit \(U+002C\) (COMMA). 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 \(U+002C\) (COMMA),
         code unit \(U+000A\) (LINE FEED), 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 \(U+000A\) (LINE FEED), \(indent\),
           \(properties\), code unit \(U+000A\), \(stepback\), and " } ".

11. Remove the last element of \(stack\).
12. Let \(indent\) be \(stepback\).
13. Return \(final\).

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 nonnegative 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 \(partial\).
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 U+002C (COMMA). 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 U+002C (COMMA), code unit U+000A (LINE FEED), 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 U+000A (LINE FEED), indent, properties, code unit U+000A, 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 Iteration

25.1.1 Common Iteration Interfaces

An interface is a set of property keys whose associated values match a specific specification. Any object that provides all the properties as described by an interface’s specification conforms to that interface. An interface is not represented by an distinct object. There may be many separately implemented objects that conform to any interface. An individual object may conform to multiple interfaces.

25.1.1.1 The Iterable Interface

The Iterable interface includes the following property:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>@@iterator</td>
<td>A zero arguments function that returns an object.</td>
<td>The function returns an object that conforms to the iterator interface.</td>
</tr>
</tbody>
</table>

25.1.1.2 The Iterator Interface

The Iterator interface includes the following properties:
next  A function that returns an object.
The function returns an object that conforms to the `IteratorResult` interface. If a previous call to the `next` method of an `Iterator` has returned an `IteratorResult` object whose `done` property is `true`, then all subsequent calls to the `next` method of that object must also return an `IteratorResult` object whose `done` property is `true`.

NOTE  Arguments may be passed to the `next` function but their interpretation and validity is dependent upon the target `Iterator`. The `for-of` statement and other common users of `Iterators` do not pass any arguments, so `Iterators` that expect to be used in such a manner must be prepared to deal with being called with no arguments.

### 25.1.1.3 The `IteratorResult` Interface

The `IteratorResult` interface includes the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>done</td>
<td>Either <code>true</code> or <code>false</code>.</td>
<td>This is the result status of an <code>iterator</code> <code>next</code> method call. If the end of the <code>iterator</code> was reached, <code>done</code> is <code>true</code>. If the end was not reached, <code>done</code> is <code>false</code> and a value is available. If a <code>done</code> property (either own or inherited does not exist), it is considered to have the value <code>false</code>.</td>
</tr>
<tr>
<td>value</td>
<td>Any ECMAScript language value.</td>
<td>If <code>done</code> is <code>false</code>, this is the current iteration element value. If <code>done</code> is <code>true</code>, this is the return value of the <code>iterator</code>, if it supplied one. If the <code>iterator</code> does not have a return value, <code>value</code> is <code>undefined</code>. In that case, the <code>value</code> property may be absent from the conforming object if it does not inherit an explicit <code>value</code> property.</td>
</tr>
</tbody>
</table>

### 25.1.2 The `%IteratorPrototype% Object

The value of the `[[Prototype]]` internal slot of the `%IteratorPrototype% object is the standard built-in `Object.prototype` object (19.1.3). The `%IteratorPrototype% object is an ordinary object.

NOTE  All objects defined in this specification that implement the `Iterator` interface also inherit from `%IteratorPrototype%`. ECMAScript code may also define objects that inherit from `%IteratorPrototype%`. The `%IteratorPrototype% object provides a place where additional methods that are applicable to all `Iterator` objects may be added.

The following expression is one way that ECMAScript code can access the `%IteratorPrototype% object:

`Object.getPrototypeOf(Object.getPrototypeOf([[[Symbol.iterator()]]]))`

### 25.2 GeneratorFunction Objects

`GeneratorFunction` 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.
25.2.1 The GeneratorFunction Constructor

The GeneratorFunction constructor is the %GeneratorFunction% intrinsic. When GeneratorFunction is called as a function rather than as a constructor, it creates and initializes 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 [[ECMAScriptCode]] internal slot whose value is undefined, it initializes the this value using the argument values. This permits GeneratorFunction to be used both as factory method and to perform constructor instance initialization.
GeneratorFunction may be subclassed and subclass constructors may perform a super invocation of the GeneratorFunction constructor to initialize 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.
7. Let `bodyText` be `ToString(bodyText)`.
8. ReturnIfAbrupt(`bodyText`).
9. Let `parameters` be the result of parsing `P`, interpreted as UTF-16 encoded Unicode text as described in 6.1.4, using `FormalParameters` as the goal symbol. Throw a SyntaxError exception if the parse fails.
10. Let `funcBody` be the result of parsing `bodyText`, interpreted as UTF-16 encoded Unicode text as described in 6.1.4, using `FunctionBody [ Yield ]` as the goal symbol. Throw a SyntaxError exception if the parse fails or if any static semantics errors are detected.
11. If `bodyText` is strict mode code (see 10.2.1) then let `strict` be true, else let `strict` be false.
12. Let `scope` be the Global Environment.
13. Let `F` be the this value.
14. If Type(`F`) is not Object or if `F` does not have a [[ECMAScriptCode]] internal slot or if the value of `[[ECMAScriptCode]]` is not undefined, then
   a. Let `C` be the active function object.
   b. Let `proto` be the result of GetPrototypeFromConstructor(`C`, "Generator")
   c. ReturnIfAbrupt(`proto`).
   d. Let `F` be FunctionAllocate(`proto`, `strict`, "generator").
   e. ReturnIfAbrupt(`F`).
15. If the value of F’s [[FunctionKind]] internal slot is not “generator”, then throw a TypeError exception.
16. Let isExtensible be IsExtensible(F).
17. ReturnIfAbrupt(isExtensible).
18. If isExtensible is false, then throw a TypeError exception.
20. Let status be FunctionInitialize(F, Normal, strict, parameters, body, scope).
21. Let prototype ObjectCreate(%GeneratorPrototype%).
22. If ReferencesSuper(funcBody) is true or ReferencesSuper(parameters) is true, then
   a. Perform MakeMethod(F, undefined, undefined).
23. Let status be the result of the abstract operation MakeConstructor with arguments F, true, and prototype.
24. ReturnIfAbrupt(status).
25. Let hasName be HasOwnProperty(F, "name").
26. ReturnIfAbrupt(hasName).
27. If hasName is false, then
   a. Let status be SetFunctionName(F, "anonymous").
   b. ReturnIfAbrupt(status).
28. 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 initializes 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 Construct (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 value of the name property of the GeneratorFunction is “GeneratorFunction”.

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. Return FunctionAllocate(proto, false, "generator").

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 constructor 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 [[ECMAScriptCode]] internal slot or any other of the internal slots listed in Table 26 or Table 48. 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 %GeneratorPrototype% 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 26. 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 initialize 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 %IteratorPrototype% (25.1.2). 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.return ( value )

The return method performs the following steps:
1. Let g be the this value.
2. Let C be Completion{[[type]]: return, [[value]]: value, [[target]]: empty}.
3. Return GeneratorResumeAbrupt(g, C).

25.3.1.4 Generator.prototype.throw ( exception )

The throw method performs the following steps:
1. Let g be the this value.
2. Let C be Completion{[[type]]: throw, [[value]]: exception, [[target]]: empty}.
3. Return GeneratorResumeAbrupt(g, C).
25.3.1.5 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]".

25.3.1.6 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 48.

<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 generator.</td>
</tr>
</tbody>
</table>

25.3.3 Generator Abstract Operations

25.3.3.1 GeneratorStart (generator, generatorBody)

The abstract operation GeneratorStart with arguments generator and generatorBody performs the following steps:

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 discard 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 GeneratorValidate ( generator )

The abstract operation GeneratorValidate with argument generator 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. Assert: generator also has a [[GeneratorContext]] internal slot.
4. Let state be the value of generator’s [[GeneratorState]] internal slot.
5. If state is undefined or state is "executing" then, throw a TypeError exception.
6. Return state.

25.3.3.3 GeneratorResume ( generator, value )

The abstract operation GeneratorResume with arguments generator and value performs the following steps:

1. Let state be GeneratorValidate(generator).
2. ReturnIfAbrupt(state).
3. If state is "completed", then return CreateIterResultObject(undefined, true).
4. Assert: state is either "suspendedStart" or "suspendedYield".
5. Let genContext be the value of generator’s [[GeneratorContext]] internal slot.
6. Let methodContext be the running execution context.
7. Suspends methodContext.
8. Set generator’s [[GeneratorState]] internal slot to "executing".
9. Push genContext onto the execution context stack; genContext is now the running execution context.
10. 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.
11. Assert: When we return here, genContext has already been removed from the execution context stack and methodContext is the currently running execution context.
12. Return result.

25.3.3.4 GeneratorResumeAbrupt(generator, abruptCompletion)

The abstract operation GeneratorResumeAbrupt with arguments generator and abruptCompletion performs the following steps:

1. Let state be GeneratorValidate(generator).
2. ReturnIfAbrupt(state).
3. 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.
4. If state is "completed", then
   a. If abruptCompletion.[[type]] is return then,
     i. Return CreateIterResultObject(abruptCompletion.[[value]], true).
   b. Return abruptCompletion.
5. Assert: state is "suspendedYield".
6. Let genContext be the value of generator’s [[GeneratorContext]] internal slot.
7. Let `methodContext` be the running execution context.
9. Set generator’s `[[GeneratorState]]` internal slot to “executing”.
10. Push `genContext` onto the execution context stack; `genContext` is now the running execution context.
11. Resume the suspended evaluation of `genContext` using `abruptCompletion` as the result of the operation that suspended it. Let `result` be the value returned by the resumed computation.
12. Assert: When we return here, `genContext` has already been removed from the execution context stack and `methodContext` is currently running execution context.
13. Return `result`.

25.3.3.5 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. Return `NormalCompletion(iterNextObj)`.
9. NOTE: This returns to the evaluation of the operation that had most previously resumed evaluation of `genContext`.

25.4 Promise Objects

A Promise is an object that is used as a placeholder for the eventual results of a deferred (and possibly asynchronous) computation.

Any Promise object is in one of three mutually exclusive states: `fulfilled`, `rejected`, and `pending`:

- A promise `p` is `fulfilled` if `p.then(\ell, z)` will immediately enqueue a Job to call the function `\ell`.
- A promise `p` is `rejected` if `p.then(\ell, z)` will immediately enqueue a Job to call the function `z`.
- A promise is `pending` if it is neither `fulfilled` nor `rejected`.

A promise is said to be `settled` if it is not pending, i.e. if it is either `fulfilled` or `rejected`.

A promise is `resolved` if it is `settled` or if it has been “locked in” to match the state of another promise. Attempting to `resolve` or `reject` a resolved promise has no effect. A promise is `unresolved` if it is not resolved. An unresolved promise is always in the pending state. A resolved promise may be `pending`, `fulfilled` or `rejected`.

© Ecma International 2014
25.4.1 Promise Abstract Operations

25.4.1.1 PromiseCapability Records

A PromiseCapability is a Record value used to encapsulate a promise object along with the functions that are capable of resolving or rejecting that promise object. PromiseCapability records are produced by the NewPromiseCapability abstract operation.

PromiseCapability Records have the fields listed in Table 49.

Table 49 — PromiseCapability Record Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Promise]]</td>
<td>An object</td>
<td>An object that is usable as a promise.</td>
</tr>
<tr>
<td>[[Resolve]]</td>
<td>A function object</td>
<td>The function that is used to resolve the given promise object.</td>
</tr>
<tr>
<td>[[Reject]]</td>
<td>A function object</td>
<td>The function that is used to reject the given promise object.</td>
</tr>
</tbody>
</table>

25.4.1.1.1 IfAbruptRejectPromise (value, capability)

IfAbruptRejectPromise is a short hand for a sequence of algorithm steps that use a PromiseCapability record. An algorithm step of the form:

1. IfAbruptRejectPromise(value, capability).

means the same thing as:

1. If value is an abrupt completion,
   a. Let rejectResult be the result of calling the [[Call]] internal method of capability. [[Reject]] with undefined as thisArgument and (value. [[value]]) as argumentsList.
   b. ReturnIfAbrupt(rejectResult).
   c. Return capability. [[Promise]].
2. Else if value is a Completion Record, then let value be value. [[value]].

25.4.1.2 PromiseReaction Records

The PromiseReaction is a Record value used to store information about how a promise should react when it becomes resolved or rejected with a given value. PromiseReaction records are created by the then method of the Promise prototype, and are used by a PromiseReactionJob.

PromiseReaction records have the fields listed in Table 50.
### Table 50 — PromiseReaction Record Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[Capabilities]]</code></td>
<td>A PromiseCapability record</td>
<td>The capabilities of the promise for which this record provides a reaction handler.</td>
</tr>
<tr>
<td><code>[[Handler]]</code></td>
<td>A function object or a String</td>
<td>The function that should be applied to the incoming value, and whose return value will govern what happens to the derived promise. If <code>[[Handler]]</code> is &quot;Identity&quot; it is equivalent to a function that simply returns its first argument. If <code>[[Handler]]</code> is &quot;Thrower&quot; it is equivalent to a function that throws its first argument as an exception.</td>
</tr>
</tbody>
</table>

#### 25.4.1.3 CreateResolvingFunctions ( promise )

When `CreateResolvingFunctions` is performed with argument `promise`, the following steps are taken:

1. Let `alreadyResolved` be a new Record (`[[value]]: false`).
2. Let `resolve` be a new built-in function object as defined in Promise Resolve Functions (25.4.1.3.2).
3. Set the `[[Promise]]` internal slot of `resolve` to `promise`.
4. Set the `[[AlreadyResolved]]` internal slot of `resolve` to `alreadyResolved`.
5. Let `reject` be a new built-in function object as defined in Promise Reject Functions (25.4.1.3.1).
6. Set the `[[Promise]]` internal slot of `reject` to `promise`.
7. Set the `[[AlreadyResolved]]` internal slot of `reject` to `alreadyResolved`.
8. Return a new Record (`[[Resolve]]: resolve, [[Reject]]: reject`).

#### 25.4.1.3.1 Promise Reject Functions

A promise reject function is an anonymous built-in function that has `[[Promise]]` and `[[AlreadyResolved]]` internal slots.

When a promise reject function `F` is called with argument `reason`, the following steps are taken:

1. Assert: `F` has a `[[Promise]]` internal slot whose value is an Object.
2. Let `promise` be the value of `F`'s `[[Promise]]` internal slot.
3. Let `alreadyResolved` be the value of `F`'s `[[AlreadyResolved]]` internal slot.
4. If `alreadyResolved.[[value]]` is `true`, then return `undefined`.
5. Set `alreadyResolved.[[value]]` to `true`.
6. Return `RejectPromise(promise, reason)`. 

#### 25.4.1.3.2 Promise Resolve Functions

A promise resolve function is an anonymous built-in function that has `[[Promise]]` and `[[AlreadyResolved]]` internal slots.

When a promise resolve function `F` is called with argument `resolution`, the following steps are taken:

1. Assert: `F` has a `[[Promise]]` internal slot whose value is an Object.
2. Let `promise` be the value of `F`'s `[[Promise]]` internal slot.
3. Let `alreadyResolved` be the value of `F`'s `[[AlreadyResolved]]` internal slot.
4. If `alreadyResolved.[[value]]` is `true`, then return `undefined`.
5. Set `alreadyResolved.[[value]]` to `true`. 

© Ecma International 2014
6. If SameValue(resolution, promise) is true, then
   a. Let selfResolutionError be a newly-created TypeError object.
   b. Return RejectPromise(promise, selfResolutionError).
7. If Type(resolution) is not Object, then
   a. Return FulfillPromise(promise, resolution).
8. Let then be Get(resolution, "then").
9. If then is an abrupt completion, then
   a. Return RejectPromise(promise, then.[[value]]). 
10. Let then be then.[[value]].
11. If IsCallable(then) is false, then
   a. Return FulfillPromise(promise, resolution).
12. Perform EnqueueJob ("PromiseJobs", PromiseResolveThenableJob, (promise, resolution, then))
13. Return undefined.

25.4.1.4 FulfillPromise (promise, value)

When the FulfillPromise abstract operation is called with arguments promise and value the following steps are taken:

1. Assert: the value of promise's [[PromiseState]] internal slot is "pending".
2. Let reactions be the value of promise's [[PromiseFulfillReactions]] internal slot.
3. Set the value of promise's [[PromiseResult]] internal slot to value.
4. Set the value of promise's [[PromiseFulfillReactions]] internal slot to undefined.
5. Set the value of promise's [[PromiseRejectReactions]] internal slot to undefined.
6. Set the value of promise's [[PromiseState]] internal slot to "fulfilled".
7. Return TriggerPromiseReactions(reactions, value).

25.4.1.5 NewPromiseCapability (C)

The abstract operation NewPromiseCapability takes a constructor function, and attempts to use that constructor function in the fashion of the built-in Promise constructor to create a Promise object and extract its resolve and reject functions. The promise plus the resolve and reject functions are used to initialize a new PromiseCapability record which is returned as the value of this abstract operation.

1. If IsConstructor(C) is false, throw a TypeError exception.
2. Assert: C is a constructor function that supports the parameter conventions of the Promise constructor (see 25.4.3.1).
3. Let promise be CreateFromConstructor(C).
4. ReturnIfAbrupt(promise).
5. If Type(promise) is not Object, then throw a TypeError exception.
6. Return CreatePromiseCapabilityRecord(promise, C).

NOTE This abstract operation supports Promise subclassing, as it is generic on any constructor that calls a passed executor function argument in the same way as the Promise constructor. It is used to generalize static methods of the Promise constructor to any subclass.

25.4.1.5.1 CreatePromiseCapabilityRecord( promise, constructor )

When the CreatePromiseCapabilityRecord abstract operation is called with arguments promise and constructor the following steps are taken:

1. Assert: promise is an uninitialized object created as if by invoking @@create on constructor.
2. Assert: IsConstructor(constructor) is true.
3. Let promiseCapability be a new PromiseCapability { [[Promise]]: promise, [[Resolve]]: undefined, [[Reject]]: undefined }.
4. Let executor be a new built-in function object as defined in GetCapabilitiesExecutor Functions (25.4.1.5.1).
5. Set the [[Capability]] internal slot of executor to promiseCapability.
6. Let constructorResult be the result of calling the [[Call]] internal method of constructor, passing promise and (executor) as the arguments.
7. ReturnIfAbrupt(constructorResult).
8. If IsCallable(promiseCapability. [[Resolve]]) is false, then throw a TypeError exception.
9. If IsCallable(promiseCapability. [[Reject]]) is false, then throw a TypeError exception.
10. If Type(constructorResult) is Object and SameValue(promise, constructorResult) is false, then throw a TypeError exception.

25.4.1.5.2 GetCapabilitiesExecutor Functions

A GetCapabilitiesExecutor function is an anonymous built-in function that has a [[Capability]] internal slot.

When a GetCapabilitiesExecutor function F is called with arguments resolve and reject the following steps are taken:
1. Assert: F has a [[Capability]] internal slot whose value is a PromiseCapability Record.
2. Let promiseCapability be the value of F's [[Capability]] internal slot.
3. If promiseCapability. [[Resolve]] is not undefined, then throw a TypeError exception.
4. If promiseCapability. [[Reject]] is not undefined, then throw a TypeError exception.
5. Set promiseCapability. [[Resolve]] to resolve.
6. Set promiseCapability. [[Reject]] to reject.
7. Return undefined.

25.4.1.6 IsPromise ( x )

The abstract operation IsPromise checks for the promise brand on an object.
1. If Type(x) is not Object, return false.
2. If x does not have a [[PromiseState]] internal slot, return false.
3. If the value of x's [[PromiseState]] internal slot is undefined, return false.
4. Return true.

25.4.1.7 RejectPromise ( promise, reason )

When the RejectPromise abstract operation is called with arguments promise and reason the following steps are taken:
1. Assert: the value of promise's [[PromiseState]] internal slot is "pending".
2. Let reactions be the value of promise's [[PromiseRejectReactions]] internal slot.
3. Set the value of promise's [[PromiseResult]] internal slot to reason.
4. Set the value of promise's [[PromiseRejectReactions]] internal slot to undefined.
5. Set the value of promise's [[PromiseRejectReactions]] internal slot to undefined.
6. Set the value of promise's [[PromiseState]] internal slot to "rejected".
7. Return TriggerPromiseReactions(reactions, reason).
25.4.1.8 TriggerPromiseReactions (reactions, argument)

The abstract operation TriggerPromiseReactions takes a collection of functions to trigger in the next Job, and calls them, passing each the given argument. Typically, these reactions will modify a previously-returned promise, possibly calling in to a user-supplied handler before doing so.

1. Repeat for each reaction in reactions, in original insertion order
   a. Perform EnqueueJob("PromiseJobs", PromiseReactionJob, (reaction, argument)).
2. Return undefined.

25.4.2 Promise Jobs

25.4.2.1 PromiseReactionJob (reaction, argument)

The job PromiseReactionJob with parameters reaction and argument applies the appropriate handler to the incoming value, and uses the handler's return value to resolve or reject the derived promise associated with that handler.

1. Assert: reaction is a PromiseReaction Record.
2. Let promiseCapability be reaction.[[Capabilities]].
3. Let handler be reaction.[[Handler]].
4. If handler is "Identity", then let handlerResult be NormalCompletion(argument).
5. Else if handler is "Thrower", then let handlerResult be Completion{[[type]$: throw, [[value]$: argument], [[target]$: empty]}
6. Else, let handlerResult be the result of calling the [[Call]] internal method of handler passing undefined as thisArgument and (argument) as argumentsList.
7. If handlerResult is an abrupt completion, then
   a. Let status be the result of calling the [[Call]] internal method of promiseCapability.[[Reject]] passing undefined as thisArgument and (handlerResult.[[value]]) as argumentsList.
   b. NextJob status.
8. Let handlerResult be handlerResult.[[value]].
9. Let status be the result of calling the [[Call]] internal method of promiseCapability.[[Resolve]] passing undefined as thisArgument and (handlerResult) as argumentsList.

25.4.2.2 PromiseResolveThenableJob (promiseToResolve, thenable, then)

The job PromiseResolveThenableJob with parameters promiseToResolve, thenable, and then performs the following steps:

1. Let resolvingFunctions be CreateResolvingFunctions(promise).
2. Let thenCallResult be the result of calling the [[Call]] internal method of then passing thenable as thisArgument and (resolvingFunctions.[[Resolve]], resolvingFunctions.[[Reject]]) as argumentsList.
3. If thenCallResult is an abrupt completion, then
   a. Let status be the result of calling the [[Call]] internal method of resolvingFunctions.[[Reject]] passing undefined as thisArgument and (thenCallResult.[[value]]) as argumentsList.
   b. NextJob status.
4. NextJob thenCallResult.

NOTE This Job uses the supplied thenable and its then method to resolve the given promise. This process must take place as a Job to ensure that the evaluation of the then method occurs after evaluation of any surrounding code has completed.
25.4.3 The Promise Constructor

The Promise constructor is the %Promise% intrinsic object and the initial value of the Promise property of the global object. When Promise is called as a function rather than as a constructor, it initializes its this value with the internal state necessary to support the Promise.prototype methods.

The Promise 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 Promise behaviour must include a super call to Promise.

25.4.3.1 Promise (executor)

When the Promise function is called with argument executor the following steps are taken:

1. Let promise be the this value.
2. If Type(promise) is not Object, then throw a TypeError exception.
3. If promise does not have a [[PromiseState]] internal slot, then throw a TypeError exception.
4. If promise's [[PromiseState]] internal slot is not undefined, then throw a TypeError exception.
5. If IsCallable(executor) is false, then throw a TypeError exception.
6. Return InitializePromise(promise, executor).

NOTE The executor argument must be a function object. It is called for initiating and reporting completion of the possibly deferred action represented by this Promise object. The executor is called with two arguments: resolve and reject. These are functions that may be used by the executor function to report eventual completion or failure of the deferred computation. Returning from the executor function does not mean that the deferred action has been completed but only that the request to eventually perform the deferred action has been accepted.

The resolve function that is passed to an executor function accepts a single argument. The executor code may eventually call the resolve function to indicate that it wishes to resolve the associated Promise object. The argument passed to the resolve function represents the eventual value of the deferred action and can be either the actual fulfillment value or another Promise object which will provide the value if it is fulfilled.

The reject function that is passed to an executor function accepts a single argument. The executor code may eventually call the reject function to indicate that the associated Promise is rejected and will never be fulfilled. The argument passed to the reject function is used as the rejection value of the promise. Typically it will be an Error object.

The resolve and reject functions passed to an executor function by the Promise constructor have the capability to actually resolve and reject the associated promise. Subclasses may have different constructor behaviour that passes in customized values for resolve and reject.

25.4.3.1.1 InitializePromise (promise, executor)

The abstract operation InitializePromise initializes a newly allocated promise object using an executor function.

1. Assert: promise has a [[PromiseState]] internal slot and its value is undefined.
2. Assert: IsCallable(executor) is true.
3. Set promise's [[PromiseState]] internal slot to "pending".
4. Set promise's [[PromiseFulfillReactions]] internal slot to a new empty List.
5. Set promise's [[PromiseRejectReactions]] internal slot to a new empty List.
6. Let resolvingFunctions be CreateResolvingFunctions(promise).
7. Let completion be the result of calling the [[Call]] internal method of executor with undefined as thisArgument and (resolvingFunctions.[[Resolve]], resolvingFunctions.[[Reject]]) as argumentsList.
8. If completion is an abrupt completion, then
   a. Let status be the result of calling the [[Call]] internal method of resolvingFunctions.[[Reject]]
      with undefined as thisArgument and (completion.[[value]]) as argumentsList.
   b. ReturnIfAbrupt(status).
9. Return promise.

25.4.3.2 new Promise ( ... argumentsList )

When Promise is called as part of a new expression it is a constructor; it initializes a newly created object.

Promise called as part of a new expression with argument list argumentsList performs the following steps:
   1. Let F be the Promise 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 Construct(F, argumentsList).

If Promise is implemented as an ECMAScript function object, its [[Construct]] internal method will perform
the above steps.

25.4.4 Properties of the Promise Constructor

The value of the [[Prototype]] internal slot of the Promise constructor is the Function prototype object
(19.2.3).

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

25.4.4.1 Promise.all ( iterable )

The all function returns a new promise which is fulfilled with an array of fulfillment values for the passed
promises, or rejects with the reason of the first passed promise that rejects. It resoves all elements of the
passed iterable to promises as it runs this algorithm:

   1. Let C be the this value.
   2. Let promiseCapability be NewPromiseCapability(C).
   3. ReturnIfAbrupt(promiseCapability).
   4. Let iterator be GetIterator(iterable).
   5. IfAbruptRejectPromise(iterator, promiseCapability).
   6. Let values be ArrayCreate(0).
   7. Let remainingElementsCount be a new Record { [[value]]: 1 }.
   8. Let index be 0.
   9. Repeat
      a. Let next be IteratorStep(iterator).
      b. IfAbruptRejectPromise(next, promiseCapability).
      c. If next is false,
         i. Set remainingElementsCount.[[value]] to remainingElementsCount.[[value]] - 1.
         ii. If remainingElementsCount.[[value]] is 0,
             1. Let resolveResult be the result of calling the [[Call]] internal method of
                promiseCapability.[[Resolve]] with undefined as thisArgument and (values) as
                argumentsList.
             2. ReturnIfAbrupt(resolveResult).
iii. Return promiseCapability.[[Promise]].
.d. Let nextValue be IteratorValue(next).
e. IfAbruptRejectPromise(nextValue, promiseCapability).
f. Let nextPromise be Invoke(C, "resolve", (nextValue)).
g. IfAbruptRejectPromise(nextPromise, promiseCapability).
h. Let resolveElement be a new built-in function object as defined in Promise.all Resolve Element Functions.
i. Set the [[AlreadyCalled]] internal slot of resolveElement to a new Record ( [[value]]: false ).
j. Set the [[Index]] internal slot of resolveElement to index.
k. Set the [[Values]] internal slot of resolveElement to values.
l. Set the [[Capabilities]] internal slot of resolveElement to promiseCapability.
m. Set the [[RemainingElements]] internal slot of resolveElement to remainingElementsCount.
.n. Set remainingElementsCount.[[value]] to remainingElementsCount.[[value]] + 1.
o. Let result be Invoke(nextPromise, "then", (resolveElement, promiseCapability.[[Reject]])).
p. IfAbruptRejectPromise(result, promiseCapability).
q. Set index to index + 1.

Note: The all function requires its this value to be a constructor function that supports the parameter conventions of the Promise constructor.

25.4.4.1.1 Promise.all Resolve Element Functions

A Promise.all resolve element function is an anonymous built-in function that is used to resolve a specific Promise.all element. Each Promise.all resolve element function has [[Index]], [[Values]], [[Capabilities]], [[RemainingElements]], and [[AlreadyCalled]] internal slots.

When a Promise.all resolve element function F is called with argument x, the following steps are taken:

1. Let alreadyCalled be the value of F's [[AlreadyCalled]] internal slot.
2. If alreadyCalled.[[value]] is true, then return undefined.
3. Set alreadyCalled.[[value]] to true.
4. Let index be the value of F's [[Index]] internal slot.
5. Let values be the value of F's [[Values]] internal slot.
6. Let promiseCapability be the value of F's [[Capabilities]] internal slot.
7. Let remainingElementsCount be the value of F's [[RemainingElements]] internal slot.
8. Let result be CreateDataProperty(values, ToString(index), x).
9. IfAbruptRejectPromise(result, promiseCapability).
10. Set remainingElementsCount.[[value]] to remainingElementsCount.[[value]] - 1.
11. If remainingElementsCount.[[value]] is 0,
   a. Return the result of calling the [[Call]] internal method of promiseCapability.[[Resolve]] with undefined as thisArgument and (values) as argumentsList.
12. Return undefined.

25.4.4.2 Promise.prototype

The initial value of Promise.prototype is the Promise prototype object (25.4.4.6.1).

This property has the attributes ( [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false ).
25.4.4.3 Promise.race ( iterable )

The `race` function returns a new promise which is settled in the same way as the first passed promise to settle. It resolves all elements of the passed iterable to promises as it runs this algorithm.

1. Let C be the this value.
2. Let promiseCapability be NewPromiseCapability(C).
3. ReturnIfAbrupt(promiseCapability).
4. Let iterator be GetIterator(iterable).
5. IfAbruptRejectPromise(iterator, promiseCapability).
6. Repeat
   a. Let next be IteratorStep(iterator).
   b. IfAbruptRejectPromise(next, promiseCapability).
   c. If next is false, return promiseCapability.[[Promise]].
   d. Let nextValue be IteratorValue(next).
   e. IfAbruptRejectPromise(nextValue, promiseCapability).
   f. Let nextPromise be Invoke(C, "resolve", (nextValue)).
   g. IfAbruptRejectPromise(nextPromise, promiseCapability).
   h. Let result be Invoke(nextPromise, "then", (promiseCapability.[[Resolve]], promiseCapability.[[Reject]])).
   i. IfAbruptRejectPromise(result, promiseCapability).

NOTE 1 If the iterable argument is empty or if none of the promises in iterable ever settle then the pending promise returned by this method will never be settled.

NOTE 2 The `race` function expects its this value to be a constructor function that supports the parameter conventions of the Promise constructor. It also expects that its this value provides a resolve method.

25.4.4.4 Promise.reject ( r )

The `reject` function returns a new promise rejected with the passed argument.

1. Let C be the this value.
2. Let promiseCapability be NewPromiseCapability(C).
3. ReturnIfAbrupt(promiseCapability).
4. Let rejectResult be the result of calling the [[Call]] internal method of promiseCapability.[[Reject]] with undefined as thisArgument and (r) as argumentsList.
5. ReturnIfAbrupt(rejectResult).
6. Return promiseCapability.[[Promise]].

NOTE The `reject` function requires that its this value to be a constructor function that supports the parameter conventions of the Promise constructor.

25.4.4.5 Promise.resolve ( x )

The `resolve` function returns either a new promise resolved with the passed argument, or the argument itself if the argument a promise produced by this constructor.

1. Let C be the this value.
2. If IsPromise(x) is true.
   a. Let constructor be the value of x’s [[PromiseConstructor]] internal slot.
   b. If SameValue(constructor, C) is true, return x.
3. Let promiseCapability be NewPromiseCapability(C).
4. ReturnIfAbrupt(promiseCapability).
5. Let resolveResult be the result of calling the [[Call]] internal method of promiseCapability.\[Resolve\] with \texttt{undefined} as thisArgument and (x) as argumentsList.
6. ReturnIfAbrupt(resolveResult).
7. Return promiseCapability.\[Promise\].

**NOTE** The \texttt{resolve} function requires that its this value to be a constructor function that supports the parameter conventions of the Promise constructor.

### 25.4.4.6 Promise [ @@create ] ()

The @@create method of a Promise function object \texttt{F} performs the following steps:
1. Let \texttt{F} be the this value.
2. Return AllocatePromise(\texttt{F}).

The value of the name property of this function is "\texttt{[Symbol.create]}".

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

#### 25.4.4.6.1 AllocatePromise ( constructor )

The abstract operation AllocatePromise allocates a new promise object using the constructor argument.
1. Let \texttt{obj} be OrdinaryCreateFromConstructor(\texttt{constructor}, "\texttt{PromisePrototype}\texttt{"}, [[PromiseState]], [[PromiseConstructor]], [[PromiseResult]], [[PromiseFulfillReactions]], [[PromiseRejectReactions]])
2. Set the value of \texttt{obj}'s [[PromiseConstructor]] internal slot to \texttt{constructor}.
3. Return \texttt{obj}.

### 25.4.5 Properties of the Promise Prototype Object

The value of the [[Prototype]] internal slot of the Promise prototype object is the standard built-in Object prototype object (19.1.3). The Promise prototype object is an ordinary object. It does not have a [[PromiseState]] internal slot or any of the other internal slots of Promise instances.

#### 25.4.5.1 Promise.prototype.catch ( onRejected )

When the catch method is called with argument \texttt{onRejected} the following steps are taken:
1. Let promise be the this value.
2. Return Invoke(promise, "then", (\texttt{undefined}, \texttt{onRejected})).

#### 25.4.5.2 Promise.prototype.constructor

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

#### 25.4.5.3 Promise.prototype.then ( onFulfilled , onRejected )

When the then method is called with arguments \texttt{onFulfilled} and \texttt{onRejected} the following steps are taken:
1. Let promise be the this value.
2. If IsPromise(promise) is \texttt{false}, throw a TypeError exception.
3. If IsCallable(onFulfilled) is \texttt{false}, then
a. Let onFulfilled be "Identity".
4. If IsCallable(onRejected) is false, then
   a. Let onRejected be "Thrower".
5. Let C be Get(promise, "constructor").
6. ReturnIfAbrupt(C).
7. Let promiseCapability be NewPromiseCapability(C).
8. ReturnIfAbrupt(promiseCapability).
9. Let fulfillReaction be the PromiseReaction { [[Capabilities]]: promiseCapability, [[Handler]]: onFulfilled }.
10. Let rejectReaction be the PromiseReaction { [[Capabilities]]: promiseCapability, [[Handler]]: onRejected }.
11. If the value of promise's [[PromiseState]] internal slot is "pending",
   a. Append fulfillReaction as the last element of the List that is the value of promise's [[PromiseFulfillReactions]] internal slot.
   b. Append rejectReaction as the last element of the List that is the value of promise's [[PromiseRejectReactions]] internal slot.
12. Else if the value of promise's [[PromiseState]] internal slot is "fulfilled",
   a. Let value be the value of promise's [[PromiseResult]] internal slot.
   b. Perform EnqueueJob("PromiseJobs", PromiseReactionJob, (fulfillReaction, value)).
13. Else if the value of promise's [[PromiseState]] internal slot is "rejected",
   a. Let reason be the value of promise's [[PromiseResult]] internal slot.
   b. Perform EnqueueJob("PromiseJobs", PromiseReactionJob, (rejectReaction, reason)).
14. Return promiseCapability.}

25.4.5.4 Promise.prototype[@@toStringTag ]

The initial value of the @@toStringTag property is the string value "Promise".

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

25.4.6 Properties of Promise Instances

Promise instances are ordinary objects that inherit properties from the Promise prototype object (the intrinsic, %PromisePrototype%). Promise instances are initially created with the internal slots described in Table 51.
Table 51 — Internal Slots of Promise Instances

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![PromiseState]</td>
<td>A string value that governs how a promise will react to incoming calls to its then method. The possible values are: undefined, &quot;pending&quot;, &quot;fulfilled&quot;, and &quot;rejected&quot;.</td>
</tr>
<tr>
<td>![PromiseConstructor]</td>
<td>The function object that was used to construct this promise. Checked by the resolve method of the Promise constructor.</td>
</tr>
<tr>
<td>![PromiseResult]</td>
<td>The value with which the promise has been fulfilled or rejected, if any. Only meaningful if ![PromiseState] is not &quot;pending&quot;.</td>
</tr>
<tr>
<td>![PromiseFulfillReactions]</td>
<td>A List of PromiseReaction records to be processed when/if the promise transitions from the &quot;pending&quot; state to the &quot;fulfilled&quot; state.</td>
</tr>
<tr>
<td>![PromiseRejectReactions]</td>
<td>A List of PromiseReaction records to be processed when/if the promise transitions from the &quot;pending&quot; state to the &quot;rejected&quot; state.</td>
</tr>
</tbody>
</table>

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.apply ( target, thisArgument, argumentsList )

When the apply function is called with arguments target, thisArgument, and argumentsList the following steps are taken:

1. If IsCallable(target) is false, then throw a TypeError exception.
2. Let args be CreateListFromArrayLike(argumentsList).
3. ReturnIfAbrupt(args).
4. Perform the PrepareForTailCall abstract operation.
5. Return the result of calling the ![Call] internal method of target with arguments thisArgument and args.

26.1.2 Reflect.construct ( target, argumentsList )

When the construct function is called with arguments target and argumentsList the following steps are taken:

1. If IsConstructor(target) is false, then throw a TypeError exception.
2. Let \( args \) be \( \text{CreateListFromArrayLike}(\text{argumentsList}) \).
3. ReturnIfAbrupt(\( args \)).
4. Return the result of calling the \([\text{Construct}]\) internal method of \( target \) with argument \( args \).

### 26.1.3 Reflect.defineProperty (\( target, propertyKey, attributes \))

When the \texttt{defineProperty} function is called with arguments \( target, propertyKey, \) and \( attributes \) the following steps are taken:

1. Let \( obj \) be \( \text{ToObject}(target) \).
2. ReturnIfAbrupt(\( obj \)).
3. Let \( key \) be \( \text{ToPropertyKey}(propertyKey) \).
4. ReturnIfAbrupt(\( key \)).
5. Let \( desc \) be the result of calling \( \text{ToPropertyDescriptor} \) with \( attributes \) as the argument.
6. ReturnIfAbrupt(\( desc \)).
7. Return the result of calling the \([\text{DefineOwnProperty}]\) internal method of \( obj \) with arguments \( key \), and \( desc \).

### 26.1.4 Reflect.deleteProperty (\( target, propertyKey \))

When the \texttt{deleteProperty} function is called with arguments \( target \) and \( propertyKey \), the following steps are taken:

1. Let \( obj \) be \( \text{ToObject}(target) \).
2. ReturnIfAbrupt(\( obj \)).
3. Let \( key \) be \( \text{ToPropertyKey}(propertyKey) \).
4. ReturnIfAbrupt(\( key \)).
5. Return the result of calling the \([\text{Delete}]\) internal method of \( obj \) with argument \( key \).

### 26.1.5 Reflect.enumerate (\( target \))

When the \texttt{enumerate} function is called with argument \( target \) the following steps are taken:

1. Let \( obj \) be \( \text{ToObject}(target) \).
2. ReturnIfAbrupt(\( obj \)).
3. Let \( iterator \) be the result of calling \( \text{ToEnumerate} \) internal method of \( obj \).
4. Return \( iterator \).

### 26.1.6 Reflect.get (\( target, propertyKey [, receiver ] \))

When the \texttt{get} function is called with arguments \( target, propertyKey, \) and \( receiver \) the following steps are taken:

1. Let \( obj \) be \( \text{ToObject}(target) \).
2. ReturnIfAbrupt(\( obj \)).
3. Let \( key \) be \( \text{ToPropertyKey}(propertyKey) \).
4. If \( receiver \) is not present, then
   a. Let \( receiver \) be \( target \).
5. Return the result of calling the \([\text{Get}]\) internal method of \( obj \) with arguments \( key \), and \( receiver \).
26.1.7 Reflect.getOwnPropertyDescriptor (target, propertyKey)

When the `getOwnPropertyDescriptor` function is called with arguments `target` and `propertyKey`, the following steps are taken:

1. Let `obj` be `ToObject(target)`.  
2. ReturnIfAbrupt(`obj`).  
3. Let `key` be `ToPropertyKey(propertyKey)`.  
4. ReturnIfAbrupt(`key`).  
5. 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)`.

26.1.8 Reflect.getPrototypeOf (target)

When the `getPrototypeOf` function is called with argument `target` the following steps are taken:

1. Let `obj` be `ToObject(target)`.  
2. ReturnIfAbrupt(`obj`).  
3. Return the result of calling the `[[GetPrototypeOf]]` internal method of `obj`.

26.1.9 Reflect.has (target, propertyKey)

When the `has` function is called with arguments `target` and `propertyKey`, the following steps are taken:

1. Let `obj` be `ToObject(target)`.  
2. ReturnIfAbrupt(`obj`).  
3. Let `key` be `ToPropertyKey(propertyKey)`.  
4. ReturnIfAbrupt(`key`).  
5. Return the result of calling the `[[HasProperty]]` internal method of `obj` with argument `key`.

26.1.10 Reflect.isExtensible (target)

When the `isExtensible` function is called with argument `target` the following steps are taken:

1. Let `obj` be `ToObject(target)`.  
2. ReturnIfAbrupt(`obj`).  
3. Return the result of calling the `[[IsExtensible]]` internal method of `obj`.

26.1.11 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. Let `keys` be the result of calling the `[[OwnPropertyKeys]]` internal method of `obj`.  
4. ReturnIfAbrupt(`keys`).  
5. Return `CreateArrayFromList(keys)`.

26.1.12 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`.
26.1.13 Reflect.set (target, propertyKey, V [, receiver])

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`.

26.1.14 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 Loader Objects

Loader objects are able to load the source code of an ECMAScript `Module` in the context of a specific Realm.

26.2.1 The Reflect.Loader Constructor

The initialize value of `Reflect.Loader` is the `%Loader% intrinsic object. `Reflect.Loader` is the constructor for Loader objects. When `Reflect.Loader` is called as a function rather than as a constructor, it initializes its `this` value with the internal state necessary to support the `Reflect.Loader.prototype` built-in methods.

The `Reflect.Loader` 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 support the specified Loader behaviour must include a `super` call to `Reflect.Loader`.

26.2.1.1 Reflect.Loader ([options])

When the Reflect.Loader function is called with optional argument `options` the following steps are taken:

1. Let `loader` be the `this` value.
2. If `Type(loader)` is not `Object`, throw a `TypeError` exception.
3. If `loader` does not have a `[[LoaderRecord]]` internal slot, throw a `TypeError` exception.
4. If the value of `loader`'s `[[LoaderRecord]]` internal slot is not `undefined`, throw a `TypeError` exception.
5. Let `realmObject` be the result of `GetOption(options, "realm")`.
6. ReturnIfAbrupt(`realmObject`).
7. If `realmObject` is `undefined`, let `realm` be the Realm of the running execution context.
8. Else,
a. If $\text{Type}(\text{realmObject})$ is not Object or $\text{realmObject}$ does not have a $\text{[[RealmRecord]]}$ internal slot, throw a $\text{TypeError}$ exception.
b. Let $\text{realm}$ be the value of $\text{realmObject}$'s $\text{[[RealmRecord]]}$ internal slot.
c. If $\text{realm}$ is $\text{undefined}$, throw a $\text{TypeError}$ exception.

For each name in the List ("normalize", "locate", "fetch", "translate", "instantiate"),
a. Let $\text{hook}$ be the result of $\text{GetOption}(\text{options}, \text{name})$.
b. $\text{ReturnIfAbrupt}(\text{hook})$.
c. If $\text{hook}$ is not $\text{undefined}$,
i. If $\text{IsCallable}(\text{hook})$ is $\text{false}$, throw a $\text{TypeError}$ exception.
ii. Let $\text{result}$ be $\text{CreateDataPropertyOrThrow}(\text{loader, name, hook})$.
iii. $\text{ReturnIfAbrupt}(\text{result})$.

10. NOTE the following step ensures that this function was not reentrantly applied to $\text{loader}$ during the above steps.
11. If the value of $\text{loader}$'s $\text{[[LoaderRecord]]}$ internal slot is not $\text{undefined}$, throw a $\text{TypeError}$ exception.
12. Let $\text{loaderRecord}$ be $\text{CreateLoaderRecord}(\text{realm, loader})$.
13. Set $\text{loader}$'s $\text{[[LoaderRecord]]}$ internal slot to $\text{loaderRecord}$.
14. Return $\text{loader}$.

26.2.1.2 new $\text{Reflect.Loader}(...\text{argumentsList})$

When $\text{Reflect.Loader}$ is called as part of a $\text{new}$ expression it is a constructor: it initializes a newly created object. It performs the following steps:

1. Let $F$ be the $\text{Reflect.Loader}$ function object on which the $\text{new}$ operator was applied.
2. Let $\text{argumentsList}$ be the $\text{argumentsList}$ argument of the $\text{[[Construct]]}$ internal method that was invoked by the $\text{new}$ operator.
3. Return the result of $\text{Construct}(F, \text{argumentsList})$.

If $\text{Reflect.Loader}$ is implemented as an ECMAScript function object, its $\text{[[Construct]]}$ internal method will perform the above steps.

26.2.2 Properties of the $\text{Loader}$ Constructor

The value of the $\text{[[Prototype]]}$ internal slot of the $\text{Reflect.Loader}$ constructor is the Function prototype object (19.2.3).

Besides the $\text{length}$ property (whose value is 0), the $\text{Reflect.Loader}$ constructor has the following properties:

26.2.2.1 $\text{Reflect.Loader.prototype}$

The initial value of $\text{Reflect.Loader.prototype}$ is the intrinsic %LoaderPrototype% object (26.2.3).

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

26.2.2.2 $\text{Reflect.Loader[@@create]}()$

The $\text{@@create}$ method of a $\text{Reflect.Loader}$ function object $F$ performs the following steps:

1. Let $F$ be the $\text{this}$ value.
2. Let `obj` be the result of calling `OrdinaryCreateFromConstructor(F, "%LoaderPrototype%", ([LoaderRecord]))).
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 }.

26.2.3 Properties of the `Reflect.Loader` Prototype Object

The value of the [[Prototype]] internal slot of the `Reflect.Loader` prototype object is the standard built-in `Object` prototype object (19.1.3). The `Reflect.Loader` prototype object is an ordinary object. It does not have a [[LoaderRecord]] internal slot.

The phrase "this Loader" within the specification of the following methods refers to the result returned by performing the abstract operation thisLoader with the this value of the current method invocation passed as the argument.

The abstract operation thisLoader with argument `value` performs the following steps:

1. If `Type(value)` is `Object` and `value` has a [[LoaderRecord]] internal slot, then
   a. Let `r` be `value`’s [[LoaderRecord]] internal slot.
   b. If `r` is not `undefined`, then return `value`.
2. Throw a `TypeError` exception.

26.2.3.1 `Reflect.Loader.prototype.constructor`

The initial value of `Reflect.Loader.prototype.constructor` is the built-in `%Loader%` constructor.

26.2.3.2 `Reflect.Loader.prototype.define (name, source [, options ])`

The `define` method installs a module in this loader’s module registry for `source` using `name` as the registry key. The module is not immediately available. The `translate` and `instantiate` hooks are called asynchronously, and dependencies are loaded asynchronously. `define` returns a Promise object that resolves to `undefined` when the new module and its dependencies are installed in the registry.

When the `define` method is called with arguments `name`, `source`, and optional argument `options` the following steps are taken:

1. Let `loader` be this Loader.
2. ReturnIfAbrupt(`loader`).
3. Let `loaderRecord` be `loader`’s [[LoaderRecord]] internal slot.
4. Let `name` be `ToString(name)`.
5. ReturnIfAbrupt(`name`).
6. Let `address` be `GetOption(options, "address")`.
7. ReturnIfAbrupt(`address`).
8. Let `metadata` be `GetOption(options, "metadata")`.
9. ReturnIfAbrupt(`metadata`).
10. If `metadata` is `undefined` then let `metadata` be `ObjectCreate(%ObjectPrototype%)`.
11. Let `p` be `PromiseOfStartLoadPartwayThrough("translate", loaderRecord, name, metadata, source, address)
12. ReturnIfAbrupt(`p`).
13. Let \( G \) be a new function as defined by `ReturnUndefined`.
14. Let \( p \) be the result of calling `PromiseThen(p, G)`.
15. Return \( p \).

The `length` property of the `define` method is 2.

### 26.2.3.3 Reflect.Loader.prototype.delete ( name )

The `delete` method removes an entry whose key is `name` from this loader's module registry. It performs the following steps:

1. Let `loader` be this Loader.
2. ReturnIfAbrupt(`loader`).
3. Let `loaderRecord` be `loader`'s `[[LoaderRecord]]` internal slot.
4. Let `name` be `ToString(name)`.
5. ReturnIfAbrupt(`name`).
6. Let `modules` be the value of `loaderRecord.([[Modules]])`.
7. Repeat for each Record `[[name]], [[value]]` \( p \) that is an element of `modules`,
   a. If `SameValue(p.[[key]], name)` is true, then
      i. Set `p.[[key]]` to empty.
      ii. Set `p.[[value]]` to empty.
      iii. Return true.
8. Return false.

### 26.2.3.4 Reflect.Loader.prototype.entries ()

The following steps are taken:

1. Let `loader` be this Loader.
2. ReturnIfAbrupt(`loader`).
3. Return the result of `CreateLoaderIterator(loader, "key+value")`.

### 26.2.3.5 Reflect.Loader.prototype.get ( name )

If this Loader's module registry contains a Module with the given normalized `name`, return it. Otherwise, return `undefined`. If the module is in the registry but has never been evaluated, first synchronously evaluate the bodies of the module and any dependencies that have not evaluated yet.

When the `get` method is called with the argument `name`, the following steps are taken:

1. Let `loader` be this Loader.
2. ReturnIfAbrupt(`loader`).
3. Let `loaderRecord` be `loader`'s `[[LoaderRecord]]` internal slot.
4. Let `name` be `ToString(name)`.
5. ReturnIfAbrupt(`name`).
6. Let `modules` be the value of `loaderRecord.([[Modules]])`.
7. Repeat for each Record `[[key]], [[value]]` \( p \) that is an element of `modules`,
   a. If `SameValue(p.[[key]], name)` is true, then
      i. Let `module` be `p.[[value]]`.
      ii. Let `result` be `EnsureEvaluated(module, (), loaderRecord)`.
      iii. ReturnIfAbrupt(`result`).
     iv. Return `p.[[value]]`.
8. Return `undefined`. 
26.2.3.6 get Reflect.Loader.prototype.global

Reflect.Loader.prototype.global is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let loader be this Loader.
2. ReturnIfAbrupt(loader).
3. Let loaderRecord be loader’s [[LoaderRecord]] internal slot.
4. Let realm be the value of loaderRecord.[[Realm]].
5. Return realm.[[globalThis]].

26.2.3.7 Reflect.Loader.prototype.has ( name )

When the Reflect.Loader.prototype.has method is called with argument name, the following steps are taken:

1. Let loader be this Loader.
2. ReturnIfAbrupt(loader).
3. Let loaderRecord be loader’s [[LoaderRecord]] internal slot.
4. Let name be ToString(name).
5. ReturnIfAbrupt(name).
6. Let modules be the value of loaderRecord.[[Modules]].
7. Repeat for each Record {
4. foreach (p) {
5. If SameValue(p.[[key]], name) is true, then return true.
6. Return false.

NOTE This method does not call any hooks or run any module code.

26.2.3.8 Reflect.Loader.prototype.import ( name [, options ])

The import method asynchronously loads, links, and evaluates a module and all its dependencies if these actions have not already been performed. The argument name is the registry key for the module. import returns a Promise that resolves to the Module object once it has been committed to the registry and evaluated.

When the import method is called with argument name and optional arguments options the following steps are taken:

1. Let loader be this Loader.
2. ReturnIfAbrupt(loader).
3. Let loaderRecord be loader’s [[LoaderRecord]] internal slot.
4. Let p be the result of calling LoadModule(loaderRecord, name, options).
5. ReturnIfAbrupt(p).
6. Let F be a new function object as defined by EvaluateLoadedModule
7. Set F’s [[Loader]] internal slot to loaderRecord.
8. Let p be PromiseThen(p, F).

If the optional argument options is an object with an address property the string value of that property is used as the module location and module loading starts with the fetch step. If an address property is not present, module loading starts with the locate step.

The length property of the import method is 1.
NOTE Invoking the `import` method is the dynamic equivalent (when combined with normalization) of:

```
ImportDeclaration ::= import ModuleSpecifier ;
```

26.2.3.9 `Reflect.Loader.prototype.keys()`

The following steps are taken:
1. Let `loader` be this `Loader`.
2. ReturnIfAbrupt(`loader`).
3. Return the result of `CreateLoaderIterator(loader, "key")`.

26.2.3.10 `Reflect.Loader.prototype.load(name[, options])`

The `load` method asynchronously loads and links and all its dependencies if these actions have not already been performed. The argument `name` is the registry key for the module. `load` returns a Promise that resolves to the `Module` object once it has been committed to the registry.

When the `load` method is called with argument `name` and optional arguments `options` the following steps are taken:
1. Let `loader` be this `Loader`.
2. ReturnIfAbrupt(`loader`).
3. Let `loaderRecord` be `loader`’s `[[LoaderRecord]]` internal slot.
4. Let `p` be the result of calling `LoadModule(loaderRecord, name, options)`.
5. ReturnIfAbrupt(`p`).
6. Let `p` be `PromiseThen(p, %ReturnUndefined%)`.

If the optional argument `options` is an object with an `address` property. The string value of that property is used as the module location and module loading starts with the fetch step. If an `address` property is not present, module loading starts with the locate step.

The `length` property of the `load` method is 1.

NOTE The `load` method differs from the `import` method in that it does not force evaluation of the loaded module.

26.2.3.11 `Reflect.Loader.prototype.module(source[, options])`

The `module` method asynchronously loads, links, and evaluates an anonymous module from `source`. The module’s dependencies, if any, are loaded and committed to the registry. The anonymous module itself is not added to the registry. `module` returns a Promise object that resolves to a new `Module` object once the given module body has been evaluated.

When the `module` method is called with argument `source` and optional arguments `options` the following steps are taken:
1. Let `loader` be this `Loader`.
2. ReturnIfAbrupt(`loader`).
3. Let `loaderRecord` be `loader`’s `[[LoaderRecord]]` internal slot.
4. If `options` was not passed, then let `options` be `undefined`.
5. Let `address` be `GetOption(options, "address")`.
6. ReturnIfAbrupt(`address`).
7. Let load be CreateLoad(undefined).
8. Set load.([Address]) to address.
9. Let linkSet be CreateLinkSet(loaderRecord, load).
10. Let successCallback be a new function object as defined by EvaluateLoadedModule.
11. Set successCallback's ( [[Loader]] ) internal slot to loaderRecord.
12. Set successCallback's ( [[Load]] ) internal slot to load.
13. Let p be the result of calling PromiseThen(linkSet.([Done]), successCallback).
14. Let sourcePromise be PromiseOf(source).
15. Perform ProceedToTranslate(loaderRecord, load, sourcePromise).

The optional argument options is an object with an address property.

The length property of the module method is 1.

26.2.3.12 Reflect.Loader.prototype.newModule ( obj )

In the prototype this is the Module Factory Function. However, this factory seems to have only specialized utility and it seems to unnecessarily clutter the "global" namespace of Module abstractions. Making it a method of module loaders seems like a more sanity thing to do, but we can break it out if that's what people really want.

Also need to reconcile with are execute factory returns by the instantiate hook. Is this method intended to be able as an execute factory. If so it probably needs to accept multiple arguments.

When the newModule method is called with argument obj it creates a new Module objects whose export properties are derived form the properties of obj. The following steps are performed:

1. If Type(obj) is not Object, throw a TypeError exception.
2. Let mod be CreateLinkedModuleInstance( )
3. Let keys be the result of calling the ObjectKeys abstract operation passing obj as the argument.
4. ReturnIfAbrupt(keys).
5. For each key in keys, do
   a. Let value be the result of Get(obj, key).
   b. ReturnIfAbrupt(value).
   c. Let F be the result of calling CreateConstantGetter(key, value).
   d. Let desc be the PropertyDescriptor ( [[Configurable]]: false, [[Enumerable]]: true, [[Get]]: F, [[Set]]: undefined).
   e. Let status be the result of calling the DefinePropertyOrThrow abstract operation passing mod, key, and desc as arguments.
   f. ReturnIfAbrupt(status).
6. Call the ( [[PreventExtensions]] ) internal method of mod.
7. Return mod.

26.2.3.13 get Reflect.Loader.prototype.realm

Reflect.Loader.prototype.realm is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let loader be this Loader.
2. ReturnIfAbrupt(loader).
3. Let `loaderRecord` be `loader`'s `[[LoaderRecord]]` internal slot.
4. Return `RealmObjectFor(loaderRecord, [[Realm]])`.

26.2.3.14 `Reflect.Loader.prototype.set (name, module)`

Store a Module obj in this Loader's module registry, overwriting any existing entry with the same name.

The following steps are taken:
1. Let `loader` be this Loader.
2. ReturnIfAbrupt(`loader`).
3. Let `loaderRecord` be `loader`'s `[[LoaderRecord]]` internal slot.
4. Let `name` be `ToString(name)`.
5. ReturnIfAbrupt(`name`).
6. If `Type(module)` is not `Object`, throw a `TypeError` exception.
7. Let `modules` be the value of `loaderRecord. [[Modules]]`.
   a. If `SameValue(p. [[key]], name)` is `true`, then
      1. Set `p. [[value]]` to `module`.
      2. Return `loader`.
8. Let `p` be the Record `{[[key]]: name, [[value]]: module}`.
9. Append `p` as the last record of `loaderRecord. [[Modules]]`.
10. Return `loader`.

26.2.3.15 `Reflect.Loader.prototype.values()`

The following steps are taken:
1. Let `loader` be this Loader.
2. ReturnIfAbrupt(`loader`).
3. Return the result of `CreateLoaderIterator(loader, "value")`.

26.2.3.16 `Reflect.Loader.prototype[@@iterator]()`

The initial value of the `@@iterator` property is the same function object as the initial value of the `entries` property.

26.2.3.17 `Reflect.Loader.prototype[@@toStringTag]`

The initial value of the `@@toStringTag` property is the string value "Reflect.Loader".

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

26.2.3.18 `Reflect.Loader Pipeline Hook Properties`

Loader hooks are methods that are called at various points in the process of loading a module. `Reflect.Loader.prototype` provides default implementations for the hook methods. However, individual Loader object may over-ride these defaults using own properties.

26.2.3.18.1 `Reflect.Loader.prototype.normalize (name, refererName, refererAddress)`

When the `normalize` loader hook is called with arguments `name`, `refererName`, and `refererAddress loadRequest`, the following steps are taken:

Commented [AWB22142]: TODO: need to define. Lazily create Realm objects?
1. Assert: Type(name) is String.
2. Return name.

This is a Loader hook that may be over-ridden by an own property of Loader instances. The normalize hook is called once per distinct ModuleSpecifier String value in a ModuleBody, while the module ModuleBody with that is being loaded. The name argument is the StringValue of a ModuleSpecifier.

The normalize hook returns an eventual String, the normalized module name, which is used for the rest of the import process. In particular, the [[Loads]] and [[Modules]] Lists of a ModuleLinkage record are both keyed by normalized module names. The module registry contains at most one module for a given normalized module name.

After calling this hook, if the normalized module name is in the registry or the load table, no new LoadRecord is created. Otherwise the loader initiates a load for that module that starts by calling the locate hook.

26.2.3.18.2 Reflect.Loader.prototype.locate (loadRequest)

When the locate method is called with argument loadRequest the following steps are taken:

1. Return the result of Get(loadRequest, "name").

This is a Loader hook that may be over-ridden by an own property of Loader instances. The locate hook is called for each distinct normalized import ModuleSpecifier immediately after the normalize hook returns successfully, unless the module is already loaded or loading.

The locate hook is called to obtain to determine the Loader-dependent resource address (URL, path, etc.) corresponding to normalized module name. The resource address is used later in the Loader pipeline to retrieve the source code of the requested module.

When a locate hook is called by an Loader object the argument loadRequest is a LoadRequest object (15.2.3.2). The value of the name property is the normalized module name. The locate hook returns an eventual value that is used as the resource address. When the returned value is resolved, loading will continue with the fetch hook.

NOTE The System.locate hook typically is significantly more complicated than the default locate hook.

26.2.3.18.3 Reflect.Loader.prototype.fetch (loadRequest)

When the fetch loader hook is called with argument loadRequest, the following steps are taken:

1. Throw a TypeError exception.

This is a Loader hook that will normally be over-ridden by an own property of Loader instances. The fetch hook is called by a Loader for all modules whose source code was not directly provided to the Loader. It is also used to process the import keyword. The fetch hook is not called for module bodies directly provided as arguments to loader.module() or loader.define(). However, the fetch hook may be called when loading other modules imported by such modules.

When a fetch hook is called by an Loader object the argument loadRequest is a LoadRequest object (15.2.3.2) with an address property. The value of the address property identifies the module source code to fetch. The fetch hook returns an eventual String containing the source code of the module.

Commented [AWB22143]: So it's ok for multiple ModuleSpecifiers to normalize to the same name?
26.2.3.18.4 Reflect.Loader.prototype.translate (loadRequest)

When the translate method is called, the following steps are taken:

1. Return the result of Get(loadRequest, "source").

This is a Loader hook that may be over-ridden by an own property of Loader instances. The translate hook is called for each ModuleBody including those passed to loader.module() or loader.define(). The translate hook is called prior to parsing the ModuleBody and provides a Loader the opportunity to modify or replace the source code that will be parsed.

NOTE An example of the use of the translate hook would be to translate source code for another programming language into an ECMAScript ModuleBody.

When a translate hook is called by a Loader object the argument loadRequest is a LoadRequest object (15.2.3.2) with address and source properties. The value of the address property identifies the module source code to fetch. The value of the source property is the resolved value returned from the fetch hook. The translate hook returns either an eventual String value ECMAScript that will be parsed as a ModuleBody.

26.2.3.18.5 Reflect.Loader.prototype.instantiate (loadRequest)

When the instantiate loader hook is called with argument loadRequest, the following steps are taken:

1. Return undefined.

This hook allows a Loader to provide interoperability with other module systems.

When a instantiate hook is called by an Loader object the argument loadRequest is a LoadRequest object (15.2.3.2) with address and source properties. loadRequest.name, loadRequest.metadata, and loadRequest.address are the same values passed to the fetch and translate hooks. loadRequest.source is the value produced by the translate hook.

If the instantiate hook returns an eventual undefined, then the loader uses the default linking behaviour. It parses loadRequest.source as a Module, looks at its imports, loads its dependencies asynchronously, and finally links them together and adds them to the registry.

Otherwise, the instantiate hook must return an eventual instantiationRequest object. An instantiationRequest object has two required properties. The value of the deps property is an array of strings. Each string is the name of a module upon which the module identified by loadRequest has dependencies. The value of the execute property is a function which the loader will use to create the module and link it with its clients and dependencies. The function should expect to receive the same number of arguments as the size of the deps array and must return an eventual Module object. The arguments are Module objects and have a one-to-one correspondence with elements of the deps array.

The module is evaluated during the linking process. First all of the modules it depends upon are linked and evaluated, and then passed to the execute function. Then the resulting module is linked with the downstream dependencies.

NOTE This feature is provided in order to permit custom loaders to support using import to import pre-ES6 modules such as AMD modules. The design requires incremental linking when such modules are present, but it ensures that modules implemented with standard source-level module declarations can still be statically validated.

© Ecma International 2014
26.2.4 Properties of Reflect.Loader Instances

Loader instances are ordinary objects that inherit properties from the `%LoaderPrototype% intrinsic object. Loader instances each have a [[Loader]] internal slot whose value after initialization is the Loader Record that the Load instance reflects.

26.2.5 Loader Iterator Objects

A Loader Iterator object represents a specific iteration over the module registry of some specific Loader instance object. There is not a named constructor for Loader Iterator objects. Instead, Loader iterator objects are created by calling certain methods of Loader instance objects.

26.2.5.1 CreateLoaderIterator Abstract Operation

Several methods of Loader objects return Iterator objects. The abstract operation CreateLoaderIterator with arguments loader and kind is used to create such iterator objects. It performs the following steps:

1. Assert: loader is an initialized Loader instance object.
2. Let iterator be ObjectCreate(%LoaderIteratorPrototype%, ([Loader], [LoaderNextIndex], [LoaderIterationKind])).
3. Set iterator's [[Loader]] internal slot to loader.
4. Set iterator's [[LoaderNextIndex]] internal slot to 0.
5. Set iterator's [[LoaderIterationKind]] internal slot to kind.
6. Return iterator.

26.2.5.2 The `%LoaderIteratorPrototype% Object

All Loader Iterator Objects inherit properties from the `%LoaderIteratorPrototype% intrinsic object. The `%LoaderIteratorPrototype% intrinsic object is an ordinary object and its [[Prototype]] internal slot is the `%IteratorPrototype% intrinsic object (25.1.2). In addition, `%LoaderIteratorPrototype% has the following properties:

26.2.5.2.1 %LoaderIteratorPrototype%.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 Loader Iterator Instance (26.2.5.3), throw a TypeError exception.
4. Let m be the value of the [[Loader]] internal slot of O.
5. Let loaderRecord be m's [[LoaderRecord]] internal slot.
6. Let index be the value of the [[LoaderNextIndex]] internal slot of O.
7. Let itemKind be the value of the [[LoaderIterationKind]] internal slot of O.
8. If m is undefined, then return CreateIterResultObject(undefined, true).
9. Let entries be the List that is the value of loaderRecord.[[Modules]].
10. Repeat while index is less than the total number of elements of entries. The number of elements must be redetermined each time this method is evaluated.
   a. Let c be the Record {[[key]], [[value]]} that is the value of entries[index].
   b. Set index to index+1;
   c. Set the [[LoaderNextIndex]] internal slot of O to index.
   d. If c.[[key]] is not empty, then
      i. If itemKind is "key" then, let result be c.[[key]].
      ii. Else if itemKind is "value" then, let result be c.[[value]].
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 CreateIterResultObject(result, false).

NOTE Setting the [[Loader]] internal slot of O to undefined when the iterator is exhausted ensures that the same iterator cannot be restarted if new entries are subsequently added. This condition is tested in step 8.

26.2.5.2.2 %LoaderIteratorPrototype%[@@iterator] ()

The following steps are taken:
   1. Return the this value.

The value of the name property of this function is "[Symbol.iterator]".

26.2.5.2.3 %LoaderIteratorPrototype%@toStringTag

The initial value of the @@toStringTag property is the string value "Loader Iterator".

26.2.5.3 Properties of Loader Iterator Instances

Loader Iterator instances are ordinary objects that inherit properties from the %LoaderIteratorPrototype% intrinsic object. Loader Iterator instances are initially created with the internal slots described in Table 52.

Table 52 — Internal Slots of Loader Iterator Instances

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Loader]</td>
<td>The Loader object that is being iterated.</td>
</tr>
<tr>
<td>[LoaderNextIndex]</td>
<td>The integer index of the next Loader registry data element to be examined by this iterator.</td>
</tr>
<tr>
<td>[LoaderIterationKind]</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>

26.3 The System Object

The System object is the Loader Object instance associated with the Realm of the current global object.

26.4 Proxy Objects

26.4.1 The Proxy Constructor Function

The Proxy Constructor is a B
26.4.1 Proxy (target, handler)

The `Proxy` function is not intended to be directly called as a function. If it is called, the following steps are performed:

1. Throw a `TypeError` exception.

26.4.1.2 new Proxy (target, handler)

When `Proxy` is called as part of a `new` expression it is a constructor: it creates and initializes a new exotic proxy object. `Proxy` called as part of a new expression with arguments `target` and `handler` performs the following steps:

1. Return `ProxyCreate(target, handler)`.

If `Proxy` is implemented as an ECMAScript function object, it must have a `[[Construct]]` internal method that performs the above steps.

26.4.2 Properties of the Proxy Constructor Function

26.4.2.1 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 26.4.2.1.1.
4. Set the `[[RevokableProxy]]` internal slot of `revoker` to `p`.
5. Let `result` be `ObjectCreate(%ObjectPrototype%)`.
6. CreateDataProperty(`result`, "proxy", `p`).
7. CreateDataProperty(`result`, "revoke", `revoker`).
8. Return `result`.

26.4.2.1.1 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 :: See 10.1
    any Unicode code point

InputElementDiv :: See clause 11
    WhiteSpace
    LineTerminator
    Comment
    Token
    RightBracePunctuator
    DivPunctuator

InputElementRegExp :: See clause 11
    WhiteSpace
    LineTerminator
    Comment
    Token
    RightBracePunctuator
    RegularExpressionLiteral

InputElementTemplateTail :: See clause 11
    WhiteSpace
    LineTerminator
    Comment
    Token
    DivPunctuator
    TemplateSubstitutionTail
WhiteSpace ::
  \t<TAB>
  \v<VT>
  \f<FF>
  \p<SP>
  \b<NBSP>
  \u<BOM>
  \v<USP>

LineTerminator ::
  \n<LF>
  \r<CR>
  \v<LS>
  \f<PS>

LineTerminatorSequence ::
  \n<LF>
  \r<CR>
  [lookahead \n ≠ \n]
  \v<LS>
  \f<PS>
  \r<CR> \n
Comment ::
  MultiLineComment
  SingleLineComment

MultiLineComment ::
  / * MultiLineCommentCharsopt */

MultiLineCommentChars ::
  MultiLineNotAsteriskChar MultiLineCommentCharsopt
  * PostAsteriskCommentCharsopt

PostAsteriskCommentChars ::
  MultiLineNotForwardSlashOrAsteriskChar MultiLineCommentCharsopt
  * PostAsteriskCommentCharsopt

MultiLineNotAsteriskChar ::
  SourceCharacter but not *

MultiLineNotForwardSlashOrAsteriskChar ::
  SourceCharacter but not one of / or *

SingleLineComment ::
  / \ / SingleLineCommentCharsopt

SingleLineCommentChars ::
  SingleLineCommentChar SingleLineCommentCharsopt

SingleLineCommentChar ::
  SourceCharacter but not LineTerminator
Token ::

- IdentifierName
- Punctuator
- NumericLiteral
- StringLiteral
- Template

IdentifierName ::
- IdentifierStart
- IdentifierName IdentifierPart

IdentifierStart ::
- UnicodeIDStart
- $\backslash$ UnicodeEscapeSequence

IdentifierPart ::
- UnicodeIDContinue
- $\backslash$ UnicodeEscapeSequence
- \u{200d}
- \u{200f}

UnicodeIDStart ::
- any Unicode code point with the Unicode property "ID_Start" or "Other_ID_Start"

UnicodeIDContinue ::
- any Unicode code point with the Unicode property "ID_Continue", "Other_ID_Continue", or "Other_ID_Start"

ReservedWord ::
- Keyword
- FutureReservedWord
- NullLiteral
- BooleanLiteral

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

FutureReservedWord :: enum

See 11.5

See 11.6

See 11.6

See 11.6

See 11.6

See 11.6.2

See 11.6.2.1

See 11.6.2.2
The following tokens are also considered to be FutureReservedWords when parsing strict mode code (see 10.2.1).

<table>
<thead>
<tr>
<th>implements</th>
<th>package</th>
<th>protected</th>
<th>static</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface</td>
<td>private</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

DivPunctuator :: one of
/ /=

RightBracePunctuator :: one of
}

NullLiteral ::
null

BooleanLiteral ::
true
false

NumericLiteral ::
DecimalLiteral
BinaryIntegerLiteral
OctalIntegerLiteral
HexIntegerLiteral

DecimalLiteral ::
DecimalIntegerLiteral . DecimalDigits opt ExponentPart opt
. DecimalDigits ExponentPart opt
DecimalIntegerLiteral ExponentPart opt

DecimalIntegerLiteral ::
0
NonZeroDigit DecimalDigits opt

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

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

ExponentPart ::
  ExponentIndicator SignedInteger

ExponentIndicator :: one of
  e E

SignedInteger ::
  DecimalDigits
  + DecimalDigits
  - DecimalDigits

BinaryIntegerLiteral ::
  0b 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 HexDigit

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

See 11.8.3
StringLiteral :: See 11.8.4
   " DoubleStringCharacters opt "
   ' SingleStringCharacters opt '

DoubleStringCharacters :: See 11.8.4
   DoubleStringCharacter DoubleStringCharacters opt

SingleStringCharacters :: See 11.8.4
   SingleStringCharacter SingleStringCharacters opt

DoubleStringCharacter :: See 11.8.4
   SourceCharacter but not one of " or \ or LineTerminator
   \ EscapeSequence
   LineContinuation

SingleStringCharacter :: See 11.8.4
   SourceCharacter but not one of " or \ or LineTerminator
   \ EscapeSequence
   LineContinuation

LineContinuation :: See 11.8.4
   \ LineTerminatorSequence

EscapeSequence :: See 11.8.4
   CharacterEscapeSequence
   0 [lookahead a DecimalDigit]
   HexEscapeSequence
   UnicodeEscapeSequence

CharacterEscapeSequence :: See 11.8.4
   SingleEscapeCharacter
   NonEscapeCharacter

SingleEscapeCharacter :: See 11.8.4
   one of " \ b f n r t v

NonEscapeCharacter :: See 11.8.4
   SourceCharacter but not one of EscapeCharacter or LineTerminator

EscapeCharacter :: See 11.8.4
   SingleEscapeCharacter
   DecimalDigit
   \ u

HexEscapeSequence :: See 11.8.4
   \ HexDigit HexDigit

UnicodeEscapeSequence :: See 11.8.4
   \ u { HexDigits }
   \ u{ HexDigits }
Hex4Digits ::
    HexDigit HexDigit HexDigit HexDigit

RegularExpressionLiteral ::
    / RegularExpressionBody / RegularExpressionFlags

RegularExpressionBody ::
    RegularExpressionFirstChar RegularExpressionChars

RegularExpressionChars ::
    [empty]
    RegularExpressionChars RegularExpressionChar

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

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

RegularExpressionBackslashSequence ::
    \ RegularExpressionNonTerminator

RegularExpressionNonTerminator ::
    SourceCharacter but not LineTerminator

RegularExpressionClass ::
    [ RegularExpressionClassChars ]

RegularExpressionClassChars ::
    [empty]
    RegularExpressionClassChars RegularExpressionClassChar

RegularExpressionClassChar ::
    RegularExpressionNonTerminator but not one of ] or \ or [ ]
    RegularExpressionBackslashSequence

RegularExpressionFlags ::
    [empty]
    RegularExpressionFlags IdentifierPart

Template ::
    NoSubstitutionTemplate
    TemplateHead

NoSubstitutionTemplate ::
    " TemplateCharacters4+ 

TemplateHead ::
    "$ { 

Formated: German (Switzerland)
Field Code Changed
Formated: German (Switzerland)
Formated: German (Switzerland)
A.2 Expressions

PrimaryExpression :  
  this '
  Identifier Literal ArrayLiteral ObjectLiteral ( Expression )

ArrayLiteral :  
  [ Elisionopt ]
  [ ElementList ]
  [ ElementList , Elisionopt ]

ElementList :  
  Elisionopt AssignmentExpression ElementList , Elisionopt AssignmentExpression

Elision :  
  ,
  Elision ,

ObjectLiteral :  
  { }
  { PropertyDefinitionList }
  { PropertyDefinitionList , }

See 11.8.6

See 11.8.6

See 11.8.6

See 11.8.6

See 11.8.6

See 11.1

See 11.1.4

See 11.1.4

See 11.1.4

See 11.1.5
PropertyDefinitionList : See 11.1.5
  PropertyDefinition
  PropertyDefinitionList , PropertyDefinition

PropertyDefinition : See 11.1.5
  PropertyName : AssignmentExpression
  get PropertyName ( ) { FunctionBody }
  set PropertyName ( PropertySetParameterList ) { FunctionBody }

PropertyName : See 11.1.5
  IdentifierName
  StringLiteral
  NumericLiteral

PropertySetParameterList : See 11.1.5
  Identifier

MemberExpression : See 11.2
  PrimaryExpression
  FunctionExpression
  MemberExpression [ Expression ]
  MemberExpression . IdentifierName
  new MemberExpression Arguments

NewExpression : See 11.2
  MemberExpression
  new NewExpression

CallExpression : See 11.2
  MemberExpression Arguments
  CallExpression Arguments
  CallExpression [ Expression ]
  CallExpression . IdentifierName

Arguments : See 11.2
  ()
  ( ArgumentList )

ArgumentList : See 11.2
  AssignmentExpression
  ArgumentList , AssignmentExpression

LeftHandSideExpression : See 11.2
  NewExpression
  CallExpression
PostfixExpression:
  LeftHandSideExpression
  LeftHandSideExpression [no LineTerminator here] ++
  LeftHandSideExpression [no LineTerminator here] --

UnaryExpression:
  PostfixExpression
  delete UnaryExpression
  void UnaryExpression
typeof UnaryExpression
  ++ UnaryExpression
  -- UnaryExpression
  + UnaryExpression
  - UnaryExpression
  ~ UnaryExpression
  ! 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  
   EqualityExpression & EqualityExpression  

BitwiseXORExpression :  
   BitwiseANDExpression  
   BitwiseXORExpression ^ BitwiseANDExpression  

BitwiseORExpression :  
   BitwiseXORExpression  
   BitwiseXORExpression | BitwiseXORExpression  

LogicalANDExpression :  
   BitwiseORExpression  
   LogicalANDExpression && BitwiseORExpression  

LogicalORExpression :  
   LogicalANDExpression  
   LogicalORExpression || LogicalANDExpression  

ConditionalExpression :  
   LogicalORExpression  
   LogicalORExpression ? AssignmentExpression : AssignmentExpression  

AssignmentExpression :  
   ConditionalExpression  
   LeftHandSideExpression = AssignmentExpression  
   LeftHandSideExpression AssignmentOperator AssignmentExpression  

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

See 11.9  
See 11.10  
See 11.10  
See 11.11  
See 11.11  
See 11.12  
See 11.13  
See 11.13
Expression : See 11.14
  AssignmentExpression
  Expression , AssignmentExpression

A.3 Statements

Statement : See clause 12
  Block
  VariableStatement
  EmptyStatement
  ExpressionStatement
  IfStatement
  IterationStatement
  ContinueStatement
  BreakStatement
  ReturnStatement
  WithStatement
  LabelledStatement
  SwitchStatement
  ThrowStatement
  TryStatement
  DebuggerStatement

Block : See 12.1
  { StatementListopt }

StatementList : See 12.1
  Statement
  StatementList Statement

VariableStatement : See 12.2
  var VariableDeclarationList ;

VariableDeclarationList : See 12.2
  VariableDeclaration
  VariableDeclarationList , VariableDeclaration

VariableDeclaration : See 12.2
  Identifier Initializeropt

Initializer : See 12.2
  = AssignmentExpression

EmptyStatement : See 12.3
  ;
ExpressionStatement :
  [lookahead = [l, function]] Expression ;

IfStatement :
  if ( Expression ) Statement else Statement
  if ( Expression ) Statement

IterationStatement :
  do Statement while ( Expression );
  while ( Expression ) Statement
  for ( Expressionopt ; Expressionopt ; Expressionopt ) Statement
  for ( var VariableDeclarationList ; Expressionopt ; Expressionopt ) Statement
  for ( LeftHandSideExpression in Expression ) Statement
  for ( var VariableDeclaration in Expression ) Statement

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

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

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

WithStatement :
  with ( Expression ) Statement

SwitchStatement :
  switch ( Expression ) CaseBlock

CaseBlock :
  { CaseClausesopt }
  { CaseClausesopt DefaultClause CaseClausesopt }

CaseClauses :
  CaseClause
  CaseClauses CaseClause
CaseClause:
  case Expression : StatementListopt
    See 12.11

DefaultClause:
  default : StatementListopt
    See 12.11

LabelledStatement:
  Identifier : Statement
    See 12.12

ThrowStatement:
  throw [no LineTerminator here] Expression ;
    See 12.13

TryStatement:
  try Block Catch
  try Block Finally
  try Block Catch Finally
    See 12.14

Catch:
  catch ( Identifier ) Block
    See 12.14

Finally:
  finally Block
    See 12.14

DebuggerStatement:
  debugger ;
    See 12.15

A.4 Functions and Scripts

FunctionDeclaration:
  function Identifier ( FormalParameterListopt ) { FunctionBody }
    See clause 13

FunctionExpression:
  function Identifieropt ( FormalParameterListopt ) { FunctionBody }
    See clause 13

FormalParameterList:
  Identifier
    See clause 13
  FormalParameterList , Identifier

FunctionBody:
  SourceElementsopt
    See clause 13
Program : See clause 14
    SourceElements opt

SourceElements : See clause 14
    SourceElement
    SourceElements SourceElement

SourceElement : See clause 14
    Statement
    FunctionDeclaration

A.5 Number Conversions

StringNumericLiteral ::=
    StrWhiteSpace opt
    StrWhiteSpaceopt StringNumericLiteral StrWhiteSpace opt

StrWhiteSpace ::=
    StrWhiteSpaceChar StrWhiteSpaceopt

StrWhiteSpaceChar ::=
    LineTerminator
    WhiteSpace

StringNumericLiteral ::=
    StrDecimalLiteral
    HexIntegerLiteral

StrDecimalLiteral ::=
    StrUnsignedDecimalLiteral
    + StrUnsignedDecimalLiteral
    - StrUnsignedDecimalLiteral

StrUnsignedDecimalLiteral ::=
    Infinity
    DecimalDigits . DecimalDigits opt ExponentPart opt
    . DecimalDigits ExponentPart opt
    DecimalDigits ExponentPart opt

DecimalDigits ::=
    DecimalDigit
    DecimalDigits DecimalDigit

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

ExponentPart ::=
    ExponentIndicator SignedInteger

See 7.1.3.1
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

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
uriMark ::: one of See 15.1.3
- _ : ! ~ * ( )

A.7 Regular Expressions

Pattern :: See 15.10.1
Disjunction

Disjunction :: See 15.10.1
Alternative
Alternative | Disjunction

Alternative :: See 15.10.1
[empty]
Alternative Term

Term :: See 15.10.1
Assertion
Atom
Atom Quantifier

Assertion :: See 15.10.1
^ $ \ b \ B
( ? = Disjunction )
( ? ! Disjunction )

Quantifier :: See 15.10.1
QuantifierPrefix
QuantifierPrefix ?

QuantifierPrefix :: See 15.10.1
* + ?
{ DecimalDigits }
{ DecimalDigits , }
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
  abcdefghijklmnopqrstuvwxyz
  ABCDEFGHIJKLMNOPQRSTUVWXYZ

IdentityEscape ::
  SourceCharacter but not IdentifierPart
  <ZWJ>
  <ZWNJ>

DecimalEscape ::
  DecimalIntegerLiteral [lookahead \ DecimalDigit]

CharacterClassEscape :: one of
d D s S w W
CharacterClass ::
  [ \[lookahead e (*)\] 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
Annex B
(normative)

Additional ECMAScript Features for Web Browsers

The ECMAScript language syntax and semantics defined in this annex are required when the ECMAScript host is a web browser. The content of this annex is normative but optional if the ECMAScript host is not a web browser.

B.1 Additional Syntax

B.1.1 Numeric Literals

The syntax and semantics of 11.8.3 is extended as follows except that this extension is not allowed for strict mode code:

Syntax
NumericLiteral ::
    DecimalLiteral
    BinaryIntegerLiteral
    OctalIntegerLiteral
    HexIntegerLiteral
    LegacyOctalIntegerLiteral

LegacyOctalIntegerLiteral ::
    0 OctalDigit
    LegacyOctalIntegerLiteral OctalDigit

B.1.1.1 Static Semantics

- The MV of LegacyOctalIntegerLiteral :: 0 OctalDigit is the MV of OctalDigit.
- The MV of LegacyOctalIntegerLiteral :: LegacyOctalIntegerLiteral OctalDigit is (the MV of LegacyOctalIntegerLiteral times 8) plus the MV of OctalDigit.

B.1.2 String Literals

The syntax and semantics of 11.8.4 is extended as follows except that this extension is not allowed for strict mode code:

Syntax
EscapeSequence ::
    CharacterEscapeSequence
    LegacyOctalEscapeSequence
    HexEscapeSequence
    UnicodeEscapeSequence
LegacyOctalEscapeSequence ::
  OctalDigit [lookahead ≠ OctalDigit]
  ZeroToThree OctalDigit [lookahead ≠ OctalDigit]
  FourToSeven OctalDigit
  ZeroToThree OctalDigit OctalDigit

ZeroToThree :: one of
  0 1 2 3

FourToSeven :: one of
  4 5 6 7

This definition of EscapeSequence is not used when parsing TemplateCharacter (11.8.6).

B.1.2.1 Static Semantics

- The CV of EscapeSequence :: LegacyOctalEscapeSequence is the CV of the LegacyOctalEscapeSequence.
- The CV of LegacyOctalEscapeSequence :: OctalDigit is code unit whose value is the MV of the OctalDigit.
- The CV of LegacyOctalEscapeSequence :: ZeroToThree OctalDigit is the code unit whose value is (8 times the MV of the ZeroToThree) plus the MV of the OctalDigit.
- The CV of LegacyOctalEscapeSequence :: FourToSeven OctalDigit is the code unit whose value is (8 times the MV of the FourToSeven) plus the MV of the OctalDigit.
- The CV of LegacyOctalEscapeSequence :: ZeroToThree OctalDigit OctalDigit is the code unit whose value is (64 (that is, $8^2$) times the MV of the ZeroToThree) plus (8 times the MV of the first OctalDigit) plus the MV of the second OctalDigit.
- The MV of ZeroToThree :: 0 is 0.
- The MV of ZeroToThree :: 1 is 1.
- The MV of ZeroToThree :: 2 is 2.
- The MV of ZeroToThree :: 3 is 3.
- The MV of FourToSeven :: 4 is 4.
- The MV of FourToSeven :: 5 is 5.
- The MV of FourToSeven :: 6 is 6.
- The MV of FourToSeven :: 7 is 7.

B.1.3 HTML-like Comments

TODO See 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 of 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 \(\text{a}^+\) ::
  \(\text{~U}^+\) ExtendedTerm
  \(\text{~U}^+\) Assertion\(\text{~U}\)
  \(\text{~U}\) Atom\(\text{~U}\)
  \(\text{~U}\) Atom\(\text{~U}\) Quantifier

ExtendedTerm ::
  Assertion
  AtomNoBrace Quantifier
  Atom
  QuantifiableAssertion Quantifier

AtomNoBrace ::
  PatternCharacterNoBrace
    \ AtomEscape
    CharacterClass
    ( Disjunction )
    ( ? : Disjunction )

Atom\(\text{~U}\) ::
  PatternCharacter
    \ AtomEscape\(\text{~U}\)
    CharacterClass\(\text{~U}\)
    ( Disjunction )
    ( ? : Disjunction )

PatternCharacterNoBrace ::
  SourceCharacter but not one of
    ^ $ \ . * + ? ( ) [ ] { } |

PatternCharacter ::
  SourceCharacter but not one of
    ^ $ \ . * + ? ( ) [ ] { } |

QuantifiableAssertion ::
  ( ? = Disjunction )
  ( ? ! Disjunction )

Assertion\(\text{~U}\) ::
  ^ $ \ b \ B
  \(\text{~U}\) ( ? = Disjunction\(\text{~U}\) )
  \(\text{~U}\) ( ? ! Disjunction\(\text{~U}\) )
  \(\text{~U}\) QuantifiableAssertion
AtomEscape₁₄ ::
  [+U] DecimalEscape
  [-U] DecimalEscape but only if the integer value of DecimalEscape is <= NCapturingParens
  [+U] CharacterEscape₁₄
  [+U] CharacterClassEscape
  [-U] CharacterEscape

CharacterEscape₁₄ ::
  ControlEscape
  c ControlLetter
  HexEscapeSequence
  RegExpUnicodeEscapeSequence
  [+] LegacyOctalEscapeSequence
  IdentityEscape

IdentityEscape₁₄ ::
  [+U] SyntaxCharacter
  [-U] SourceCharacter but not c
  [-U] <ZWJ>
  [-U] <ZWNJ>

NonemptyClassRanges₁₄ ::
  ClassAtom₁₃
  ClassAtom₁₃ NonemptyClassRangesNoDash₁₃
  [+U] ClassAtom₁₃ - ClassAtom₁₃ ClassRanges₁₃
  [-U] ClassAtomInRange - ClassAtomInRange ClassRanges

NonemptyClassRangesNoDash₁₃ ::
  ClassAtom₁₃
  ClassAtomNoDash₁₃ NonemptyClassRangesNoDash₁₃
  [+U] ClassAtomNoDash₁₃ - ClassAtomNoDash₁₃ ClassRanges₁₃
  [-U] ClassAtomNoDashInRange - ClassAtomNoDashInRange ClassRanges

ClassAtom₁₃ ::
  ~

ClassAtomNoDash₁₃ ::
  SourceCharacter but not one of \ or ] or -

ClassAtomInRange ::
  ~

ClassAtomNoDashInRange ::
  SourceCharacter but not one of \ or ] or -
  \ ClassEscape₁₉
  \ IdentityEscape
ClassEscape \[ U \] ::
[4U] DecimalEscape
[[-U] DecimalEscape but only if the integer value of DecimalEscape is \( \leq NCapturingParens \)

b
[4U] CharacterEscape
[4U] CharacterClassEscape
[-U] CharacterClassEscape
[-U] CharacterEscape

B.1.4.1 Pattern Semantics

The semantics of 21.2.2 is extended as follows:

Within 21.2.2.5 reference to "Atom :: ( \( \text{Disjunction} \) )" are to be interpreted as meaning "Atom :: ( \( \text{Disjunction} \) ) or AtomNoBrace :: ( \( \text{Disjunction} \) )".

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 PatternCharacterNoBrace.
2. Let \( A \) be a one-element CharSet containing the character \( ch \).
3. Call CharacterSetMatcher(\( A \), false) and return its Matcher result.

CharacterEscape (21.2.2.10) includes the following additional evaluation rule:

The production CharacterEscape :: LegacyOctalEscapeSequence evaluates by evaluating the CV of the LegacyOctalEscapeSequence (see B.1.2) and returning its character result.

ClassAtom (21.2.2.17) includes the following additional evaluation rules:

The production ClassAtomInRange :: - evaluates by returning the CharSet containing the one character -.

The production ClassAtomInRange :: ClassAtomNoDashInRange evaluates by evaluating ClassAtomNoDashInRange to obtain a CharSet and returning that CharSet.

ClassAtomNoDash (21.2.2.18) includes the following additional evaluation rules:

The production ClassAtomNoDashInRange :: SourceCharacter but not one of \( \backslash \) or \( \} \) or - evaluates by returning a one-element CharSet containing the character represented by SourceCharacter.

The production ClassAtomNoDashInRange :: \( \backslash \) ClassEscape but only if..., evaluates by evaluating 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 Built-in 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 code units have been replaced by a hexadecimal escape sequence.

For those code units being replaced whose value is \( U+00\text{FF} \) or less, a two-digit escape sequence of the form \( \%xx \) is used. For those characters being replaced whose code unit value is greater than \( U+00\text{FF} \), a four-digit escape sequence of the form \( \%u\text{xxxx} \) is used.

When the \( \text{escape} \) function is called with one argument \( \text{string} \), the following steps are taken:

1. Let \( \text{string} \) be \( \text{ToString}(\text{string}) \).
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 \( \text{char} \) is the code point of one of the 69 nonblank code units in
      "ABCDEFHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789@*_-./",
      then:
      i. Let \( S \) be a String containing the single code unit \( \text{char} \).
   c. Else if \( \text{char} > 256 \),
      i. Let \( S \) be a String containing six code units "\%u\text{wxyz}\" where \( \text{wxyz} \) are the code units of the
         four hexadecimal digits encoding the value of \( \text{char} \).
   d. Else, \( \text{char} < 256 \),
      i. Let \( S \) be a String containing three code units "\%xy\" where \( \text{xy} \) are the code units of two
         hexadecimal digits encoding the value of \( \text{char} \).
      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 \( \text{unescape} \) 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 \( \text{escape} \) function is replaced with the code unit 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.  
   c. Let `R` be a new `String` value computed by concatenating the previous value of `R` and `c`.  
   d. Increase `k` by 1.  
7. 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 `RequireObjectCoercible(this value)`.  
2. ReturnIfAbrupt(`O`).  
3. If `Type(proto)` is neither `Object` nor `Null`, then return `undefined`.  
4. If `Type(O)` is not `Object`, then return `undefined`.  
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 `undefined`.  

© Ecma International 2014
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 position `start` and running for `length` code units (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 `RequireObjectCoercible(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 code units 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 code units from `S` beginning with the code unit 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 `CreateHTML(S, "a", "name", name)`.

B.2.3.2.1 CreateHTML (string, tag, attribute, value) Abstract Operation

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 `RequireObjectCoercible(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 `ToString(value)`.
   b. ReturnIfAbrupt(`V`).
   c. Let `escapedV` be the string value that is the same as `V` except that each occurrence of the code unit U+0022 (QUOTATION MARK) in `V` has been replaced with the six code unit sequence "&quot;".
   d. Let `p1` be the string value that is the concatenation of the following string values:
      ● The string value of `p1`
      ● Code unit U+0020 (SPACE)
• The string value of `attribute`
• Code unit U+003D (EQUALS SIGN)
• Code unit U+0022 (QUOTATION MARK)
• The string value of `escapedV`
• Code unit U+0022 (QUOTATION MARK)

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 p3, "</tag">.

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 `CreateHTML(S, "big", "", ")`.

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 `CreateHTML(S, "blink", "", ")`.

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 `CreateHTML(S, "b", "", ")`.

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 `CreateHTML(S, "tt", "", ")`.

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 `CreateHTML(S, "font", "color", 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 `CreateHTML(S, "font", "size", 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 CreateHTML(\( S \), "i", "", "").

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 CreateHTML(\( S \), "a", "href", \( url \)).

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 CreateHTML(\( S \), "small", "", "").

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 CreateHTML(\( S \), "strike", "", "").

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 CreateHTML(\( S \), "sub", "", "").

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 CreateHTML(\( S \), "sup", "", "").

B.2.4 Additional Properties of the Date.prototype Object

B.2.4.1 Date.prototype.getFullYear ()

NOTE  The getFullYear method is preferred for nearly all purposes, because it avoids the “year 2000 problem.”

When the getFullYear method is called with no arguments, the following steps are taken:
1. Let \( t \) be this time value.
2. ReturnIfAbrupt(\( t \)).
3. If \( t \) is NaN, return NaN.
4. Return YearFromTime(LocalTime(\( t \)) – 1900).
B.2.4.2 Date.prototype.setYear (year)

NOTE The setFullYear method is preferred for nearly all purposes, because it avoids the "year 2000 problem."

When the setYear method is called with one argument year, the following steps are taken:

1. Let t be LocalTime(this time value); but if this time value is NaN, let t be +0.
2. Let y be ToNumber(year).
3. ReturnIfAbrupt(y).
4. If y is NaN, set the [[DateValue]] internal slot of this Date object to NaN and return NaN.
5. If y is not NaN and 0 ≤ ToInteger(y) ≤ 99 then let yyyy be ToInteger(y) + 1900. Otherwise, let yyyy be y.
6. Let d be MakeDay(yyyy, MonthFromTime(t), DateFromTime(t)).
7. Let date be UTC(MakeDate(d, TimeWithinDay(t))).
8. Set the [[DateValue]] internal slot of this Date object to TimeClip(date).
9. Return the value of the [[DateValue]] internal slot of this Date object.

B.2.4.3 Date.prototype.toGMTString ()

NOTE The property toString is preferred. The toGMTString property is provided principally for compatibility with old code. It is recommended that the toString 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.toString.

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 [[RegExpMatcher]] internal slot, then
   a. Throw a TypeError exception.
3. If Type(pattern) is Object and pattern has a [[RegExpMatcher]] internal slot, then
   a. If the value of pattern's [[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 [[OriginalSource]] internal slot.
   d. Let F be the value of pattern's [[OriginalFlags]] internal slot.
4. Else,
   a. Let P be pattern.
   b. Let F be flags.

NOTE The compile method completely reinitializes 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 Initializers

The following Early Error rule is added to those in 12.2.5.1:

```
ObjectLiteral : { PropertyDefinitionList }
and
ObjectLiteral : { PropertyDefinitionList , }
```

- It is a Syntax Error if PropertyNameList of PropertyDefinitionList contains any duplicate entries for "__proto__" and at least two of those entries were obtained from productions of the form PropertyDefinition : PropertyName : AssignmentExpression.

In 12.2.5.9 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. If IsFunctionDefinition(AssignmentExpression) is true, then
   a. Assert: propValue is an ECMAScript function object.
   b. Let referencesSuper be the value of propValue’s [[NeedsSuper]] internal slot.
   c. Let thisMode be the value of propValue’s [[ThisMode]] internal slot.
   d. If thisMode is not lexical and referencesSuper is true, then
      i. If propValue’s [[HomeObject]] internal slot is undefined, then
         1. Assert: AssignmentExpression is not a class definition whose constructor references super.
         2. Set the propValue’s [[HomeObject]] internal slot to object.
         3. Set the propValue’s [[MethodName]] internal slot to propKey.
   e. If IsAnonymousFunctionDefinition(AssignmentExpression) is true, then
      i. SetFunctionName(propValue, propKey).
      ii. Assert: SetFunctionName will not return an abrupt completion.
8. Let desc be the Property Descriptor{[[Value]]: propValue, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}
9. Return DefinePropertyOrThrow(object, propKey, desc).
```
B.3.2 Labelled Function Declarations

Prior to the Sixth Edition, the ECMAScript specification LabelledStatement did not allow for the association of a statement label with a FunctionDeclaration. However, a labelled FunctionDeclaration was an allowable extension for non-strict mode code and most browser-hosted ECMAScript implementations supported that extension. In the Sixth Edition, the grammar productions for LabelledStatement permits use of FunctionDeclaration as a LabelledItem but 13.12.1 includes an Early Error rule that produces a Syntax Error if that occurs. For web browser compatibility, that rule is modified with the addition of the underlined text:

LabelledItem : FunctionDeclaration
  • It is a Syntax Error if any strict mode source code matches this rule.

B.3.3 Block-Level Function Declarations Web Legacy Compatibility Semantics

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. Unfortunately, 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 implementation 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 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 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 semantics of Block level function declarations provided by ECMA-262 Edition 6. Any pre-existing ECMAScript code that employees that use case will operate using the Block level function declarations semantics defined by clauses 9.13, and 14 of this specification.

Sixth edition interoperability for the second and third use cases requires the following extensions to the clause 9 and clause 14 semantics. These extensions are applied to each non-strict mode function \( g \) for each FunctionDeclaration \( f \) that is directly contained in the StatementList of a Block, CaseClause, or DefaultClause that is part of the function code of \( g \).

1. Let \( F \) be StringValue of the BindingIdentifier of FunctionDeclaration \( f \).
2. If replacing the FunctionDeclaration \( f \) with a VariableStatement that has \( F \) as a BindingIdentifier would not produce any Early Errors for \( g \), then
   a. During FunctionDeclarationInstantiation (9.2.13) for \( g \) perform the following steps immediately before performing step 25:
      i. NOTE A var binding for \( F \) is only instantiated here if it is not also a VarDeclaredName, the name of a formal parameter, or another FunctionDeclarations.
      ii. If neither parameterNames nor instantiatedVarNames contains \( F \), then
         1. Let \( status \) be the result of calling bodyEnv’s CreateMutableBinding concrete method passing \( F \) as the argument.
         2. Assert: \( status \) is never an abrupt completion.
         3. Call the InitializeBinding concrete method of bodyEnv with arguments \( F \) and undefined.
         4. Append \( F \) to instantiatedVarNames.
   b. In place of the FunctionDeclaration Evaluation algorithm provide in 14.1.22, perform the following steps to evaluate the FunctionDeclaration \( f \):
      1. Let \( fenv \) be the running execution context’s VariableEnvironment.
      2. Let \( benv \) be the running execution context’s LexicalEnvironment.
      3. Let \( fo bj \) be the result of calling the GetBindingValue concrete method of \( benv \) with arguments \( F \) and false.
      4. ReturnIfAbrupt(\( fo bj \)).
      5. Let \( status \) be the result of calling \( fenv \)’s SetMutableBinding concrete method with arguments \( F \), \( fo bj \), and false.
      6. Assert: \( status \) is never an abrupt completion.
      7. Return NormalCompletion(\emptyset).

If an ECMAScript implementation has a mechanism for reporting diagnostic warning messages, a warning should be produced for each function \( g \) whose function code contains a FunctionDeclaration for which steps 2.a and 2.b above will be performed.

B.3.4 FunctionDeclarations in IfStatement Statement Clauses.

The following rules for IfStatement augment those in 13.5:

```javascript
IfStatement[\( \gamma \text{End} \) \( \text{Value} \) \( \text{Statement} \text{End} \text{Value} \text{Statement} \text{End} ]:
  if ( \text{Expression}[\( \gamma \text{End} ] ) \text{FunctionDeclaration}[\( \gamma \text{End} \) \( \text{elseStatement} \text{End} \text{Value} \text{Statement} \text{End} ]
  if ( \text{Expression}[\( \gamma \text{End} ] ) \text{Statement}[\( \gamma \text{End} \) \( \text{elseFunctionDeclaration} \text{End} ]
  if ( \text{Expression} [\( \gamma \text{End} ] ) \text{FunctionDeclaration}[\( \gamma \text{End} ] \text{elseFunctionDeclaration} \text{End} ]
  if ( \text{Expression} [\( \gamma \text{End} ] ) \text{FunctionDeclaration}[\( \gamma \text{End} ]
```

The above rules are only applied when parsing non-strict mode code. If any non-strict code is match by one of these rules subsequent processing of that code takes places as if each matching occurrence of FunctionDeclaration was the sole StatementListItem of a BlockStatement occupying that position in the
source code. The semantics of such a synthetic *BlockStatement* includes the web legacy compatibility semantics specified in B.3.3.

### B.3.5 VariableStatements in Catch blocks

The content of subclause 13.14.1 is replaced with the following:

**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*, unless that element is only bound by a *VariableStatement* or the VariableDeclarationList of a for statement, or the ForBinding of a for-in statement.

**NOTE** The *Block* of a *Catch* clause may contain *var* declarations that bind a name that is also bound by the *CatchParameter*. At runtime, such bindings are instantiated in the VariableDeclarationEnvironment. They do not shadow the same-named bindings introduced by the *CatchParameter* and hence the initializer of such *var* declarations will assign to the corresponding catch parameter rather than the *var* binding. The relaxation of the normal static semantic rule does not apply to names only bound by for-of statements.
Annex C
(Informative)

The Strict Mode of ECMAScript

The strict mode restriction and exceptions

- Use of the identifiers "implements", "interface", "package", "private", "protected", "public", and "static" is prohibited 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}, or to a non-existent property of an object whose [[Extensible]] internal slot has the value false. In these cases a TypeError exception is thrown (12.14).
- The identifier eval or arguments may not appear as the LeftHandSideExpression of an Assignment operator (12.14) or of a PostfixExpression (12.14) or as the UnaryExpression operated upon by a Prefix Increment (12.5.7) or a Prefix Decrement (12.5.8) 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).
- 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. (9.2.13).
- It is a SyntaxError if the IdentifierName eval or the IdentifierName arguments occurs as a BindingIdentifier within strict mode code (12.1.1).
- 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, 12.2.1, 19.2.3.1, 19.2.3.3).

Commented [AWB18148]: This entire section needs to be updated to include strict mode restrictions that apply to new ES6 features.
• 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 (12.5.4).

• When a delete operator occurs within strict mode code, a TypeError is thrown if the property to be deleted has the attribute { [[Configurable]]: false } (12.5.4).

• 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 same BindingIdentifier appears more than once in the FormalParameters of a strict mode function. An attempt to create such a function using a Function or Generator constructor is a SyntaxError (14.1.2, 19.2.1, 25.2.1).

• 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 (9.2.1, 9.4.4).

• It is a SyntaxError if the IdentifierName eval or the IdentifierName arguments occurs as the BindingIdentifier of a FunctionDeclaration, FunctionExpression, GeneratorDeclaration, or GeneratorExpression (12.1.1, 14.1.2, 14.4.1).
Annex D
(informative)

Corrections and Clarifications with Possible Compatibility Impact

D.1 In Edition 6

9.1.4.2.1, 9.1.4.2.2: The 5th Edition moved the capture of the current array length prior to the integer conversion of the array index or new length value. However, the captured length value could become invalid if the conversion process has the side-effect of changing the array length. The 6th Edition specifies that the current array length must be captured after the possible occurrence of such side-effects.

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 behaviour 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 "/".

D.2 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.

10.2.1.3: 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.

15.3.3.12: 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.i.ii and 3.i.iii the variable name was inverted resulting in an incorrectly inverted test.

15.5.4.9: Normative requirement concerning canonically equivalent strings deleted from paragraph following algorithm because it is listed as a recommendation in NOTE 2.

15.5.4.14: In split algorithm step 11.a and 13.a, the positional order of the arguments to SplitMatch was corrected to match the actual parameter signature of SplitMatch. In step 13.a.iii.7.d, lengthA replaces A.length.
15.5.5.2: In first paragraph, removed the implication that the individual character property access had “array index” semantics. Modified algorithm steps 3 and 5 such that they do not enforce “array index” requirement.

15.9.1.15: Specified legal value ranges for fields that lacked them. Eliminated “time-only” formats. Specified default values for all optional fields.

15.10.2.2: The step numbers of the algorithm for the internal closure produced by step 2 were incorrectly numbered in a manner that implied that they were steps of the outer algorithm.

15.10.2.6: In the abstract operation IsWordChar the first character in the list in step 3 is “a” rather than “A”.

15.10.2.8: In the algorithm for the closure returned by the abstract operation CharacterSetMatcher, the variable defined by step 3 and passed as an argument in step 4 was renamed to ch in order to avoid a name conflict with a formal parameter of the closure.

15.10.6.2: Step 9.e was deleted because it performed an extra increment of i.

15.11.1.1: Removed requirement that the message own property is set to the empty String when the message argument is undefined.

15.11.1.2: Removed requirement that the message own property is set to the empty String when the message argument is undefined.

15.11.4.4: Steps 6-10 modified/adDED to correctly deal with missing or empty message property value.

15.12.3: 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.

D.3 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 `ArrayInitializer` 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.
Annex E
(informative)

Additions and Changes That Introduce Incompatibilities with Prior Editions

E.1 In the 6th Edition

11: In Edition 6, Function calls are not allowed to return a Reference value.

12.2.5.1: Early error relating to duplicate property names in Object Initializers have been eliminated in Edition 6.

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 initialization 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 initializer.

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 Initializer 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 make 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 make 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 `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 `TypeError` to be thrown.

19.1.2.14: In Edition 6, if the argument to `Object.keys` 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.

19.1.2.15: In Edition 6, if the argument to `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 `TypeError` to be thrown.

19.1.2.17: In Edition 6, if the argument to `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 `TypeError` to be thrown.

19.2.4.1: In Edition 6, the `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 `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 `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 RegExp prototype object is not a RegExp instance. In previous editions it was a RegExp instance whose pattern was the empty string.

21.2.5 In Edition 6, 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.

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 controls are no longer stripped from ECMAScript source text before processing. In Edition 5, if such a character appears in a `StringLiteral` or `RegularExpressionLiteral` the character will be incorporated into the literal where in Edition 3 the character would not be incorporated into the literal.

7.2: Unicode character `<BOM>` is now treated as whitespace and its presence in the middle of what appears to be an identifier could result in a syntax error which would not have occurred in Edition 3.

7.3: Line terminator characters that are preceded by an escape sequence are now allowed within a string literal token. In Edition 3 a syntax error would have been produced.

7.8.5: Regular expression literals now return a unique object each time the literal is evaluated. This change is detectable by any programs that test the object identity of such literal values or that are sensitive to the shared side effects.

7.8.5: Edition 5 requires early reporting of any possible RegExp constructor errors that would be produced when converting a `RegularExpressionLiteral` to a RegExp object. Prior to Edition 5 implementations were permitted to defer the reporting of such errors until the actual execution time creation of the object.

7.8.5: In Edition 5 unescaped `/` characters may appear as a `CharacterClass` in a regular expression literal. In Edition 3 such a character would have been interpreted as the final character of the literal.

10.4.2: In Edition 5, indirect calls to the `eval` function use the global environment as both the variable environment and lexical environment for the eval code. In Edition 3, the variable and lexical environments of the caller of an indirect `eval` was used as the environments for the eval code.

15.4.3: In Edition 5 all methods of `Array.prototype` are intentionally generic. In Edition 3 `toString` and `toLocaleString` were not generic and would throw a `TypeError` exception if applied to objects that were not instances of Array.

10.6: In Edition 5 the array indexed properties of argument objects that correspond to actual formal parameters are enumerable. In Edition 3, such properties were not enumerable.
10.6: In Edition 5 the value of the [[Class]] internal slot of an arguments object is "Arguments". In Edition 3, it was "Object". This is observable if toString is called as a method of an arguments object.

12.6.4: for-in statements no longer throw a TypeError if the in expression evaluates to null or undefined. Instead, the statement behaves as if the value of the expression was an object with no enumerable properties.

15: In Edition 5, the following new properties are defined on built-in objects that exist in Edition 3:
Object.getPrototypeOf, Object.getOwnPropertyDescriptor, Object.getOwnPropertyNames, Object.create, Object.defineProperty, Object.preventExtensions, Object.isSealed, Object.isFrozen, Object.isExtensible, Object.keys, Function.prototype.bind, Array.prototype.indexOf, Array.prototype.lastIndexOf, Array.prototype.map, Array.prototype.filter, Array.prototype.reduce, Array.prototype.reduceRight, String.prototype.trim, Date.now, Date.prototype.toISOString, Date.prototype.toJSON.

15: Implementations are now required to ignore extra arguments to standard built-in methods unless otherwise explicitly specified. In Edition 3, the handling of extra arguments was unspecified and implementations were explicitly allowed to throw a TypeError exception.

15.1.1: The value properties NaN, Infinity, and undefined of the Global Object have been changed to be read-only properties.

15.1.2.1. Implementations are no longer permitted to restrict the use of eval in ways that are not a direct call. In addition, any invocation of eval that is not a direct call uses the global environment as its variable environment rather than the caller’s variable environment.

15.1.2.2: The specification of the function parseInt no longer allows implementations to treat Strings beginning with a 0 as octal values.

15.3.3.3: In Edition 3, a TypeError is thrown if the second argument passed to 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. 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, \a 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.
Bibliography


